Génération automatique de tests logiciels dans le contexte de la certification aéronautique

Clothilde Jeangoudoux

SORBONNE UNIVERSITÉ

École doctorale Informatique, Télécommunications et Électronique (Paris)

THÈSE

pour obtenir le grade de

DOCTEUR

Spécialité : Informatique

Présentée par

Clothilde JEANGOUDOUX

Génération Automatique de Tests Logiciels dans le Contexte de la Certification Aéronautique

---

Laboratoire d'informatique de Paris 6 (UMR 7606)

Thèse CIFRE en collaboration avec **Safran Electronics & Defense**

Soutenance publique prévue le 21 mai 2019 devant le jury composé de :

Rapporteurs :	M.	Jean-Michel MULLER	Directeur de recherche CNRS
	M.	Gilles TROMBETTONI	Professeur à l'université de Montpellier
Examinateurs :	Mme.	Sylvie BOLDO	Directrice de recherche Inria
	M.	Fabrice KORDON	Professeur à Sorbonne Université
Directeur de thèse :	M.	Stef GRAILLAT	Professeur à Sorbonne Université
Encadrants :	M.	Christoph LAUTER	Assistant Professor à University of Alaska
	M.	Fabrice LARRIBE	Ingénieur à Safran Electronics & Defense

TABLE DES MATIÈRES

LISTE DES FIGURES

LISTE DES TABLEAUX

LISTE DES ALGORITHMES

INTRODUCTION

LES avions, les hélicoptères ou encore les drones sont des témoins des progrès de la technologie dans les domaines de la conception des matériaux, de l'électronique embarquée ou encore de l'informatique. En 2017, l'Organisation de l'aviation civile internationale (OACI) estime que le trafic aérien mondial a transporté 4.1 milliards de passagers sur environ 20.5 millions de vols dont 88 accidents, d'après ses statistiques annuelles mondiales [1]. Les accidents sont généralement causés par une combinaison de facteurs, pouvant être relatives aux conditions météorologiques, aux décisions du pilote ou aux défaillances mécaniques. Les accidents sont peu nombreux relativement au nombre de vol quotidien, mais souvent mortels et très médiatisés, comme l'a été par exemple l'accident du Boeing 737 d'Ethiopian Airlines le 10 mars 2019 [2].

Les concepteurs d'avions et d'équipement aéronautique sont donc confrontés à un certain nombre de problématiques relatives au développement et à la validation de leurs produits. Comment développer un produit aéronautique de confiance ? Comment évaluer la confiance que l'on peut avoir dans ce produit aéronautique ? À quel moment peut-on dire que l'on est suffisamment confiant dans son fonctionnement pour intégrer ce produit dans un avion transportant des passagers civils ? Voici les questions auxquelles nous nous intéresserons dans cette thèse, en nous penchant sur une étape particulière du cycle de développement d'un produit logiciel embarqué : la validation et la vérification par le test.

Certification aéronautique d'un logiciel embarqué

Du pilote automatique au système de gestion des moteurs en passant par le système de divertissement multimédia, les logiciels embarqués sont omniprésents dans les avions commerciaux récents. Une erreur logicielle, résultant d'un comportement non prévu ou maîtrisé lors du processus de développement et de validation du logiciel embarqué peut

1. ICAO 2017 annual report
2. Aviation Safaty Network – Boeing 737 Ethiopian Airlines

se propager dans le reste de l'avion et engendrer une défaillance critique au niveau de l'appareil, mettant en péril son intégrité et celle de ses passagers.

Les industriels développant des équipements aéronautiques s'engagent légalement à fournir un produit estimé suffisamment sûr pour être autorisé à voler dans un espace aérien public ainsi qu'à transporter des passagers civils. En France, la Direction générale de l'Aviation civile (DGAC) délivre un certificat de navigabilité, document autorisant un avion produit par des industriels à voler. La certification de navigabilité environnementale d'un aéronef ainsi que ses pièces et équipements permet d'assurer qu'un avion sera en mesure de voler quelles que soient les conditions éprouvées conformément à la législation. Pour faciliter et guider le processus de certification, les agences gouvernementales chargées de la sécurité des espaces aériens, comme la Federal Aviation Administration (FAA) pour les États-Unis et l'European Aviation Safety Agency (EASA) pour l'Union Européenne, pourvoient un certain nombre de recommandations pratiques sous la forme du document *DO–178C, Software considerations in airborne systems and equipment certification* [1]. D'après ces préconisations, il est nécessaire de fournir un document de spécification fonctionnelle décrivant par une série d'exigences le comportement fonctionnel que doit satisfaire le logiciel développé. La validation du logiciel est ensuite effectuée par un certain nombre d'activités, dont la vérification par le test du comportement fonctionnel du logiciel par rapport à celui décrit dans les exigences.

Les codes numériques en particulier peuvent être sujets à des erreurs pouvant avoir des conséquences importantes sur le fonctionnement désiré d'un logiciel. Lors du décollage par exemple, le logiciel peut prendre en charge le calcul d'un certain nombre de paramètres en fonction des mesures effectuées par les instruments à bord. Les variables calculées dans le modèle mathématique déterminant le comportement fonctionnel du logiciel sont calculées avec des nombres réels. Ces nombres ne peuvent pas être représentés exactement sur un ordinateur qui ne dispose que d'une mémoire finie. Les valeurs réelles sont donc représentées sur ordinateur par des valeurs approchées à une certaine précision. Les erreurs dues à la représentation des variables numériques peuvent se combiner et s'amplifier pendant l'exécution du programme, produisant une sortie aberrante. Fort heureusement, la propagation des erreurs est un phénomène bien connu en informatique, et de nombreuses méthodes ont été élaborées pour quantifier et maîtriser ces erreurs dans un logiciel.

Imaginons qu'un logiciel calcule la fonction exponentielle, son comportement fonctionnel peut ainsi être traduit par $y = e^x$. Le logiciel implémentant cette exigence va exécuter un certain nombre d'instructions afin de renvoyer un résultat approchant le résultat exact y. Il est cependant impossible de calculer cette valeur exacte à l'aide d'un ordinateur. En effet, les valeurs représentées sur un ordinateur sont soumises à des contraintes de capacité mémoire, et sont ainsi représentées avec une précision finie. Prenons par exemple $x = 1.0$ et supposons qu'un ordinateur puisse représenter des valeurs décimales sur 3 chiffres signi-

ficatifs. Le résultat de l'exponentielle $y = e^{1.0}$ renvoyer par le logiciel sera donc $y = 2.71$ ou $y = 2.72$ selon l'arithmétique utilisée. Dans les deux cas, une erreur survient inévitablement entre le produit du logiciel et le résultat exact.

La philosophie de la validation et vérification par le test est de trouver les erreurs qui se trouvent dans un système sous test. Ces erreurs sont des comportements non conformes à la description fonctionnelle du système sous test. Un test est représenté par un ensemble de stimulations données en entrées du logiciel, produisant un résultat. Ce résultat est comparé à un comportement attendu défini par la spécification fonctionnelle, pour déterminer un verdict assurant si le logiciel est conforme ou non aux exigences pour ce test.

Supposons, dans l'exemple précédent, que l'exigence indique une règle de calcul permettant de déterminer quelle approximation doit rendre le logiciel lorsqu'il ne peut pas calculer le résultat exact. On peut donc déterminer si le résultat attendu par la spécification est $y = 2.71$ ou $y = 2.72$. Si cette règle est de renvoyer la valeur la plus grande la plus proche du résultat exact, alors le comportement attendu défini par cette exigence est $y = 2.72$. Ainsi, si lors du test par la stimulation $x = 1.0$, le logiciel sous test calcule le résultat $y = 2.71$, on peut alors affirmer que ce logiciel n'est pas conforme avec le comportement fonctionnel décrit par l'exigence.

Le test permet de montrer la présence d'erreurs spécifiques sous certaines conditions dans le logiciel. En général, le test de logiciels numériques n'est pas exhaustif, l'ensemble des valeurs possibles que peuvent prendre les simulations étant trop grand : dans le cas de notre exemple simpliste, cela représenterait déjà 10^3 valeurs à tester. Il s'agit donc de sélectionner un sous-ensemble représentatif des configurations possibles du logiciel est estimé suffisant pour cette étape de vérification. La génération d'un test par rapport à une spécification fonctionnelle revient donc à répondre aux questions suivantes : quel comportement attendu par la spécification fonctionnelle permet de trouver des erreurs dans le logiciel ? Quelles stimulations permettent d'atteindre ce comportement attendu ?

La génération de tests pour des logiciels numériques complexes est une étape difficile du processus de validation. Il n'existe actuellement pas d'outils ou de méthodes compatibles avec le contexte de certification aéronautique pour la génération de tests numériques. En effet, la génération de tests se fait dans ce contexte à partir d'une spécification fonctionnelle et non du code. Les mêmes problématiques de propagation d'erreur s'appliquent alors à la méthode de génération des tests, afin de fournir des stimulations et un comportement attendu suffisamment précis pour parvenir à un verdict fiable.

Vers l'automatisation de la génération de tests numériques

La première étape de la génération automatique de test est donc de déterminer un formalisme permettant d'exprimer le comportement fonctionnel du logiciel par une syntaxe

formelle. Ce formalisme doit cependant être distinct de la solution proposée, c'est-à-dire le code source, en faisant abstraction des détails d'architecture et d'implémentation. Le choix de ce formalisme est guidé par le type de fonctionnalités exprimées, la spécification de l'interface homme/machine du système de commandes de vol ayant des caractéristiques différentes de la spécification du calculateur de régulation moteurs. Dans cette thèse, nous nous intéressons aux logiciels applicatifs numériques ayant un rôle critique dans le fonctionnement général d'un appareil et étant soumis à des critères de certification stricts.

La génération de tests étant une activité difficile et coûteuse du cycle de développement d'un logiciel [4], de nombreuses études ont été menées pour l'automatisation de ce processus ou l'amélioration de la qualité numérique des tests. Bien que non utilisable en l'état pour la génération de tests pour des logiciels numériques dans le contexte de la certification aéronautique, les principes de ces méthodes peuvent être transposés sur un ensemble d'exigences formalisées.

Effectuer la génération automatique de tests à partir d'un modèle formel du comportement fonctionnel défini par les exigences implique l'utilisation d'arithmétique des ordinateurs en précision finie, afin de déterminer les valeurs associées aux comportements attendus et aux stimulations des différents tests. Tout comme le logiciel embarqué, le générateur de tests est lui aussi soumis à des approximations et erreurs de calcul, pouvant impacter le résultat. Les contraintes d'utilisation du générateur de tests ne sont cependant pas les mêmes que celles du logiciel embarqué (mémoire, consommation d'énergie). Il est donc raisonnable d'étudier l'utilisation d'arithmétiques plus consommatrices de ressources mais fiables pour la génération de tests numériques exploitables pour le processus de certification.

Contributions

Les défis de cette thèse consistent à proposer une solution de génération automatique de tests numériques compatibles avec les contraintes associées à

- la certification aéronautique et le développement de produits industriels,
- la génération automatique de test à partir d'une description fonctionnelle,
- l'arithmétique des ordinateurs pour l'analyse certifiée de codes numériques.

L'étude de ces contraintes nous permet d'identifier les problèmes associés à chaque domaine, et ainsi apporter des solutions compatibles avec nos besoins. Nous nous intéressons ainsi aux problèmes de génération de tests à partir d'un formalisme choisi, mais également à l'arithmétique des ordinateurs afin de proposer un générateur de tests pourvu d'une arithmétique fiable. Nous présentons en premier nos contributions relatives à la

génération automatique de tests, dont les spécificités motivent nos contributions relatives à l'arithmétique des ordinateurs fiable. La solution de génération automatique de tests est cependant étudiée dans le dernier chapitre de la thèse, sous forme de synthèse des différentes problématiques abordées.

Génération automatique de tests pour la certification aéronautique

Ces travaux ont été effectués à Safran Electronics & Defense d'après l'analyse de 500 exigences de Safran Helicopter Engines décrivant le produit logiciel calculant les paramètres généraux d'un moteur d'hélicoptère Arrano. Les exigences décrivant des comportements numériques ne forment pas plus de 10% du volume total d'exigences du produit, mais elles représentent plus de 60% de l'effort de test (nombre total de tests, temps d'écriture des tests, réalisation de ces tests). Proposer une solution de génération automatique de tests numériques fiables compatibles avec la certification aéronautique est donc motivé par une volonté de réduction significative des coûts de développement d'un logiciel soumis à certification.

Formalisation des exigences avec la programmation par contraintes Nous étudions différents formalismes afin d'identifier le plus adapté à la fois pour l'expression d'exigences décrivant un comportement fonctionnel numérique, mais également pour la génération de tests numériques fiable. La programmation par contraintes sur des variables continues présente les qualités recherchées. Dans le chapitre 6, nous adaptons les exigences numériques définies selon une tolérance à un problème de satisfaction de contraintes (CSP) équivalent. Nous utilisons l'outil `RealPaver` [5, 6] permettant d'écrire et résoudre des contraintes et problèmes d'optimisation non linéaires en renvoyant un ensemble de solutions fiables grâce à l'utilisation d'une arithmétique rigoureuse : l'arithmétique d'intervalles [7]. L'arithmétique d'intervalle permet de représenter un nombre et l'erreur qui lui est associée (de représentation et de calculs). C'est une méthode fiable de représentation d'un résultat.

Une solution de tests par mutation Après étude des solutions de génération automatique de tests existant pour le logiciel, nous proposons une solution basée sur le test par mutation [8]. Le test par mutation est une méthode d'évaluation de la qualité d'une campagne de test par la génération de versions modifiées (par des simulations d'erreurs) d'un programme sous test, appelées des mutants. Si un mutant n'est pas détecté par une campagne de test, alors la campagne de test n'est pas robuste à la faute simulée par ce mutant. On peut se servir de ces mutants afin de générer des test pour renforcer une campagne de test, ou bien en générer une entière [9]. Dans cette contribution, nous proposons d'appliquer cette méthode de génération de test sur nos CSP numériques modélisant les exigences. Nous utilisons de nouveau l'arithmétique d'intervalle afin de renvoyer des résultats fiables

et nous développons un prototype de solution en utilisant les propriétés de résolution de contraintes numériques de `RealPaver`.

Ces travaux en collaboration avec Martine Ceberio ont fait l'objet du résumé [10] présenté en 2018 lors de la conférence internationale *18th International Symposium on Scientific Computing, Computer Arithmetics and Verified Numerics, (SCAN18)*.

Arithmétique des ordinateurs fiable

Nos contributions dans ce domaine visent à aborder la problématique de la fiabilité entre les valeurs décrites dans les exigences et le format des tests correspondant. Les exigences sont décrites dans un format décimal, tandis que les stimulations sont données au logiciel dans un format à précision finie binaire. Nous souhaitons créer un environnement dans lequel les différents formats de représentation des valeurs sur un ordinateur puissent cohabiter tout en maîtrisant les erreurs de conversion et de calculs.

Arithmétique d'intervalles décimale multi-précision L'arithmétique d'intervalles est un élément essentiel de notre solution de génération automatique de tests. La précision des intervalles dépend cependant de la précision des formats utilisés pour représenter les bornes. L'arithmétique multi-précision permet d'effectuer des calculs à l'aide de variable à précision finie plus grande que la précision des formats utilisés dans les logiciels embarqués [11]. Afin de fournir des résultats les fidèles possibles au comportement fonctionnel décrit par les exigences, nous souhaitons utiliser des intervalles décimaux dans un format multi-précision. Nous proposons une bibliothèque arithmétique d'intervalle multi-précision décimale, fournissant des opérations de l'arithmétique de base et quelques fonctions élémentaires rapides et fiables, et des résultats précis en décimal.

Ce travail a fait l'objet de l'article [12] publié dans le journal *Reliable Computing* en 2017, ainsi que d'un résumé [13] présenté en 2016 lors de la conférence internationale *17th International Symposium on Scientific Computing, Computer Arithmetics and Verified Numerics, (SCAN17)*.

Arithmétique en bases mixtes Une autre approche consiste à améliorer la précision des opérations entre les formats binaire et décimal en proposant des opérations en bases mixtes, c'est-à-dire prenant en entrées des variables dans les formats binaires et décimaux de base et en renvoyant un résultat binaire ou décimal. Au lieu d'augmenter la précision de calculs, on assure ainsi la précision de chacun des calculs en base mixte dans un format comparable au format de sortie du logiciel, la comparaison entre des formats en base mixte ayant déjà été étudiée [14]. Proposer cette arithmétique permet de réduire les erreurs de

conversion entre les bases, et évite le choix d'une base unique pour effectuer des calculs qui manipulent des nombres dans différents formats. Nous proposons une étude formelle de la faisabilité de la solution, ainsi qu'un algorithme de génération des opérations arithmétiques de base dans ce format mixte. Cet algorithme repose sur l'opération en base mixte Fused-Multiply-Add (FMA), permettant d'effectuer une multiplication et une addition dans une même opération. Cet algorithme est rigoureusement prouvé et la solution implémentée est rapide et minutieusement testée.

Ce travail a fait l'objet d'un article [15] paru en 2018 dans les actes de la conférence internationale *IEEE 25th Symposium on Computer Arithmetic (ARITH)*.

Organisation du document

Cette thèse est composée de trois parties et six chapitres, ainsi que d'une conclusion.

Première partie : Le chapitre 1 présente en détail les spécificités des activités de développement et de validation et vérification par le test dans le contexte de la certification aéronautique. Nous présentons les problématiques et les contraintes liées à la certification, auxquelles doivent se conformer nos travaux.

Le chapitre 2 fournit une présentation détaillée de l'arithmétique des ordinateurs et des différentes arithmétiques permettant d'effectuer des calculs plus sûrs. Nous illustrons sur un exemple les problématiques de génération de test liées à l'utilisation de l'arithmétique des ordinateurs.

Le chapitre 3 propose un état de l'art des différentes méthodes de formalisation des exigences et de génération de tests existante, pour la plupart à partir d'un logiciel. Nous présentons les forces et faiblesses de chacune de ces méthodes, ainsi que la pertinence de leur utilisation dans le contexte présent, afin de motiver notre choix de solution de génération automatique de tests.

Deuxième partie : Dans le chapitre 4, nous présentons la bibliothèque MPDI, bibliothèque d'arithmétique d'intervalles décimale multi-précision. Nous étudions les algorithmes de chaque opération et nous détaillons la solution logicielle développée.

La chapitre 5 aborde le sujet de l'arithmétique en bases mixtes, dont les opérations reposent sur l'algorithme de FMA. Nous proposons un algorithme dont nous effectuons la preuve et nous décrivons les performances de la solution implémentée.

Troisième partie : Le chapitre 6 porte sur la solution de génération automatique de tests. Le choix du CSP comme formalisme pour les exigences est motivé par les contraintes obligeant à distinguer les exigences d'une solution logicielle. Nous définissons les algorithmes

permettant la génération automatique de tests numériques fiables et nous détaillons un prototype de solution.

Une synthèse des contributions que les perspectives qui en découlent sont présentées dans le dernier chapitre.

Liste des publications

Journal scientifique international avec revue par les pairs

2017 S. Graillat, C. Jeangoudoux et C. Lauter. MPDI : A Decimal Multiple-Precision Interval Arithmetic Library. Dans le *Reliable Computing Journal*. [12]

Conférence internationale avec revue par les pairs

2018 C. Jeangoudoux et C. Lauter. A Correctly Rounded Mixed-Radix Fused-Multiply-Add. Pour *IEEE 25th Symposium on Computer Arithmetic (ARITH)*. [15]

Résumés pour conférences internationales

2018 C. Jeangoudoux, M. Ceberio, A.G. Contreras et F. Larribe. Constraints over Intervals for Specification Based Automatic Software Test Generation. Résumé pour le *18th International Symposium on Scientific Computing, Computer Arithmetic, and Verified Numerical Computations (SCAN)*. [10]

2016 S. Graillat, C. Jeangoudoux et C. Lauter. MPDI : A Decimal Multiple-Precision Interval Arithmetic Library. Résumé pour le *17th International Symposium on Scientific Computing, Computer Arithmetic, and Verified Numerical Computations (SCAN)* [13].

PREMIÈRE PARTIE

ÉTAT DE L'ART

CHAPITRE 1

VALIDATION ET VÉRIFICATION DANS LE CONTEXTE DE LA CERTIFICATION AÉRONAUTIQUE

Du pilote automatique au système de capteurs, en passant par le contrôle des commandes de bord ou encore le système de divertissement multimédia, les avions actuels sont équipés de systèmes informatiques nombreux et variés. L'utilisation de ces systèmes a permis de remplacer des appareils mécaniques, en remplissant les mêmes fonctions tout en réduisant la charge de matériel à bord, ou bien d'ajouter de nouvelles fonctionnalités pour faciliter le travail de pilotage ou le service à bord.

La première utilisation de ces systèmes informatiques embarqués remonte aux années 1960, avec l'apparition d'ordinateurs de navigation embarqués dans les missions spatiales [16]. Ils se sont démocratisés avec l'aviation commerciale dans les années 1970. Alors que le transport de passagers civils se généralise, les besoins plus importants de sûreté et fiabilité des systèmes embarqués engendrent des exigences plus strictes. L'emploi de systèmes informatiques embarqués dans des avions civils est standardisé pour une première fois dans les années 1980 [1] en Europe par l'European Organisation for Civil Aviation Electronics, maintenant appelée l'Organisation européenne pour l'équipement de l'aviation civile (EUROCAE), et par la Radio Technical Commission for Aeronautics, maintenant appelée RTCA, Inc., aux États-Unis et au Canada. Les logiciels embarqués dans les aéronefs civils (avions, hélicoptères, etc.) sont encore aujourd'hui soumis à des réglementations strictes. La qualité de ces logiciels est évaluée selon de nombreux critères avant que ceux-ci ne soient approuvés pour l'intégration à l'avion : c'est le processus de certification [1] [17].

Ce chapitre a pour objectif d'introduire le contexte spécifique dans lequel s'inscrit cette thèse : la validation et vérifications de logiciels numériques soumis au processus de certification aéronautique. Dans ce chapitre, nous présenterons dans un premier temps le document DO–178C [1] fournissant des recommandations pour la conception et certifi-

1. EASA – certification aéronautique

cation de logiciels embarqués pour l'aéronautique. Nous nous intéresserons ensuite plus spécifiquement aux étapes de validation et vérification du logiciel, et nous présenterons ces spécificités sur un exemple. Nous souhaitons insister sur le fait que ce chapitre a pour but d'introduire au lecteur le contexte et aux problématiques relatives à la certification aéronautique, ainsi le langage employé reflète cette spécificité.

1.1 Certification aéronautique

1.1.1 Contexte légal

Le développement de systèmes informatiques complexes et la mise en conformité avec un vaste ensemble de réglementations légales sont des activités difficiles, onéreuses et consommatrices de ressources. L'industrie aéronautique est contrainte de fournir un service de qualité, en termes de respect d'exigences de sécurité, de robustesse, de sûreté et de fonctionnement [4]. Le développement, l'intégration et l'utilisation d'un produit informatique à un véhicule aérien transportant des passagers civils sont donc soumis à un processus de vérification visant à donner un certain niveau de confiance dans la conformité avec les réglementations et les législations en vigueur : c'est la certification.

Définition 1 (Certification [1])**.** *Reconnaissance légale par une autorité qu'un produit, un servie, une organisation ou une personne est conforme à un ensemble d'exigences. Une telle certification comprend l'activité de vérification technique du produit, du service, de l'organisation ou de la personne et la reconnaissance formelle de sa conformité avec les exigences applicables par émission d'un certificat, d'une licence, d'une autorisation ou tout autre document exigé par la législation nationale.*

En particulier, la certification d'un produit implique :

- *le processus d'évaluation de la conception d'un produit pour s'assurer qu'il est conforme à un ensemble d'exigences applicables à ce type de produit afin de démontrer un niveau de sécurité acceptable,*
- *le processus d'évaluation d'un produit individuel pour s'assurer qu'il est conforme avec la définition du type de conception certifiée,*
- *la délivrance d'un certificat tel qu'exigé par la législation nationale pour déclarer le respect ou la conformité avec les exigences, correspondant aux points ci-dessus.*

Une autorité de certification est un organisme ou une personne responsable dans l'État, le pays ou tout autre organisme compétent, chargé de la certification ou de l'approbation d'un produit conformément à un ensemble d'exigences. Les autorités de certifications les plus importantes sont la Federal Aviation Administration (FAA) et Transport Canada pour

les États-Unis et le Canada, et l'Agence européenne de la sécurité aérienne (EASA) pour l'Union Européenne. Un système informatique ayant passé la certification d'une de ces agences peut être intégré dans les aéronefs transportant des passagers civils [17, 18].

L'ensemble des exigences de sûreté et de sécurité imposées par les autorités de certification pour qu'un logiciel puisse légalement être utilisé dans un système aéronautique est cependant abstrait. Les exigences de sûreté permettent d'assurer la robustesse du système aux erreurs internes tandis que les exigences de sécurité assurent la fiabilité des fonctionnalités du système face aux facteurs externes [19]. Les technologies évoluant et les méthodes de développement, validation et vérification également, passer une certification est un exercice difficile et onéreux [4]. C'est pourquoi les industriels et professionnels de l'aviation sont arrivés à un consensus pour l'élaboration d'un certain nombre de documents, reconnus par l'EASA et la FAA [17, 18], fournissant des recommandations techniques qui portent sur différents aspects du développement et du cycle de vie d'un logiciel candidat à la certification. Il existe plusieurs catégories de documents, portant sur différents aspects de la certification des systèmes aéronautiques. Cette thèse se pose dans le contexte du développement de logiciels embarqués. Plusieurs documents de recommandations sont pertinent dans ce contexte, leurs relations et domaines d'applications sont détaillés dans la figure 1.1.

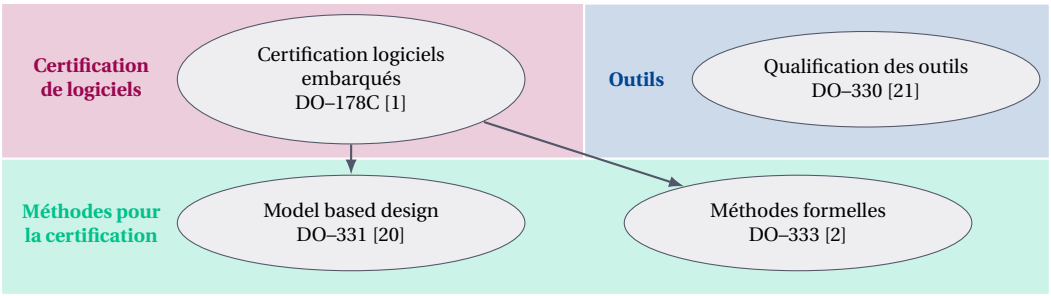


FIGURE 1.1 – Relations entre les documents de recommandations pour la certification

Le document principal de recommandations acceptées par l'industrie pour satisfaire aux exigences de navigabilité est la *DO–178C, Software Considerations in Airborne Systems and Equipment Certification* [1]. Les recommandations fournies par ce document concernent les logiciels utilisés dans des aéronefs transportant des passagers civils. Elles portent notamment sur le cycle de vie du logiciel, de sa conception fonctionnelle à son exécution sur l'ordinateur cible [2], en passant par sa validation et sa vérification. Il fournit des indications précises pour déterminer de manière cohérente et avec un niveau acceptable de confiance quels sont les aspects logiciels des systèmes et équipements embarqués

2. L'ordinateur embarqué dans le système aéronautique sur lequel le logiciel est exécuté.

conformes aux exigences de navigabilité, c'est-à-dire des fonctionnalités du système de navigation. Les détails des recommandations préconisées par ce document pour répondre aux critères de certification sont abordés dans la suite de ce chapitre.

Les recommandations définissent des objectifs à atteindre pour valider des critères de certification Elles sont indépendantes des méthodes de conception, développement ou vérification utilisées pour atteindre ces objectifs. La DO–331 [20] fournit des recommandations pour les projets qui choisissent d'utiliser le Model-Based Design (MDB) [22] pour le développement et la vérification du produit logiciel. Le MDB est une méthode de développement et vérification d'un logiciel utilisant une représentation abstraite d'un ensemble d'aspects du logiciel. Dans le cadre de la DO–331, cette représentation est une modélisation graphique ou textuelle, représentant les exigences de design (voir figure 1.3), c'est-à-dire les exigences sur les détails de développement et d'architecture du logiciel.

La DO–333 [2] fournit des recommandations sur l'utilisation de méthodes formelles pour la validation et vérification du logiciel soumis à certification. Les méthodes formelles sont des notations descriptives et méthodes analytiques utilisées pour construire, développer et raisonner sur les modèles mathématiques du comportement du système. Une méthode formelle est une analyse réalisée sur un modèle. Ces méthodes comprennent la démonstration automatique de théorèmes et les assistants de preuve [23], le Model Checking [24], ainsi que l'interprétation abstraite [25].

Enfin la dernière catégorie concerne les outils utilisés pour faciliter ou automatiser les différentes étapes de développement et conception d'un logiciel embarqué soumis à certification. La DO-330 [21] notamment décrit le cycle de qualification des logiciels utilisés pour automatiser certaines étapes du cycle de développement décrit dans la DO–178C. La qualification est un processus de certification des outils utilisés lors de la conception d'un produit soumis à certification. Ce processus est abordé plus en détail dans la section 1.2.4.

Il existe d'autres documents de recommandations qui ne sont pas représentés sur cette figure, comme la DO–278A équivalente à la DO–178C relative aux systèmes informatiques au sol, ou encore la DO–332, document portant sur les méthodes relatives aux technologies orientées objet. Les recommandations fournies par ces documents ne sont pas applicables pour les systèmes embarqués soumis à certification et ne seront pas abordés dans le contexte de cette thèse.

1.1.2 Niveaux de certification

La certification concerne tous les logiciels embarqués dans un système aéronautique. Tous ces logiciels n'ont cependant pas les mêmes fonctions, et un dysfonctionnement dans le logiciel de gestion du contenu de divertissement multimédia n'a pas le même impact sur la qualité du vol qu'une erreur se produisant dans le système de contrôles de

commandes en vol. La DO–178C [1] définit différents niveaux de criticité associés aux logiciels embarqués. La criticité d'un logiciel est définie selon l'impact d'une erreur sur son fonctionnement et selon la propagation de la faute au niveau du système aéronautique.

Une *faute* est une manifestation d'une erreur dans un logiciel. Une faute, si elle se produit, peut provoquer une défaillance. L'impact d'une faute logicielle est illustrée dans la figure 1.2. Une *défaillance* est l'incapacité d'un système ou d'un composant système à exécuter une fonction requise dans les limites spécifiées. Une défaillance peut être produite lorsqu'un défaut est rencontré. Une défaillance ou plusieurs défaillances peuvent produire une *condition de défaillance*, c'est-à-dire une incidence directe ou indirecte sur l'avion et ses occupants, en tenant compte des conditions opérationnelles et environnementales défavorables. Une condition de défaillance est classée en fonction de la gravité de ses effets ; elle peut être, de la plus à la moins importante, catastrophique, dangereuse, majeure, mineure, et sans effets sur la sécurité.

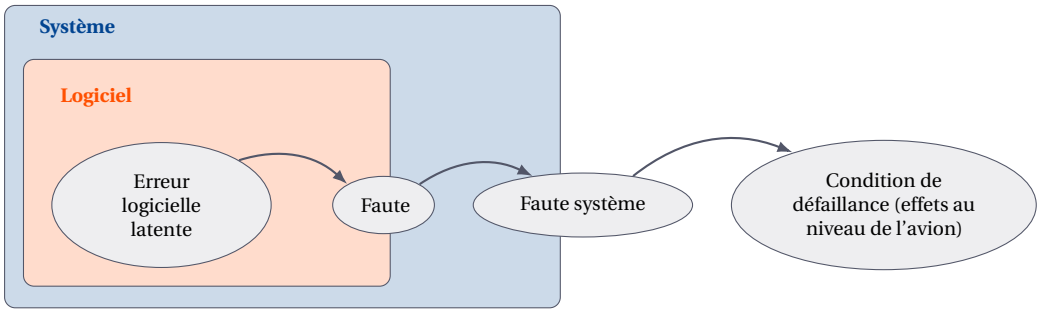


FIGURE 1.2 – Séquence d'évènement menant à une condition de défaillance (figure adaptée de la DO–178C) [1]

La certification de logiciels aéronautiques s'effectue donc selon un niveau de criticité, défini dans la DO–178C [1], appelé *Development Assurance Level*, ou DAL. Ces niveaux sont notés de A à E, et sont définis comme suit :

- **DAL–A :** un dysfonctionnement du logiciel peut entraîner un problème *catastrophique* compromettant directement les conditions de vol et/ou d'atterrissage du système aéroporté, (exemple : dysfonctionnement des instruments de vol ou des moteurs).
- **DAL–B :** un dysfonctionnement du logiciel peut entraîner un problème *dangereux* ayant un impact sur des équipements vitaux de l'appareil, (exemple : détresse physique des passagers, manœuvrabilité de l'appareil réduite).
- **DAL–C :** un dysfonctionnement du logiciel peut entraîner un problème *majeur* ayant un impact sur des équipements vitaux de l'appareil, (exemple : inconfort des

passagers et détérioration des conditions de travail de l'équipage).

- **DAL–D :** un dysfonctionnement du logiciel peut entraîner un problème *mineur* sur l'appareil, (exemple : inconfort des passagers).
- **DAL–E :** un dysfonctionnement du logiciel n'entraîne aucune répercussion sur la sécurité, la manœuvrabilité de l'appareil et les conditions de travail de l'équipage.

Un logiciel développé avec pour but la certification en DAL–A aura donc des critères et objectifs plus stricts qu'un logiciel développé en DAL–E. Dans cette thèse, nous souhaitons améliorer le processus de validation et vérification des logiciels aux contraintes numériques fortes. Ainsi, nous nous plaçons dans le contexte de certification DAL–A.

1.1.3 Cycle de vie d'un logiciel soumis à certification

Le cycle de vie du logiciel représente l'ensemble des processus déterminés par une organisation, jugés comme appropriés et suffisants pour produire un logiciel. Cela représente également la période de temps qui commence avec la décision de produire ou de modifier un produit logiciel et se termine lorsque le produit est mis hors service. Dans le contexte de la certification d'un logiciel destiné à être intégré dans un système aéronautique, ce n'est pas simplement la qualité du logiciel qui est évaluée, mais également toutes les étapes du processus qui ont abouti à l'exécution du logiciel dans le système aéronautique. Ainsi les autorités de certification demandent de fournir un certain nombre de documents justifiant les activités de conception, développement, validation et vérification.

Le développement d'un logiciel est entrepris pour répondre à un ou plusieurs besoins du système. Dans le contexte de la certification aéronautique, on appelle *système* un ensemble de composants matériels et logiciels organisés pour remplir une fonction spécifique ou un ensemble de fonctions. L'architecture système est la structure du matériel et du logiciel sélectionné pour répondre aux exigences du système. Ce besoin est formalisé dans un ensemble d'exigences du système. Une des préconisations de la DO–178C [1] est de raffiner ces exigences du système sur plusieurs niveaux afin de décrire au mieux les différents aspects du logiciel : ses fonctions, son architecture, ses contraintes de précision, ses algorithmes, etc.

Définition 2 (Exigences du logiciel [1])**.** *Description de la production attendue par le logiciel en fonction des entrées et de contraintes. Les exigences du logiciel incluent à la fois les exigences de haut niveau (High-Level Requirement, HLR) et les exigences de bas niveau (Low-Level Requirement, LLR).*

Les exigences de haut niveau (HLR) sont les exigences du logiciel développées à partir de l'analyse des exigences du système, des exigences liées à la sécurité et à l'architecture du système. Elles expriment les fonctionnalités du logiciel.

Les exigences de bas niveau (LLR) sont les exigences du logiciel développées à partir des HLR, des exigences dérivées et des contraintes de conception à partir desquelles le code source peut être directement implémenté sans informations supplémentaires.

Les exigences du logiciel sont donc définies à partir d'une spécification des besoins du système. Pour ce faire, les exigences (HLR et LLR) sont conçues en suivant des règles d'écriture, permettant d'assurer la conformité avec les contraintes de certification définies par DO–178C. Ces règles sont de différentes natures et ont un impact direct sur le développement du logiciel, mais également sa validation. Une exigence doit respecter les critères suivants :

- *Conformité avec les exigences du système :* les opérations définies dans les exigences du système sont détaillées et énoncées correctement dans les exigences du logiciel (HLR puis LLR), aux niveaux du comportement fonctionnel, des performances et de la robustesse/sécurité, c'est-à-dire la mesure dans laquelle le logiciel peut continuer à fonctionner correctement malgré des entrées et des conditions anormales.
- *Précision :* toute exigence doit être précise, non ambiguë et suffisamment détaillée.
- *Cohérence :* les exigences ne doivent pas entrer en conflit les unes avec les autres.
- *Vérifiabilité :* chaque exigence doit être atteignable par au moins un test.
- *Traçabilité :* c'est l'association entre plusieurs éléments, c'est-à-dire du lien entre les sorties et leur origine, ou entre une exigence et son implémentation. Par exemple un lien de traçabilité doit être établi entre chaque exigence du logiciel et au moins une exigence du système.

La traçabilité entre les exigences du système et le code exécutable embarqué est un point clef du développement d'un produit logiciel soumis à la certification. Ces liens de traçabilité permettent de justifier les choix de conception et d'implémentation, mais également de vérifier par les activités d'analyse et de relecture qu'aucun élément n'a été oublié ou ajouté dans le logiciel [26]. Il se passe généralement 20 à 30 ans entre deux générations d'avions [4], ces propriétés de traçabilité sont essentielles pour la maintenance et la gestion des évolutions du logiciel sur toute la durée de la vie du système.

L'ensemble des recommandations fournies dans la DO–178C sur les activités nécessaires dans le cycle de vie du logiciel pour faciliter le processus de certification sont résumées dans la figure 1.3, pour un logiciel soumis à des critères de certification de niveau DAL–A. Les activités de conception et développement du logiciel sont divisées en plusieurs parties : dans un premier temps, les exigences du système sont traduites en *exigences de haut niveau*, puis elles sont raffinées pour décrire le *design*, formé des *exigences de bas niveau* et de l'*architecture* du logiciel. Le design sert à définir Le *code source*, qui est compilé sur la cible pour former le *code exécutable objet* [1].

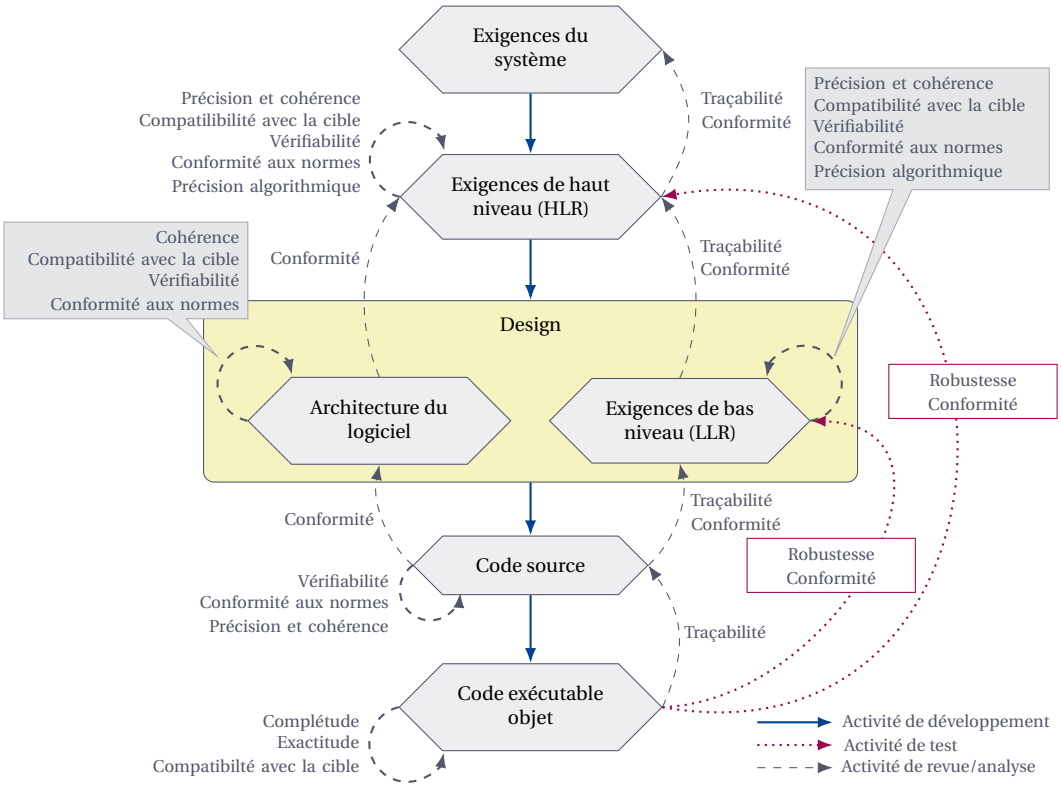


FIGURE 1.3 – Certification d'un logiciel niveau DAL–A : critères et objectifs définis par la DO–178C (figure adaptée de la DO–333) [2]

Le premier niveau de description du logiciel à partir des exigences du système est décrit par les exigences de haut niveau, ou High Level Requirements (HLR). Un lien de traçabilité est établi entre les HLR et les exigences système. Les HLR décrivent le « quoi », c'est-à-dire le comportement fonctionnel du logiciel, mais également les valeurs limites du logiciel, introduisant le concept de robustesse, c'est-à-dire la capacité d'un système à se comporter de façon maîtrisée lorsqu'il s'exécute en dehors de son contexte nominal. Les propriétés de compatibilités avec la cible, c'est-à-dire le matériel, l'architecture et la configuration sur lesquels va être implémenté le logiciel, sont également décrites à ce niveau.

Un niveau inférieur de description du logiciel est le design, qui comprend l'architecture du logiciel et les exigences de bas niveau, ou Low Level Requierements (LLR). Ils décrivent le « comment », c'est-à-dire les algorithmes utilisés et choix de conception pour répondre aux fonctionnalités requises par les HLR. Les choix de développement et d'optimisation sont alors justifiés et tracés vers les HLR.

Le code source est généré à partir des LLR et de l'architecture du logiciel, puis le code

exécutable est généré sur l'ordinateur cible. Toutes ces étapes sont soumises à vérification et validation par relecture/review selon des activités de validation plus ou moins strictes en fonction du niveau DAL visé pour la certification du logiciel. Les flèches entre les différents niveaux du cycle de développement sur la figure 1.3 représentent les activités de validation exigées pour un niveau de certification en DAL–A. Pour ce niveau de certification, le code objet doit être relu et un lien de traçabilité doit être établi depuis des exigences du système jusqu'au code objet [1].

Remarque 1. *Les exigences de haut niveau ne sont pas du code, ni le design. Les choix relevant de la conception de l'architecture du logiciel et aux détails algorithmiques tels que la précision des variables utilisées sont laissés au développeur, ou à une méthode d'automatisation.*

1.1.4 Exemple d'exigences

Nous illustrons cette remarque par un exemple classique d'exigences d'un produit logiciel : le calcul des paramètres généraux du moteur pendant toutes les phrases de sa mise en fonction (au démarrage, au sol, en vol, à l'atterrissage). Un des composants du logiciel peut être amené à calculer le débit d'air prélevé dans le moteur. Nous faisons la supposition dans cette thèse que les exigences qui nous ont été données pour travailler sont représentatives d'un certain type de fonctionnalités développées dans le contexte de logiciels certifiés, et sont écrites selon les règles et critères détaillés dans la section 1.1.3. Un extrait des exigences d'un tel logiciel seraient alors les suivantes.

E01 : Définition des intervalles normaux Les valeurs d'exécution normale des variables d'entrée et des constantes affectées aux exigences suivantes sont décrites dans le tableau 1.1.

Entrée(s)	Sortie(s)	Constante(s)	Table(s)
$x_1 \in [x_1, x_1]$	y	$c_1 = 5$	T_1
$x_2 \in [x_2, x_2]$		$c_2 = 100$	
$x_3 \in [x_3, x_3]$		$\overline{C}_3 = 300$	
$x_4 \in [x_4, x_4]$		$\underline{C}_3 = -300$	
$m_r \in [m_r, m_r]$		$c_4 = 0.01$	

Tableau 1.1 – Variables et constantes et leurs intervalles normaux intervenant dans la détermination du débit d'air prélevé y

E02 : Exigence de robustesse Lorsque les valeurs d'entrées et de constantes ne sont pas dans leur intervalle de définition, elles sont alors saturées à leurs bornes. Une valeur

supérieure à la borne supérieure de la variable est saturée à sa borne supérieure et une valeur inférieure à la borne inférieure d'une variable est saturée à sa borne inférieure. Un drapeau d'exception `incorrect range exception` est alors levé.

E03 : Exigence de performances Le débit d'air prélevé doit être calculé en moins de c_1 secondes. Dans le cas contraire, la valeur $\underline{C}_3$ est affectée au débit d'air prélevé y et un drapeau `Time out` est levé.

E04 : Exigence de précision Le débit d'air prélevé doit être calculé avec une précision relative de 10^{-7} par rapport au résultat exact calculé en précision infinie.

E05F01 : Exigence fonctionnelle Le débit d'air prélevé doit valoir le produit des variables suivantes :

- la constante c_2,
- le ratio des variables suivantes :
  - — la différence de la variable d'entrée x_1 par le débit d'air prélevé réduit d_r,
  - — la variable t_1, produite à partir de la table T_1 en fonction de l'entrée x_2.

Le résultat de ce produit devra être préalablement saturé entre $\underline{C}_3$ en valeur minimale et $\overline{C}_3$ en valeur maximale.

E05F02 : Exigence fonctionnelle Si l'entrée x_3 est passée dans le mode d'air prélevé réduit, c'est-à-dire m_r vaut `true`, alors le débit d'air prélevé réduit d_r doit valoir le ratio des variables suivantes :

- taux de détente d'air prélevé, l'entrée x_3
- la pression de prélèvement consolidée x_4, préalablement saturée à c_4 en valeur minimale (afin d'éviter une potentielle division par zéro).

1.2 Validation et vérification

La validation est le processus consistant à déterminer si les exigences sont correctes et complètes. Les méthodes utilisées dans le cycle de vie du système pour la validation du système peuvent utiliser les exigences du logiciel et les exigences dérivées du système. La vérification est l'évaluation des résultats d'un processus ou d'une méthode pour assurer l'exactitude et la cohérence des entrées et paramètres fournis à ce processus.

La validation et la vérification d'un logiciel ont pour but l'évaluation de la qualité du logiciel selon plusieurs critères afin d'apporter un certain niveau de confiance. Ces activités font partie du processus d'évaluation de la sécurité du système, c'est-à-dire de l'évaluation

continue, systématique et complète du système proposé pour montrer que les exigences pertinentes en matière de sécurité sont satisfaites. Les principales activités de ce processus comprennent : l'évaluation du risque fonctionnel, l'évaluation préliminaire de la sécurité du système et l'évaluation de la sécurité du système. Ces activités dépendront de la criticité, de la complexité, de la nouveauté et de l'expérience de service pertinente du système concerné.

Il existe plusieurs méthodes pour procéder à la validation et de la vérification d'un système, telles que les méthodes formelles décrites dans la DO–333 [2]. Une des méthodes de validation et vérification la plus répandue est le test.

1.2.1 Validation et vérification par le test

Le test est prescrit par la DO–178C comme une activité de validation et vérification permettant d'apporter une certaine confiance dans le développement et l'implémentation du code exécutable objet au regard des exigences, HLR et LLR. Quand bien même les méthodes formelles de vérification comme la preuve de théorèmes assistée [23] ou le Model Checking [24] sont de plus en plus utilisées pour la certification de logiciels aérnautiques, le test continue à être une activité obligatoire, notamment parce qu'il permet de mettre en avant les erreurs se produisant dans le logiciel embarqué sur la cible, comme par exemple des erreurs liées à la compilation, tandis que les méthodes formelles se concentrent sur la démonstration de propriétés à partir du design du logiciel. Les méthodes formelles et le tests ne sont donc pas rivales, mais plutôt des activités complémentaires pour assurer la validation et vérification du logiciel dans un environnement ciblé. La génération de test est cependant une activité coûteuse en temps et en ressources, et dans le cadre de la certification, soumise à des contraintes strictes.

Définition 3 (Test [1])**.** *Le test est un processus de mise en œuvre d'un système ou d'un composant du système pour vérifier s'il répond aux exigences spécifiées et détecter les erreurs.*

Un cas de test *est un ensemble d'entrées de test, de conditions d'exécution et de résultats attendus développés pour un objectif particulier, tel que l'exercice d'un chemin de programme particulier ou la vérification de la conformité à une exigence spécifique.*

Une procédure de test *est un ensemble d'instructions détaillées pour la configuration et l'exécution d'un ensemble donné de cas de test, et d'instructions pour l'évaluation des résultats de l'exécution des cas de test.*

La figure 1.4 illustre les différents concepts introduits dans cette définition. Un cas de test, produit à partir d'un objectif de test, est formé d'une procédure de test et d'un comportement attendu associé à cette procédure de test. Une procédure de test est un ensemble de stimulations données en entrées au logiciel. Un comportement attendu

est déterminé par un oracle de test, c'est-à-dire un mécanisme permettant d'évaluer si un test a été efficace pour trouver des erreurs (c'est-à-dire `KO`) ou non, (soit `OK`). Une procédure de test peut être compatible avec plusieurs cas de test, une même séquence de stimulations pouvant avoir un impact à plusieurs endroits du logiciel, impliquant plusieurs comportements attendus [3]. Un test est donc l'application des stimulations d'une procédure de test sur un système sous test pour observer un comportement produit. La comparaison entre le comportement attendu et le comportement produit permet de former un verdict. Il est donc inattendu, mais logique, qu'un test réussi soit un test permettant efficacement de dévoiler une erreur dans le code.

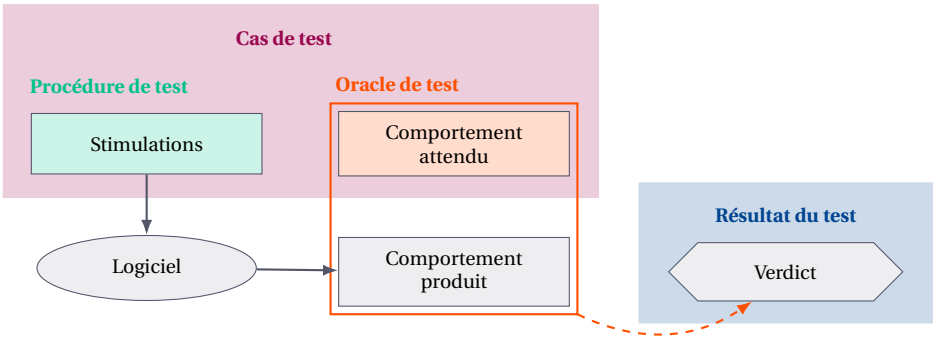


FIGURE 1.4 – Activités de test logiciel

Il existe plusieurs types de test [27, 1], correspondant aux différents objectifs qui leurs sont attribués. Le *test fonctionnel* vise à identifier des fautes aux bornes du logiciel dans les fonctionnalités individuelles du système. Les erreurs logicielles peuvent ainsi être découvertes grâce à l'identification de ces fautes. Il se base sur la description fonctionnelle du logiciel, c'est-à-dire ses HLR. Le test permet également de vérifier le fonctionnement du logiciel à plusieurs niveaux, avec le *test unitaire* produisant des résultats sur chaque composant du système, à l'inverse du *test d'intégration* qui cherche à relever les disparités dans la combinaison et l'intégration de ces différents composants. Le test représente généralement un sous ensemble des valeurs possibles que l'on peut donner en stimulations au logiciel. Ces stimulations peuvent prendre aussi bien des valeurs correspondant au comportement fonctionnel normal que des valeurs correspondant à une utilisation anormale du logiciel, ce sont les *tests de robustesse*. Enfin, le logiciel embarqué s'exécute dans un environnement aux ressources limitées, il doit donc passer des *tests de performance*.

3. Par exemple un comportement attendu par état du système observé.

1.2.2 Test basé sur les exigences

La DO–178C atteste que les objectifs du processus de vérification du logiciel sont satisfaits par une combinaison de méthodes, comprenant l'examen, l'analyse, le développement de cas de test et de l'exécution ultérieure des procédures de test. Les revues et analyses fournissent une évaluation de l'exactitude, de l'exhaustivité et de la vérifiabilité des exigences du logiciel, de son architecture et du code source. Le développement de cas et de procédures de test peut fournir une évaluation plus poussée à propos de la cohérence interne et de la complétude des exigences. L'exécution des procédures de test fournit une démonstration de la conformité et de la robustesse du code exécutable objet avec par rapport aux HLR, aux LLR, et la compatibilité avec la configuration cible.

La DO–178C insiste sur la notion de test basé sur les exigences, sa différence avec les méthodes formelles étant qu'une campagne de test s'applique directement sur le code exécutable objet compilé dans l'environnement de la cible. Les erreurs trouvées de cette façon sont donc les plus représentatives des erreurs pouvant arriver dans une situation d'utilisation réelle.

Nous représentons sur la figure 1.5 le concept de la vérification et validation selon la DO–178C. Ainsi dans le contexte de la certification, pour assurer la conformité avec les recommandations fournies par ce document, il faut effectuer en parallèle les activités de développement du logiciel et de développement des tests à partir des mêmes exigences fonctionnelles de façon indépendante. Les objectifs attachés au développement du logiciel sont axés sur les performances et l'optimisation, tandis que les objectifs attachés au développement des tests est de détecter et de signaler les erreurs éventuellement introduites lors des processus de développement de logiciel.

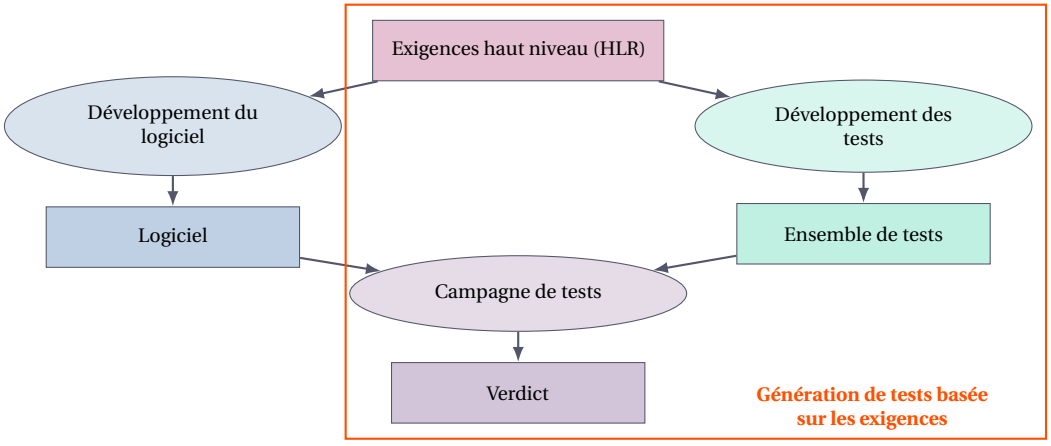


FIGURE 1.5 – Génération de tests basés sur la spécification fonctionnelle

L'activité de développement du logiciel consiste à écrire un code source à partir d'une description du design de ce logiciel, composé des LLR et de l'architecture de celui-ci, comme illustré dans la figure 1.3. Les langages comme le langage textuel Esterel [28] ou le langage graphique SCADE [29] permettent d'effectuer une description structurée et concise d'automates à états finis de très grande complexité, c'est-à-dire des langages permettant une description à un plus haut niveau d'abstraction du logiciel que le langage C ou Ada utilisés jusque-là en aéronautique. Le code source, et ainsi le code exécutable, peut être généré directement depuis ces descriptions, de manière formellement prouvée. Ces langages ont permis de maîtriser, en termes de lisibilité, de temps de développement les logiques d'activation qui se déroulent dans le temps, ramenant l'activité de développement à un niveau équivalent à celui du design, automatisant également la validation et vérification des LLR [30].

L'activité de développement des tests est donc pertinente au niveau des HLR, pour vérifier et valider les fonctionnalités du logiciel. Les HLR sont généralement écrites en langage naturel, par obligation légale. Fournir un langage formel pour décrire les HLR est possible uniquement si ce langage décrit le comportement fonctionnel, et si on peut les distinguer du design. La première étape de l'activité de développement des tests consiste à lire et comprendre les exigences. Le testeur crée ainsi un modèle de sa compréhension des HLR. Il se demande ensuite quelle partie des exigences pourrait être mal implémentée ou contenir des erreurs. En fonction des cas de tests qui émergent de cette étape, il faut ensuite trouver les points discriminants du système, renvoyant une séquence de stimulations permettant d'atteindre ces cas de tests. On a ainsi une séquence de stimulation et un comportement attendu : ce sont les procédures de test.

L'exécution de la campagne de test et la détermination du verdict est effectuée de façon automatique chez Safran, avec l'utilisation de l'outil Software Component Test Lab (SCTL), permettant d'entrer les séquences de stimulations, puis de les appliquer sur le logiciel. Le verdict de la campagne de tests est renvoyé automatiquement.

Générer automatiquement des tests logiciels s'applique donc au niveau des HLR. Les trois étapes de l'écriture des tests sont actuellement peu outillées, et aucune traçabilité n'est effectuée entre la modélisation des exigences par le testeur, qui est faite sur l'équivalent d'un brouillon, et les cas et les procédures de test. La traduction automatique du langage naturel vers un langage formel est un vaste sujet qui a déjà été abordé dans des travaux précédents [31], et n'est pas un point qui sera abordé dans cette thèse.

Remarque 2. *Nous ferons donc le choix d'un langage de modélisation du comportement fonctionnel décrit dans les HLR, à partir duquel nous pourrons effectuer la génération automatique des tests.*

1.2.3 Vérification de la qualité des tests

La vérification d'un système par le test, en plus d'avoir un coût de développement élevé, permet seulement de montrer la présence de défaillances dans le système embarqué sous certaines conditions, et ainsi de les corriger. Tant qu'un test n'est pas totalement exhaustif, ce qui n'est pas réalisable lorsqu'on dispose de ressources et de temps limités, on ne pourra jamais conclure à l'absence d'erreur dans le système. On peut cependant juger de la qualité de la vérification, qui est directement liée à la qualité de la conception des tests. La qualité d'une campagne de test peut être évaluée selon plusieurs critères de couverture, s'appliquant aussi bien sur le code que sur les exigences.

L'analyse de la couverture [32] est un processus permettant de déterminer dans quelle mesure une activité de vérification de logiciel proposée satisfait son objectif. L'analyse de couverture de test est un processus en deux étapes comprenant une analyse de couverture basée sur les exigences et une analyse de couverture structurelle. La première étape analyse les scénarios de test par rapport aux exigences du logiciel pour confirmer que les scénarios de test sélectionnés répondent aux critères spécifiés. La deuxième étape confirme que les procédures de test basées sur les exigences ont permis d'atteindre des critères de couverture définis sur la structure du code. L'analyse de la couverture structurelle permet de déterminer quelle structure de code, y compris les interfaces entre composants, n'a pas été exercée par les procédures de test basées sur les exigences. On peut ainsi identifier des problèmes inhérents au code, tel que la présence de code mort.

La couverture basée sur les exigences, ou couverture fonctionnelle est une prérogative déclarée nécessaire par la DO–178C. Cependant, aucun critère de couverture concret n'est associé à ce concept de couverture des HLR et LLR par le test, mis à part le fait qu'au moins un test est associé par traçabilité à une exigence. Les tests auxquels nous nous intéressons dans cette thèse sont les tests générés à partir des HLR.

Remarque 3. *Pour effectuer la génération automatique des tests les plus pertinents possibles, il nous faudra donc décrire formellement quels sont les critères de couverture fonctionnelle permettant d'attester de la qualité d'une campagne de test.*

Dans le cadre de la couverture structurelle, une *condition* est une expression booléenne atomique, c'est-à-dire qui ne contient aucun opérateur booléen excepté l'opérateur unaire de négation (`NOT`). Une *décision* est une expression booléenne composée de conditions et d'opérateurs booléens. Si une condition apparaît plusieurs fois dans une décision, chaque occurrence est une condition distincte.

Les critères de couverture structurelle peuvent être les suivant [27, 1] :

- *Couverture des instructions :* chaque instruction du programme a été appelée au moins une fois.

- *Couverture des conditions (CC) :* chaque point d'entrée et de sortie du programme a été invoqué au moins une fois et chaque condition du programme a pris au moins une fois tous les résultats possibles.
- *Couverture des décisions (DC) :* chaque point d'entrée et de sortie du programme a été invoqué au moins une fois et chaque décision du programme a pris au moins une fois tous les résultats possibles.
- *Modified condition/decision coverage (MCDC) [33] :* chaque point d'entrée et de sortie du programme a été invoqué au moins une fois, chaque condition dans une décision du programme a pris au moins une fois tous les résultats possibles, chaque décision du programme a eu tous les résultats possibles au moins une fois et il a été montré que chaque condition dans une décision affecte le résultat de cette décision indépendemment des autres conditions. Ce dernier point est démontré en modifiant uniquement cette condition en maintenant toutes les autres conditions de la décision figées.

Exemple illustré des critères de couverture structurelle Prenons l'exemple suivant pour illustrer les critères de couverture structurelle :

```
if ( (A || B) && C )  {
  /* instructions */
}  else {
  /* instructions */
}
```

Dans cet exemple, les expressions atomiques booléennes `A`, `B` et `C` sont des conditions, et l'expression `((A || B) && C)` est une décision.

Ainsi les valeurs que doivent prendre les expressions `A`, `B`, `C` pour permettre d'effectuer une campagne de test minimum permettant de valider la couverture structurelle selon les critères suivants sont :

- Conditions (CC) : `A = true`, `B = true`, `C = true`,
  `A = false`, `B = false`, `C = false`.
- Décisions (DC) : `A = true`, `B = true`, `C =true`, décision : `true`,
  `A = false`, `B = false`, `C = false`, décision : `false`.
- MCDC : `A = false`, `B = false`, `C = true`, décision : `false`,
  `A = false`, `B = true`, `C = true`, décision : `true`,
  `A = false`, `B = true`, `C = false`, décision : `false`,
  `A = true`, `B = false`, `C = true`, décision : `true`.

Un autre type de couverture du programme est la couverture des données d'entrées. Une couverture exhaustive des valeurs des données d'entrées est inconcevable dès lors que

les variables considérées sont des variables entières, ou que le nombre de ces variables est important. On considère alors les classes d'équivalence des entrées, c'est-à-dire la partition du domaine des entrées d'un programme telle qu'un test d'une valeur représentative de la classe équivaut à un test des autres valeurs de la classe. On peut ainsi réduire le nombre de test nécessaire pour effectuer la couverture des domaines des entrées, tout en ayant un critère sur le nombre de tests suffisant. Cependant la combinaison des entrées dans le logiciel pouvant être complexe, la détermination de ces classes d'équivalence n'est pas forcément directe et peut s'avérer être une activité délicate et difficile.

Remarque 4. *Même dans le cas d'un petit système et d'exigences simple comme dans la section 1.1.4, si on reprend l'exemple de l'exigence E05F01, il est difficile de déterminer les classes d'équivalences des entrées car*

- *l'exigence fait appel à des variables dont les valeurs sont calculées par d'autres exigences ou de tables non disponibles au testeur ; il ne connaît que les données d'entrée et de sortie : c'est le test en boite noire,*
- *les valeurs considérées sont des valeurs réelles et le calcul est effectué avec une certaine précision, le testeur doit donc produire des tests numériques, permettant d'assurer que toutes les erreurs liées aux comportements numériques normaux et anormaux ont été vérifiés sur la cible, et que le logiciel répond aux exigences de précisions dans ces cas précis.*

1.2.4 Qualification des outils

La DO–178C décrit les activités de développement nécessaires pour effectuer la certification d'un logiciel embarqué. Pour faciliter ce développement, les ingénieurs sont amenés à utiliser des outils permettant d'automatiser certaines parties de leurs activités, que ce soit le générateur de code, le compilateur, l'éditeur de lien ou encore l'outil d'exécution automatique des procédures de test. Il faut alors procéder à la qualification de l'outil. Le processus de qualification des outils a pour but d'assurer que l'outil fournit un résultat dont le niveau de confiance est au moins équivalent à celui du ou des processus éliminés, réduits ou automatisés. La qualification d'un outil est un processus nécessaire pour obtenir un crédit de certification pour un outil logiciel dans le contexte d'un système aéroporté spécifique. Notre but étant de produire un outil permettant d'automatiser entièrement ou en partie l'activité de génération de tests, nous nous intéressons donc aux contraintes supplémentaires induites par la qualification.

Le document *DO–330 Software Tool Qualification Considerations* [21] propose des lignes directrices pouvant aider la qualification des outils logiciels dans le contexte de la certification. De la même façon que la DO–178C propose différent niveaux de certification

selon la criticité de système à valider, la DO–330 identifie 5 niveaux de qualification d'outil (Tool Qualification Level – TQL) : du plus rigoureux TQL–1 au moins rigoureux TQL–5.

Il existe trois critères permettant de déterminer le niveau de qualification requis pour permettre l'utilisation de l'outil dans le cycle de développement du logiciel embarqué. Ces critères sont les suivants :

- Critère 1 : le résultat de l'outil fait partie du logiciel sujet à certification, et pourrait donc introduire une erreur.
- Critère 2 : l'outil automatise le processus de vérification et pourrait donc échouer à détecter une erreur. Le résultat de l'outil est utilisé pour la justification de la réduction ou l'élimination d'un autre processus de vérification, ou de processus de développement.
- Critère 3 : l'outil pourrait, dans le cadre de son utilisation, échouer à détecter une erreur.

Dans le contexte de cette thèse, nous cherchons à développer un outil permettant d'automatiser certaines étapes du cycle de validation et de vérification de logiciels soumis à un niveau de certification DAL–A. Les résultats de ces tests seront utilisés pour montrer le bon fonctionnement de ces logiciels, nous nous trouvons donc dans le cas du critère numéro 2. Ceci implique que l'outil développé devra répondre aux critères de qualification TQL–4 [4].

1.3 Conclusion

Nous avons introduit dans ce chapitre le contexte du développement de logiciels embarqués critiques dans un but de certification aéronautique. Dans ce contexte, le développement, mais également toutes les activités de validation et vérification sont soumis à de fortes contraintes qui rendent la conception d'un tel logiciel complexe, coûteux et fastidieux.

Nous avons défini les contraintes spécifiques et les enjeux inhérents à la génération automatique de tests pour des logiciels numériques dans ce contexte de certification. La génération de test se découpe en plusieurs activités, dans un premier temps la modélisation des exigences haut niveau pour ensuite déterminer quelles parties du logiciel peuvent être mal implémentées et finalement trouver des points discriminants permettant de distinguer un logiciel conforme avec la spécification fonctionnel d'un logiciel contenant des erreurs.

La qualité de la campagne de test doit être évaluée, en utilisant des métriques de couverture. Il existe plusieurs types de critères de couverture, chacun permettant de vé-

4. Voir tableau 12-1 de la DO–330 [21].

rifier la qualité d'une partie différente du logiciel implémenté. Nous nous intéressons particulièrement à la définition de critères de couverture fonctionnelle.

Enfin la génération de tests est une activité particulièrement difficile lorsque les variables manipulées ne sont pas uniquement booléennes. Nous avons montré que même avec un exemple simple, il est nécessaire pour le testeur d'avoir une connaissance fine des spécificités de l'arithmétique des ordinateurs afin de générer des tests précis et fiables. Nous souhaitons donc élaborer une méthode facilitant le travail du testeur pour la génération de tests numériques.

CHAPITRE 2

ARITHMÉTIQUE DES ORDINATEURS

CERTAINS systèmes informatiques embarqués ont besoin de pouvoir représenter et manipuler efficacement des nombres réels. Cette problématique fondamentale a été abordée sous différents angles depuis les débuts de l'informatique, et de nombreuses méthodes pour approcher les nombres réels ont été proposées. Alors que nous apprenons à compter et calculer en base 10 avec des nombres réels, la représentation des nombres dans un ordinateur est soumise à des contraintes de place et de complexité d'utilisation. Ainsi, la plupart des ordinateurs actuels manipulent des données binaires, et l'arithmétique des ordinateurs binaire est la plus largement déployée [34, 35].

Les problématiques de conversion entre les formats à précision finie binaire et les formats à précision finie décimale existent depuis les débuts de l'informatique [36], mais sont cependant loin d'avoir été complètement résolues. Dans de nombreux domaines comme la finance ou la certification de logiciels en aéronautique par exemple, les calculs effectués par ordinateurs ont un besoin critique d'une représentation décimale.

Ce chapitre a pour objectif d'introduire les différents types d'arithmétique que nous allons rencontrer dans ces travaux de thèse. Dans ce chapitre, nous allons présenter les notions de base de l'arithmétique flottante dans ses variantes binaire et décimale, et différentes méthodes permettant de produire des résultats précis et/ou certifiés en précision finie. Nous fournirons une photographie des bibliothèques proposant ces différents types d'arithmétique. Enfin, nous discuterons des problématiques soulevées par l'arithmétique en précision finie dans le contexte de la certification en aéronautique.

2.1 Arithmétique à virgule flottante

Les systèmes informatiques embarqués sont soumis à de fortes contraintes de consommation d'énergie et d'efficacité de temps d'exécution en plus de la précision des calculs. Pour leur validation et vérification au contraire, c'est la contrainte de précision qui prédomine les autres. Cette étape étant assistée par ordinateur, une représentation fiable et

efficace des nombres réels sur ordinateur est donc nécessaire.

L'arithmétique à virgule flottante est une des méthodes de représentation et de manipulation des nombres réels avec une précision finie la plus largement utilisée dans les ordinateurs actuels. Elle combine efficacement les contraintes de vitesse, précision, registre d'utilisation, portabilité et facilité d'implémentation et d'utilisation, propriétés dont les enjeux sont parfois difficilement compatibles [36].

L'arithmétique à virgule flottante a été normalisée une première fois [37] en 1985 sous la forme du *IEEE 754 Standard for Binary Floating-Point Arithmetic*, publié par l'Institute of Electrical and Electronics Engineers (IEEE). Cette norme a été révisée en 2008 [38] sous le titre de *IEEE-754 Standard for Floating-Point Arithmetic*, définissant de nouveaux formats à virgule flottante binaires et décimaux. La plupart des ordinateurs actuels fournissent une arithmétique à virgule flottante conforme à la norme IEEE 754. La norme est réévaluée tous les ~ 10 ans et la révision 2018 est actuellement en cours de rédaction.

Le lecteur peut se référer au *Handbook for Floating-Point Arithmetic* par Muller et al. [36] pour trouver une étude complète et détaillée de l'arithmétique à virgule flottante.

2.1.1 Nombres à virgule flottante selon la norme IEEE 754

Dans sa révision de 2008 [38], la norme IEEE 754 définit les nombres à virgule flottante en base 2 et en base 10. Soit un format (β, p) de représentation des nombres à virgule flottante à précision finie, où β est la base telle que $\beta \in \{2, 10\}$ et p est la précision telle que $p \geq 2$. Un nombre à virgule flottante dans ce format est représenté par le triplet (s, m, e) tel que [36, 35]

$$x = (-1)^s \cdot m \cdot \beta^e, \tag{2.1}$$

où

- $s \in \{0, 1\}$ est le bit de signe,
- $e \in [e_{\min}, e_{\max}]$ est l'exposant, avec $e_{\min} < 0 < e_{\max}$, $e_{\max}$ étant spécifié pour chaque formats et $e_{\min} = 1 - e_{\max}$,
- m est la mantisse représentée par p chiffres significatifs, sous la forme du nombre à virgule $m_0.m_1 m_2 \ldots m_{p-1}$, telle que $0 \leq |m| < \beta$ et $m_i \in \{0, \ldots, \beta - 1\}$.

En pratique, la mantisse est représentée en machine dans le format à virgule flottante par m' défini tel que $m' = m \cdot \beta^{p-1}$. La représentation (m', e) d'un nombre à virgule flottante n'est pas forcément unique. Par exemple, pour $\beta = 10$ et $p = 3$, le nombre 63 peut être représenté soit comme $63 \cdot 10^0$ ou comme $630 \cdot 10^{-1}$, les mantisses 63 et 630 étant représentables car inférieures à $\beta^p = 1000$. Une *cohorte* est l'ensemble de ces représentations à virgule flottante dans un même format (β, p) équivalentes à une même valeur.

La *normalisation* des nombres à virgule flottante non-nuls permet d'obtenir une représentation unique pour chaque valeur dans un format donné. Cette représentation satisfait $1 \leq |m| < \beta$, c'est-à-dire $\beta^{p-1} \leq |m'| \leq \beta^p - 1$. En base 2, la normalisation permet de gagner un bit dans la représentation, le premier bit de la mantisse normalisée étant un bit implicite fixé à 1, tel que $m_0.m_1 m_2 \dots m_{p-1} = 1.m_1 m_2 \dots m_{p-1}$.

Dans le cas où $e = e_{\min}$ et $|m| < 1$, la représentation des nombres à virgule flottante est dite *dénormalisée*. Dans ce cas, la mantisse de la représentation binaire de nombres à virgule flottante est toujours de la forme $m_0.m_1 m_2 \dots m_{p-1} = 0.m_1 m_2 \dots m_{p-1}$, le bit implicite étant fixé à 0.

Les nombres à virgule flottante décimaux tels que décrits dans la norme IEEE 754 [38] quant à eux ne bénéficient pas toujours d'attribut de normalisation. Le résultat d'une opération à virgule flottante décimale est différencié des autres membres de sa cohorte selon un *exposant privilégié*. Cet exposant est choisi tel que la valeur de l'unité en dernière position de la mantisse, est préservée si le résultat de l'opération est exact. Cette valeur est appelée le *quantum* de la représentation et est définie par β^{e-p+1} [38].

	binary32	binary64	decimal64
taille mémoire (en bits)	32	64	64
précision $p = k_2$, resp. k_{10}	24 bits	53 bits	16 chiffres
$e_{\max}$	127	1023	384

Tableau 2.1 – Formats IEEE 754 à virgule flottante

Le tableau 2.1 décrit les valeurs de deux formats à virgule flottante binaire et un format décimal. Ces formats sont des formats standards dont les différents champs d'exposant et de mantisse sont définis de façon détaillée dans la norme IEEE 754-2008 [38]. On remarque que la précision p se compte en nombre de bits en binaire et en nombre de chiffres significatifs en décimal.

On peut ainsi définir les deux ensembles de nombres à virgule flottante que nous allons considérer dans cette thèse : l'ensemble des formats binaires et l'ensemble des formats décimaux. La précision p se distingue donc par $p = k_2$ en binaire et $p = k_{10}$ en décimal. L'ensemble des nombres à virgule flottante binaire normalisés à la précision k_2 est $\left\{ m \cdot 2^E \;\middle|\; e_{\min} \leq E \leq e_{\max},\ 2^{k_2-1} \leq |m| \leq 2^{k_2} - 1 \right\}$. L'ensemble des nombres dénormalisés à la précision k_2 est $\left\{ m \cdot 2^E \;\middle|\; E = e_{\min},\ 1 \leq |m| \leq 2^{k_2-1} - 1 \right\}$.

Soit un système à virgule flottante (β, p). Notons $\mathbb{F}_p^\beta \subset \mathbb{R}$ l'ensemble des nombres à virgule flottante dans ce système. Étant donné les définitions [36] des "extrema" des nombres à virgule flottante, tels que

- la plus petite valeur normale est $\beta^{e_{\min}}$,
- la plus grande valeur finie est $\beta^{e_{\max}} \cdot \left(\beta - \beta^{1-p}\right)$,
- la plus petite valeur dénormalisée est $\beta^{e_{\min}-p+1}$,

nous pouvons définir les ensembles des nombres flottants normaux et dénormalisés binaires en conséquence tels que

$$\mathbb{F}_{k_2}^2 \;=\; \left\{ m \cdot 2^E \;\middle|\; E^{\min} \leq E \leq E^{\max},\; 1 \leq |\, m \,| \leq 2^{k_2} - 1 \right\} \bigcup \left\{ 0 \right\}. \tag{2.2}$$

De la même façon, on peut définir l'ensemble des nombres à virgule flottante décimaux à la précision k_{10} tel que

$$\mathbb{F}_{k_{10}}^{10} \;=\; \left\{ n \cdot 10^F \;\middle|\; F^{\min} \leq F \leq F^{\max},\; 1 \leq |\, n \,| \leq 10^{k_{10}} - 1 \right\} \bigcup \left\{ 0 \right\}. \tag{2.3}$$

Dans cette thèse, nous distongons $e_{\min}$ resp. $e_{\max}$ dans les formats binaires et décimaux par la notation $E^{\min}$, resp. $E^{\max}$, en binaire et $F^{\min}$, resp. $F^{\max}$ en décimal.

Les formats à virgule flottante décrits dans la norme IEEE 754-2008 présentent en outre des valeurs spéciales, pour lesquelles les opérations à virgule flottante ont des comportements particuliers. On note entre autres : les zéros signés ($+0$ et -0), les infinis ($+\infty$ et $-\infty$), et la valeur signalant qu'une opération a été effectuée avec des données invalides (par exemple $\frac{0}{0}$), appelée "Not a Number", ou `NaN`.

De plus lors d'une opération invalide, une *exception* est levée sous la forme d'un drapeau de statut. On peut compter parmi les exceptions définies dans la norme IEEE 754, en plus de l'exception `invalid`, l'exception de division par zéro, ainsi que les exceptions `overflow` et `underflow`. L'exception `overflow` survient lorsque l'exposant du résultat arrondi est supérieur à $e_{\max}$. L'exception `underflow` est signalée lorsqu'un résultat non-nul est inférieur à $\beta^{e_{\min}}$ et est inexact. Une exception est également levée lorsqu'un résultat n'est pas exactement représentable, c'est l'exception `inexact`. Ce résultat est alors arrondi.

2.1.2 Arrondis

Les nombres à virgule flottante sont donc une représentation discrète des nombres réels. L'*arrondi* d'un nombre réel est une règle de représentation de nombres tels que les nombres réels ou des nombres à virgule flottante quelconques dans un ensemble des nombres à virgule flottante choisi $\mathbb{F}$. Cette représentation doit satisfaire des critères de nombre de chiffres de précision ainsi que la façon de décider de ces chiffres : c'est la règle d'arrondi. En général, un arrondi a lieu sur le résultat d'une opération sur des nombres à virgules flottantes, celui-ci n'étant pas forcément représentable dans le système à virgule flottante choisi, c'est-à-dire que ce résultat contient plus de chiffres qu'autorisés par la règle d'arrondi. L'arrondi est effectué selon le *mode d'arrondi* choisi dans le système à

virgule flottante. La norme IEEE 754-2008 définit l'arrondi d'un résultat infiniment précis au nombre à virgule flottante à la précision p par les modes d'arrondis suivants, tels qu'illustrés dans la figure 2.1 :

- l'arrondi vers $+\infty$: $\triangle_p(x) \in \mathbb{F}$ est le plus petit nombre à virgule flottante possible, incluant $+\infty$ qui est plus grand ou égal à x,
- l'arrondi vers $-\infty$: $\triangledown_p(x) \in \mathbb{F}$ est le plus grand nombre à virgule flottante possible, incluant $-\infty$ qui est plus petit ou égal à x,
- l'arrondi vers zéro : $\bullet_p(x) \in \mathbb{F}$ est le nombre à virgule flottante le plus proche de x et dont la magnitude n'est pas plus grande que celle de x. En pratique, si $x > 0$ alors $\bullet_p(x) = \triangledown_p(x)$, et si $x < 0$ alors $\bullet_p(x) = \triangle_p(x)$.
- l'arrondi vers le plus proche : $\diamond_p(x) \in \mathbb{F}$ est le nombre à virgule flottante le plus proche de x. Dans le cas où x se trouve être exactement le milieu entre deux nombres à virgule flottante à précision finie, on définit une règle de décision (les deux définies dans la norme IEEE 754-2008 sont l'arrondi vers le nombre pair le plus proche ou encore l'arrondi vers zéro [38]).

Pour l'étude formelle des opérations à virgule flottante, on note $\circ_p(x) \in \{\triangle_p, \triangledown_p, \bullet_p, \diamond_p\}$ l'arrondi de x selon l'un de ces modes d'arrondis.

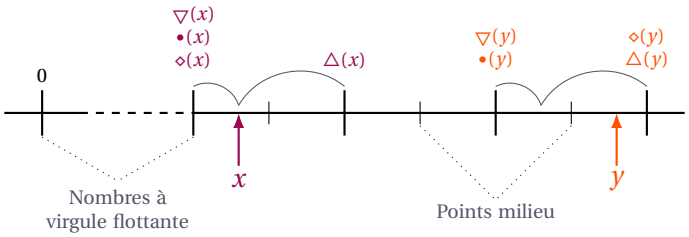


FIGURE 2.1 – Arrondi des nombres réels x et y selon différents modes d'arrondi

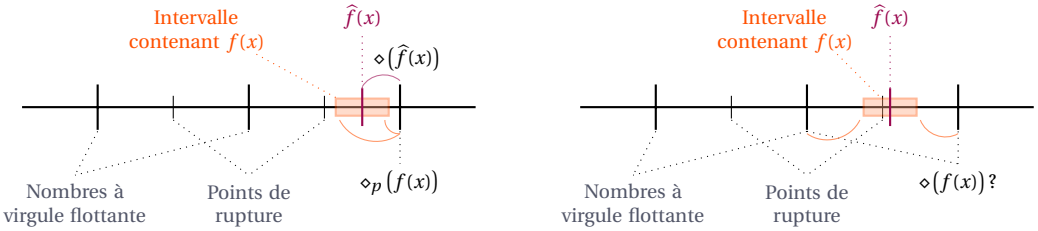
Le milieu entre deux nombres à virgule flottante à une précision donnée s'appelle le *point milieu*. Déterminer l'arrondi au plus proche d'un nombre infiniment précis revient à déterminer de quel côté du point milieu le plus proche il se situe.

Si le résultat d'une opération à virgule flottante à précision finie est arrondi au même nombre flottant que le résultat de la même opération calculée avec une précision infinie puis arrondi au format à virgule flottante choisi, cette opération effectue un *arrondi correct*. Une opération effectue un *arrondi fidèle* lorsque son arrondi est le même que les arrondis du résultat exact y tel que $\triangle(y)$ ou $\triangledown(y)$. Un arrondi fidèle est une relation et non une fonction. La norme IEEE 754-2008 exige des opérations arithmétiques standard qu'elles soient correctement arrondies.

Fournir une arithmétique à virgule flottante avec arrondi correct comporte des avantages, notamment l'amélioration de la portabilité des logiciels, et présenter des propriétés nécessaires pour implémenter l'arithmétique d'intervalles (voir section 2.2.1). Toutefois produire une arithmétique avec arrondi correct pour toutes les fonctions mathématiques usuelles, comme l'exponentielle ou le sinus, n'est cependant pas une tâche simple. L'arrondi correct est coûteux à déterminer de façon systématique ; cela requiert la recherche du pire cas pour l'arrondi dans le mode d'arrondi courant, ce qui est difficile notamment lorsqu'on est confronté au *Dilemme du fabriquant de table* (TMD) [36] (voir figure 2.2).

Supposons que nous cherchons à approcher la fonction réelle f dans le format à virgule flottante (β, p) par une fonction $\widehat{f}$ telle que $\circ_p(\widehat{f}(x)) = \circ_p(f(x))$. Selon le mode d'arrondi choisi $\circ_p$, les *points de rupture* selon ce mode d'arrondi sont les valeurs pour lesquelles la direction de l'arrondi change. Ainsi dans le cas de l'arrondi au plus proche $\diamond_p$, ces points de rupture sont les points milieux entre les points à virgule flottante de précision p, et dans le cas des autres modes d'arrondi, ces points sont les points à virgule flottante eux mêmes.

Soit x un nombre à virgule flottante, le nombre réel $f(x)$ ne peut généralement pas être représenté avec un nombre fini de chiffres significatifs. L'approximation $\widehat{f}(x) = f(x) \cdot (1+\varepsilon)$ calculée par des opérations à virgule flottante peut être très précise, mais va généralement contenir une erreur par rapport au résultat $f(x)$ (voir figure 2.2). Nous verrons dans la section 2.1.3 de quelle façon nous pouvons quantifier formellement ces erreurs.



(a) Quand il n'y a pas de point de rupture dans l'intervalle contenant $f(x)$

(b) Quand il y a un point de rupture dans l'intervalle contenant $f(x)$

FIGURE 2.2 – Dilemme du fabriquant de table pour l'arrondi correct : illustration de l'arrondi au plus proche de $\widehat{f}(x)$

La seule information que nous avons sur $f(x)$ est donc que la solution exacte de cette fonction que l'on approche avec la fonction à virgule flottante $\widehat{f}(x)$ se trouve dans un intervalle de largeur deux fois l'erreur absolue de $\widehat{f}$. Le TMD survient lorsque cet intervalle d'erreur contient un point de rupture, c'est-à-dire lorsque le résultat calculé $\widehat{f}(x)$ est si proche d'un point de rupture, par rapport à son erreur, qu'il est impossible de décider de quel côté du point de rupture se trouve le résultat réel $f(x)$. Dans le cas illustré dans la figure 2.2, le point de rupture pour le mode d'arrondi au plus proche est le point milieu

entre deux nombres à virgule flottante. Une solution au TDM est d'augmenter la précision de calculs de l'algorithme permettant de déterminer $\widehat{f}(x)$.

2.1.3 Analyse d'erreur

Les erreurs d'arrondi sont une partie inhérente au calcul en précision finie. Un des enjeux de l'arithmétique des ordinateurs est d'estimer les erreurs survenant lors des opérations en précision finie afin d'optimiser la qualité des algorithmes numériques, c'est-à-dire effectuer l'analyse d'erreur de ces algorithmes numériques. On distingue deux sens du mot précision : la précision p d'un format définissant la taille de la représentation et la précision de calculs (*accuracy* en anglais [1]) représentant le nombre de chiffres significatifs d'un résultats qui ne sont pas entâchés d'une erreur. Connaissant le résultat exact en précision infinie x, on peut évaluer l'*erreur absolue* et l'*erreur relative* [36] sur son approximation à la précision p telles que

$$\begin{aligned} &\text{erreur absolue :} && \Delta_x \;=\; \left| \, x - \circ_p(x) \, \right|, \\ &\text{erreur relative :} && \varepsilon_x \;=\; \left| \frac{x - \circ_p(x)}{x} \right|, \;\text{si } x \neq 0. \end{aligned} \tag{2.4}$$

Dans certains cas, l'expression de l'erreur de calcul comme une erreur relative ne contient pas suffisamment d'information. On souhaite donc introduire un moyen d'exprimer les erreurs selon une mesure du poids du dernier bit de la mantisse. Ce concept est formalisé sous le terme *ulp*, ou *unit in the last place*.

Définition 4 (Unit in the last place – ulp [39, 35, 36])**.** *Selon Goldberg [35], étant donné le format à virgule flottante* (β, p)*, si on a* $|\, x \,| \in \left[\beta^e, \beta^{e+1}\right[$*, alors la fonction* $\mathrm{ulp}_p(x)$ *est définie telle que*

$$\mathrm{ulp}_p(x) \;=\; \beta^{\max(e, e_{\min}) - p + 1} \tag{2.5}$$

Cette définition de l'ulp coïncide avec la définition du *quantum* pour un nombre à virgule flottante. Elle caractérise la distance séparant deux nombres flottants consécutifs ayant la même valeur d'exposant. En binaire on appelle *binade* l'intervalle $\left[2^e, 2^{e+1}\right[$ sur lequel la fonction $ulp(x)$ prend une même valeur. De la même façon, en décimal l'intervalle $\left[10^e, 10^{e+1}\right[$ est appelé *décade*. On peut déduire de cette définition de l'ulp plusieurs propriétés sur l'arrondi au plus proche de nombres réels et de nombres à virgule flottante utiles pour l'analyse d'erreur des algorithmes numériques [36, 35, 39].

On peut facilement convertir une erreur déterminée en fonction du nombre d'ulps en une erreur relative, d'après [39]. Supposons que $\left| \, x - \circ_p(x) \, \right| \;=\; \alpha \cdot \mathrm{ulp}_p(x)$ et qu'il n'y a pas

1. On distingue la précision de calculs (accuracy) de la précision de représentation (precision).

d'underflow, on a alors que

$$\left| \frac{x - \circ_p(x)}{x} \right| \leq \alpha \cdot \beta^{-p+1}. \tag{2.6}$$

Étant donné la définition de l'ulp par Goldberg (définition 4), on peut définir l'*unité d'arrondi* ou *epsilon machine* [36], notée u, d'un système à virgule flottante (β, p), telle que

$$u = \begin{cases} \frac{1}{2}\text{ulp}_p(1) = \frac{1}{2}\beta^{1-p} & \text{pour l'arrondi au plus proche,} \\ \text{ulp}_p(1) = \beta^{1-p} & \text{pour les arrondis dirigés.} \end{cases} \tag{2.7}$$

On peut remarquer que lorsque $\beta = 2$, on a que $u = 2^{-p}$ pour l'arrondi au plus proche. Les nombres à virgule flottante étant répartis de façon non-uniforme, ils sont en effet plus dense autour de zéro que lorsqu'ils sont relativement grands, l'erreur relative est un bon outil pour estimer l'impact d'une erreur sur un résultat qu'il soit proche de zéro ou non. En général, il est utile d'exprimer l'erreur absolue en terme d'ulps et l'erreur relative en terme d'unités d'arrondi.

On peut ainsi définir l'erreur relative de représentation ε_x d'un nombre réel $x \in \mathbb{R}$ dans un format à virgule flottante (β, p) en fonction de l'unité d'arrondi u, telle que

$$\circ_p(x) = x(1+\varepsilon_x), \qquad |\varepsilon_x| \leq u. \tag{2.8}$$

En l'absence d'underflow et d'overflow, une modélisation des erreurs des opérations à virgule flottante communément acquise, pour deux nombres à virgule flottante x et y est la suivante :

$$\circ_p(x \text{ op } y) = (x \text{ op } y)\cdot(1+\varepsilon), \qquad |\varepsilon| \leq u, \qquad \text{op} \in \{+, -, \times, \div\}. \tag{2.9}$$

Un type particulier d'erreurs numériques survenant au niveau des opérations arithmétiques de base, notamment la soustraction, est l'*élimination*, appelée également *élimination catastrophique* lorsque presque tous les chiffres significatifs du résultat sont perdus. Cette erreur se produit lorsque les deux sont presque égaux. La soustraction de ces deux nombres n'introduit pas d'erreur en elle-même, mais propage et amplifie une erreur préexistante.

Dans cette thèse, nous nous intéressons à l'analyse d'algorithmes numériques de façon à maîtriser et si possible éviter les effets indésirables relatifs à la propagation des erreurs numériques comme l'élimination. Dans nos contributions, nous déterminerons des bornes rigoureuses sur les erreurs de calculs et leur impacts sur les résultats de ces algorithmes et leurs implémentations.

2.2 Vers une arithmétique plus précise

Il existe de nombreuses méthodes permettant de fournir au moins autant, sinon plus de précision (accuracy) pour le résultat de calculs en précision finie que l'arithmétique à virgule flottante classique. On peut notamment mentionner l'arithmétique à virgule flottante à précision étendue formalisée dans la norme IEEE 754-2008 [38], ou encore les algorithmes compensés reposant sur les transformations exactes ("error-free transformations", EFTs) [36]. Ces derniers utilisent des propriétés d'opérations exactes de l'arithmétique flottante pour estimer les erreurs de calculs au fur et à mesure des opérations pour compenser le résultat final.

Dans cette section, nous présentons deux approches différentes pour gérer les erreur de calculs survenant lors des opérations à virgule flottante. Cette étude des méthodes utilisées pour fournir une arithmétique plus précise, loin d'être exhaustive, se concentre sur deux méthodes développées dans ces travaux de thèses. Dans un premier temps, nous présenterons l'*arithmétique d'intervalles*, une méthode permettant de maîtriser explicitement les erreurs de calculs. En second lieu, nous étudierons l'*arithmétique multi-précision*, et nous terminerons cette section par la description d'un échantillon des bibliothèques arithmétiques existantes à l'heure actuelle.

2.2.1 Arithmétique d'intervalles

Le terme "arithmétique d'intervalles" fut établi dans les années 1950 par Moore [7]. L'arithmétique d'intervalles est une des méthodes permettant de localiser et quantifier les erreurs de calculs, par exemple celles qui surviennent inévitablement avec l'utilisation d'approximations telles que les nombres flottants. Le lecteur peut se référer à [40, 41] pour plus de détails sur l'arithmétique d'intervalles.

Une valeur x représentée par un intervalle fermé, borné et non-vide, s'écrit sous la forme

$$\left[\underline{x}, \overline{x}\right] \;=\; \left\{ x \in \mathbb{R} \mid \underline{x} \;\leq\; x \;\leq\; \overline{x} \right\}. \tag{2.10}$$

Un intervalle $\left[\underline{x}, \overline{x}\right]$ est donc un ensemble connexe de $\mathbb{R}$ qui est représenté par sa borne inférieure $\underline{x}$ et sa borne supérieure $\overline{x}$ telles que $\underline{x} \;\leq\; \overline{x}$. Dans le cas où $\underline{x} \;=\; \overline{x}$, l'intervalle $\left[\underline{x}, \overline{x}\right]$ est un point. La représentation des intervalles par leurs bornes est une représentation couramment utilisée. Il existe différentes représentations des intervalles, comme par exemple la représentation *centre-rayon* d'un intervalle $\left[\underline{x}, \overline{x}\right]$ par les valeurs $\langle x_c, x_r \rangle$ où $x_c \;=\; \frac{\overline{x}+\underline{x}}{2}$ est le centre de l'intervalle et $x_r \;=\; \frac{\overline{x}-\underline{x}}{2}$ est le rayon de l'intervalle.

Décrire les nombres à l'aide d'intervalles permet de les représenter en prenant en compte l'erreur introduite par la représentation en précision finie. Ceci est particulièrement utile lorsque l'on souhaite utiliser des nombres qui ne sont pas exactement représentables

dans le format utilisé en précision finie. Par exemple le nombre 0.1 n'est pas représentable exactement en format binaire à virgule flottante quelle que soit la précision choisie. De même π est un nombre transcendant et par définition s'écrit avec un nombre de chiffres significatifs infini s'il est représenté dans une autre base que la base π.

Les opérations sur les intervalles peuvent être définies à partir de ce modèle de représentation. Étant donné deux intervalles $\left[\underline{u},\overline{u}\right]$ et $\left[\underline{v},\overline{v}\right]$, le résultat d'une opération sur ces intervalles, notée op $\in \{+,-,\times,\div,\cdots\}$ est défini comme l'ensemble convexe suivant :

$$\left[\underline{u},\overline{u}\right] \text{ op } \left[\underline{v},\overline{v}\right] = \text{conv}\{u \text{ op } v \mid \forall u \in \left[\underline{u},\overline{u}\right], \forall v \in \left[\underline{v},\overline{v}\right]\} \tag{2.11}$$

Ces opérations satisfont la *propriété d'inclusion* [36], c'est-à-dire que le résultat d'une opération mathématique sur les intervalles est inclus dans un intervalle qui est une combinaison des intervalles d'entrée. Il est donc possible d'étendre les opérations arithmétiques sur les intervalles en les déduisant des propriétés de monocité des opérations arithmétiques sur les nombres réels telles que

$$\begin{aligned}
-\left[\underline{u},\overline{u}\right] &= \left[-\overline{u},-\underline{u}\right], \\
\left[\underline{u},\overline{u}\right]^{-1} &= \left[\overline{u}^{-1},\underline{u}^{-1}\right] \text{ si les deux bornes sont positives,} \\
\left[\underline{u},\overline{u}\right] + \left[\underline{v},\overline{v}\right] &= \left[\underline{u}+\underline{v},\overline{u}+\overline{v}\right], \\
\left[\underline{u},\overline{u}\right] \times \left[\underline{v},\overline{v}\right] &= \left[\min\left(\underline{u}\times\underline{v},\underline{u}\times\overline{v},\overline{u}\times\underline{v},\overline{u}\times\overline{v}\right), \max\left(\underline{u}\times\underline{v},\underline{u}\times\overline{v},\overline{u}\times\underline{v},\overline{u}\times\overline{v}\right)\right].
\end{aligned} \tag{2.12}$$

Les modèles mathématiques courants de l'arithmétique d'intervalle, tels que décrits ci-dessus et les opérations arithmétiques de base pour l'arithmétique d'intervalles selon ces modèles sont spécifiées dans la norme IEEE 1788-2015 *Standard for Interval Arithmetic* [42], et sa révision et simplification IEEE 1788-2017. Cette dernière norme examine les difficultés pratiques de l'application de l'arithmétique d'intervalle dans un système arithmétique à virgule flottante, conformément à la norme IEEE 754-2008, en prenant en considération la cohérence entre le modèle d'intervalles et les considérations pratiques de développement.

Il est donc possible de représenter une valeur x n'ayant pas de représentation exacte dans un système à virgule flottante (β, p) par un intervalle, tel que

$$[x] = \left[\underline{x},\overline{x}\right] = \left\{ x \in \mathbb{R} \mid \underline{x} \leq x \leq \overline{x} \mid \underline{x},\overline{x} \in \mathbb{F}_p^\beta \right\}. \tag{2.13}$$

Les opérations sur les intervalles représentés par des nombres flottants satisfont la propriété d'inclusion grâce aux propriétés des arrondis dirigés vers $+\infty$ et $-\infty$. Elles sont définies telles que si le résultat de l'opération réelle $\text{op}_\mathbb{R}$ sur un intervalle de nombres à virgule flottante $\left[\underline{x},\overline{x}\right]$ est $\text{op}_\mathbb{R}\left(\left[\underline{x},\overline{x}\right]\right) = \left[\underline{u},\overline{u}\right]_\mathbb{R}$, alors l'opération équivalente dans le système à virgule flottante (β, p) est

$$\text{op}_{\mathbb{F}_p^\beta}\left(\left[\underline{x},\overline{x}\right]\right) = \left[\bigtriangledown_p\left(\underline{u}\right), \bigtriangleup_p\left(\overline{u}\right)\right]_{\mathbb{F}_p^\beta}. \tag{2.14}$$

On peut donc fournir un intervalle contenant le résultat exact d'une opération à virgule flottante en modifiant les modes d'arrondi lors du calcul de ses bornes. Utiliser les arrondis dirigés permet d'encadrer avec certitude le résultat ainsi que les erreurs de calcul pouvant survenir lors de son calcul.

Cependant, les bornes de l'intervalle d'un résultat calculé par cette méthode ne sont pas forcément fines, c'est-à-dire que l'erreur absolue associée à ce résultat peut être bien plus petite que la largeur de l'intervalle. Un des problèmes survenant avec une utilisation naïve de l'arithmétique d'intervalles est le phénomène de *décorrélation des variables*, ou *problème de dépendance*. Ce problème est dû au fait que l'arithmétique d'intervalles ne peut pas garder de trace de la corrélation entre plusieurs occurrences d'une même variable. Prenons par exemple la fonction $f(x) = \frac{x+1}{\sqrt{x}}$ que nous souhaitons évaluer sur l'intervalle $\left[\underline{x}, \overline{x}\right] = [1,4]$. La fonction étant croissante sur cet intervalle, on s'attend à ce que l'évaluation sur cet intervalle soit le résultat $\left[f\left(\underline{x}\right), f\left(\overline{x}\right)\right] = [2,2.5]$. L'évaluation suivant les règles de l'arithmétique d'intervalles nous renvoie le résultat $f\left(\left[\underline{x}, \overline{x}\right]\right) = [1,5]$, qui est un intervalle plus grand que le résultat attendu. On peut contourner ce problème en réécrivant l'expression selon les techniques d'écritures décrites dans [43]. Dans l'exemple précédent, nous pouvons réécrire la fonction $f(x)$ telle que $f_{\text{bis}}(x) = \sqrt{x} + \frac{1}{\sqrt{x}}$. L'évaluation par l'arithmétique d'intervalle nous donne ainsi $f_{\text{bis}}\left(\left[\underline{x}, \overline{x}\right]\right) = [1.5,3]$, qui est un plus petit intervalle que le résultat précédent.

Un autre phénomène associé à l'arithmétique d'intervalles menant à une surestimation de la largeur de l'intervalle résultat par rapport à l'erreur de calcul est le *phénomène d'enveloppement*. La figure 2.3 illustre cet effet sur un exemple de rotation d'un carré. Dans la première figure à gauche, le carré est représenté par deux intervalles. Après la première rotation à 45°, les bornes des intervalles permettant de représenter tous les points du carré sont réévaluées au second carré hachuré enveloppant le premier. La seconde rotation à 45° est effectuée sur les deux intervalles estimant la première rotation, c'est-à-dire le carré hachuré. Le résultat est deux intervalles représentant un carré encore plus grand que les deux précédents. On peut pallier à l'effet d'enveloppement en fractionnant les intervalles sur lesquels on effectue une opération pour ensuite faire l'union des intervalles résultats. Cette méthode est appelée bissection d'intervalles.

2.2.2 Arithmétique multi-précision

Il arrive que quantifier l'erreur de calcul ne soit pas une méthode suffisante : l'erreur de calcul peut être petite en valeur absolue mais toujours relativement grande par rapport au résultat exact. Une méthode possible pour augmenter la précision (accuracy) du résultat d'un algorithme est de travailler avec une précision de représentation intermédiaire plus grande. Il existe plusieurs méthodes permettant d'augmenter la précision de calculs, aussi

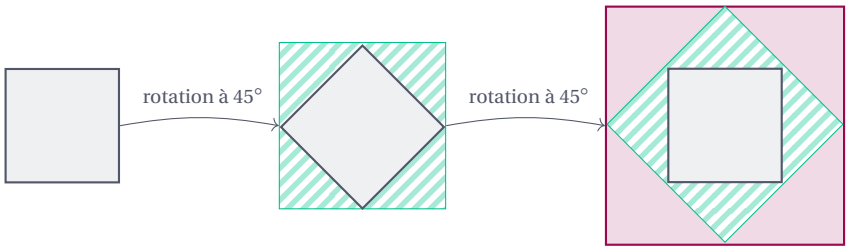


FIGURE 2.3 – Effet d'enveloppement après deux rotations de 45° d'un carré

bien disponibles au niveau du matériel qu'émulée par un logiciel. Parmi ces méthodes on peut remarquer l'arithmétique exacte utilisant les nombres rationnels ou les fractions continues [44], l'accumulateur long de Kulisch [45], les expansions en virgule flottante [46] ou encore l'arithmétique multi-précision [47, 11].

La norme IEEE 754-2008 prévoit l'utilisation de formats avec une précision plus grande. Selon la norme, un format de précision étendue est un format qui étend un format de base pris en charge avec une précision et un intervalle de définition plus grands. Certains processeurs sont pourvus de registres flottants, stockant des nombres à virgule flottante représentés en double précision étendue, c'est-à-dire sur 80 bits. Ils effectuent systématiquement tous les calculs avec des variables de 80 bits, soit 64 bits de précision, indépendamment de la précision (simple ou double) dans laquelle sont représentées les opérandes.

Mais utiliser la précision étendue n'est pas toujours la solution la plus polyvalente lorsqu'on cherche à augmenter la précision d'une opération particulière dans un algorithme. Dans le cadre de la validation et vérification, nous devrons modifier la précision de manière non-homogène, c'est-à-dire que différentes variables peuvent avoir une précision différente. De plus la précision des variables internes de nos algorithmes doit pouvoir être adaptée de façon dynamique. Nous pouvons utiliser l'arithmétique en virgule flottante multi-précision pour satisfaire ces contraintes de précision,. Cette arithmétique utilise un format dont le nombre de chiffres significatifs est limité uniquement par l'espace mémoire disponible. Dans cette arithmétique, un nombre est exprimé dans une certaine base par une mantisse dont la précision est déterminée par l'utilisateur et un exposant.

2.2.3 Paysage des bibliothèques arithmétiques

Il existe de nombreuses bibliothèques arithmétiques mettant à disposition ces méthodes de calculs plus sûres que l'arithmétique à virgule flottante classique. Notre but n'est pas d'en faire une liste exhaustive, mais de remarquer les bibliothèques et outils pertinents dans le contexte de cette thèse. Dans nos travaux, nous utilisons deux types d'arithmétiques à virgule flottante : l'arithmétique multi-précision pour augmenter la précision et

l'arithmétique d'intervalles pour la fiabilité des résultats, indispensable dans le contexte de la certification. Notre but est de faire le lien entre les exigences et le code source par le test. Les valeurs et les erreurs définies dans les exigences sont exprimées sous forme décimale, tandis que le code source effectue ses opérations en binaire, comme illustré dans les exemples de la section 2.3. Nous nous concentrerons donc également sur les bibliothèques fournissant une arithmétique certifiée dans ces deux bases.

Les bibliothèques multi-précision GMP, MPFR L'arithmétique multi-précision est implémentée dans différents langages de programmation de haut niveau comme Python ou Java sous forme de `Bignum` [48]. La plupart de ces bibliothèques reposent cependant sur la même base : ce sont des interfaces pour la bibliothèque GNU Multiple Precision Arithmetic en C, ou GMP [2] [49]. Cette bibliothèque fournit une arithmétique multi-précision exacte et rapide sur les entiers signés (de type `mpz_t`), les nombres rationnels (de type `mpz_q`) et les nombres à virgule flottante (de type `mpz_f`, aux opérations rudimentaires). La bibliothèque GMP fournit un riche catalogue de fonctions définies dans une interface normalisée, c'est-à-dire que chaque prototype de fonction GMP suit la même règle de définition, telle que `mpz_function_name`(output, inputs). La seule limitation pratique aux opérations GMP est la mémoire disponible pour les calculs.

La bibliothèque MPFR [11] est une extension multi-précision à virgule flottante binaire de GMP, offrant fiabilité, rapidité et conformité avec la norme IEEE 754-2008. Cette bibliothèque fournit un catalogue de fonctions plus développé que les nombres à virgule flottante multi-précision de GMP, tout en étant optimisée et assurant une arithmétique à virgule flottante multi-précision arrondie correctement. Chaque nombre MPFR est représenté par une mantisse binaire entière multi-précision, un exposant entier, un signe, et une précision binaire de représentation de la sortie.

Les prototypes des fonctions MPFR sont conformes aux règles d'interfaces définies par la sémantique de GMP. Ce sont des bibliothèques écrites en C permettant d'émuler l'arithmétique multi-précision bas niveau, conçues pour être rapides et fiables.

Arithmétique d'intervalles La bibliothèque MPFI [50] basée sur la bibliothèque MPFR fournit une arithmétique d'intervalles sur les nombres à virgule flottante binaires multi-précision. Les intervalles MPFI sont représentés par leurs deux bornes inférieure et supérieure qui sont des nombres à virgule flottante multi-précision MPFR. Les propriétés d'arrondi correct fournies par les arrondis dirigés des fonctions MPFR permettent de calculer les bornes des intervalles en multi-précision. MPFI est écrite en C et fournit des fonctions aux prototypes suivant les règles d'interfaces définies par la sémantique de GMP.

INTLAB [51] est une boite à outils de MATLAB/Octave fournissant une arithmétique

2. `https://gmplib.org`

d'intervalles multi-précision. INTLAB prend en charge les intervalles réels et complexes, mais également les vecteurs, matrices denses et creuses. INTLAB est conçu pour être très rapide, en particulier sur des algorithmes optimisés pour les matrices d'intervalles. Cet outil utilise également l'interface de programmation interactive de MATLAB. Le principal objectif d'INTLAB est de produire rapidement des résultats fiables tout en maintenant une interface utilisateur à haut niveau. L'utilisateur n'a pas besoin de s'attarder sur les modes d'arrondi ni les types des variables. L'arithmétique d'intervalle est activée par la conversion de nombres en virgule flottante à l'aide du programme de conversion `str2intval`.

Lorsque l'on pense à l'informatique performante et rapide, on pense d'abord aux langages de bas niveau. Cependant, les langages de haut niveau, comme Java ont leurs avantages. La bibliothèque Jinterval [52] fournit un moyen d'utiliser l'arithmétique d'intervalles avec un langage de haut niveau (Java), tout en gardant des performances de vitesse compétitives. Jinterval offre une arithmétique d'intervalle sur les nombres rationnels, représentés avec la classe `ExtendedRational`. Ce choix de représentation est motivé par la contrainte de construire un système algébrique fermé pour ces opérations arithmétiques. Jinterval est une bibliothèque rapide et efficace, assurant la conformité à la norme IEEE 1788-2015 [42]. Jinterval fournit également plusieurs représentations d'intervalle, telles que des intervalles classiques, des intervalles de Kaucher [53, 54] ou des intervalles complexes rectangulaires ou circulaires. Grâce à cette souplesse dans le choix du type d'intervalles, l'utilisateur peut facilement passer d'une arithmétique à une autre si elles sont compatibles.

Arithmétique à virgule flottante multi-précision décimale Nous nous intéressons également aux bibliothèques d'arithmétique à virgule flottante multi-précision *décimales*. Par l'évaluation de ces bibliothèques, nous pourrons déterminer si l'arithmétique multi-précision décimale est une alternative suffisamment efficace à l'arithmétique multi-précision binaire couplée à une analyse d'erreur.

Les classes Java `BigInteger`[3] et `BigDecimal`[4] [55] sont deux des classes Java qui fournissent des arithmétiques multi-précision, l'une entière et l'autre à virgule flottante décimale. La classe `BigDecimal` représente les nombres selon une mantisse entière multi-précision, représentée grâce à la classe `BigInteger`, un exposant entier, et un second entier `scale` donnant le nombre de chiffres significatifs à droite de la virgule. La précision du résultat d'une opération et son mode d'arrondi sont fournis par un constructeur `MathContext`. Cette représentation présente des caractéristiques intéressantes, comme sa facilité d'utilisation et de l'interface avec le langage Java, mais la panoplie de fonctions arithmétiques implémentées dans ces classes est très limitée. On peut trouver les opérations arithmétiques élémentaires comme l'addition, la multiplication et la racine carrée, mais

3. `https://docs.oracle.com/javase/7/docs/api/java/math/BigInteger.html`
4. `https://docs.oracle.com/javase/7/docs/api/java/math/BigDecimal.html`

ces classes ne fournissent pas les fonctions exponentielle, logarithme et trigonométriques.

La bibliothèque *mpdecimal* [56] se présente comme fournissant une arithmétique multi-précision décimale en C. Cette bibliothèque se distingue par ses performances, elle est rapide et propose l'arrondi correct pour la plupart de ses opérations. On peut noter l'exception de l'exponentielle et quelques autres fonctions pour lesquelles il faut préciser un paramètre pour obtenir l'arrondi correct. La représentation des nombres dans cette bibliothèque dépend d'un contexte contenant les informations de mode d'arrondi, de précision et les drapeaux levés par les exceptions pour chaque opération. Il est préférable de définir un seul contexte par programme, ce qui engendre quelques contraintes d'utilisation. Il est notamment difficile d'effectuer des calculs avec des précisions dynamiques, la précision étant attachée au contexte et non à une variable résultat particulière comme dans MPFR par exemple. L'arithmétique proposée correspond dons plus à une arithmétique en précision arbitraire, c'est-à-dire une précision arbitrairement grande fixe, plutôt qu'à l'arithmétique multi-précision annoncée. Cette bibliothèque propose un large catalogue de fonctions, toutes les opérations arithmétiques de base, les opérations de comparaison, les fonctions exponentielle et logarithme et les conversion vers les types de base. Les seules lacunes de cette bibliothèque seraient l'absence des fonctions trigonométriques et le manque d'interface avec la bibliothèque MPFR pour effectuer des conversion de nombre multi-précision décimaux à leurs équivalents binaires.

Les outils et bibliothèques mathématiques fiables D'autres outils et bibliothèques mathématiques existent produisant des informations supplémentaires avec un certain niveau de confiance sur les algorithmes numériques. Ce ne sont pas des solutions utilisables en l'état dans le contexte de cette thèse, mais elles peuvent fournir une panoplie de techniques utiles pour l'analyse d'erreur et la certification de programmes. Il existe notamment des solutions comme CRlibm [5] [57] qui est une bibliothèque proposant tout un éventail de fonctions mathématiques élémentaires tout en assurant l'arrondi correct.

Une autre solution est Sollya [58, 59], qui est à la fois un outil et une bibliothèque permettant le développement fiable de programmes utilisant l'arithmétique à virgule flottante. Sollya effectue les calculs en binaire, avec une précision arbitrairement grande fixée par l'utilisateur, et informe celui-ci lorsqu'un arrondi ayant un impact sur le résultat a eu lieu. Sollya est particulièrement destiné à la mise en œuvre automatisée de bibliothèques mathématiques à virgule flottante.

L'outil CADNA (*Control of Accuracy and Debugging for Numerical Applications*) [60] estime les erreurs d'arrondi lors de simulations numériques, en utilisant la méthode CESTAC (contrôle et estimation stochastique des arrondis de calcul). Cette méthode consiste à effectuer plusieurs exécution du programme dont on souhaite évaluer la stabilité numérique

5. `https://scm.gforge.inria.fr/anonscm/git/metalibm/crlibm.git/`

en utilisant un mode d'arrondi aléatoire pour chaque opération. Les différentes exécutions permettent d'estimer dans tout résultat calculé quels chiffres sont affectés par des erreurs d'arrondi. CADNA construit alors un intervalle de confiance contenant la solution exacte. L'outil CADNA permet en plus de l'estimation de l'erreur due à la propagation des erreurs d'arrondi lors de l'exécution du code, de détecter les instabilités numériques et d'estimer la précision de tous les calculs intermédiaires. Dans le contexte de certification aéronautique, nous ne pouvons cependant pas tirer crédit de résultats acquis à partir de méthodes non-déterministes.

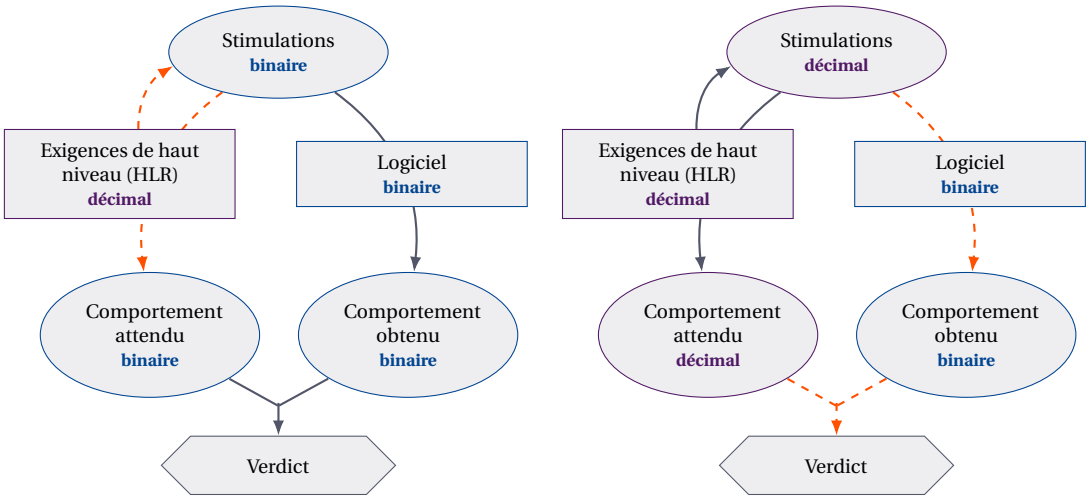
2.3 Certification aéronautique et précision finie

2.3.1 Génération de tests et erreurs de calculs

L'utilisation de la précision finie a un impact important sur l'erreur des algorithmes numériques. On parle de la qualité d'un algorithme numérique en fonction de sa capacité à maîtriser cette erreur. Certaines contraintes relatives à la conception et au développement d'un logiciel dans le contexte de la certification aéronautique peuvent s'appliquer à la précision des calculs numériques. La précision algorithmique correspond à la précision relative attendue pour les sorties de l'algorithme. Les contraintes de robustesse spécifient le comportement du logiciel lorsqu'un ensemble de stimulations ne correspond pas au comportement normal. Ces exigences de robustesse se traduisent explicitement, entre autres, comme des contraintes sur le comportement numérique de l'algorithme s'il était développé en précision infinie.

Le but de la vérification et validation par le test est alors d'assurer que le logiciel embarqué fournit un comportement numérique conforme à celui décrit dans les exigences. La première difficulté dans la maîtrise des erreurs de calculs des algorithmes numériques décrit par les exigences de haut niveau, c'est-à-dire celles qui détaillent le comportement fonctionnel de l'algorithme, réside dans le fait les valeurs numériques utilisées dans ces exigences sont des nombres décimaux tandis que le logiciel effectue ses calculs en binaire en précision finie. De plus le comportement fonctionnel de l'algorithme décrit par les HLR correspond au comportement attendu si les calculs étaient effectués en précision infinie, les considérations relatives à la précision finie relevant du design.

Plusieurs solutions sont possibles lorsqu'il s'agit de générer les tests logiciels à partir d'exigences textuelles décimales. La figure 2.4 illustre deux possibilités de génération et d'exécution des tests pour la vérification en fonction de la base dans laquelle sont effectués les calculs. Les conversions entre les bases binaires et décimales sont inévitables, et selon les choix de base de représentation des tests, ces conversions se situent à des niveaux différents et impliquent des activités plus ou moins complexes.



(a) Lorsque les tests sont générés en binaire (b) Lorsque les tests sont générés en décimal

FIGURE 2.4 – Propagation des erreurs numériques lors de la génération de tests logiciels

Le défi lorsque les tests sont générés en binaire, c'est-à-dire que les stimulations et le comportement attendu sont produits en binaire, est de fournir un formalisme des exigences permettant de produire des résultats en binaire suffisamment précis pour avoir confiance dans la qualité des tests. Le point difficile étant que l'on ne peut pas savoir a priori pour une exigence numérique quelconque quelle est la précision requise pour que la comparaison entre le comportement attendus binaire et le comportement obtenu par l'exécution des tests sur le logiciel fournisse un verdict fiable.

Lorsque les tests sont générés en décimal, les problématiques de conversion et comparaison entre les formats binaire et décimal se trouvent à des niveaux différents. Le premier niveau est la conversion des stimulations représentées en décimal dans un format binaire compatible avec le logiciel sous test. Le second niveau est la comparaison du comportement attendu décimal, obtenu par l'application des stimulations décimales sur les HLR, et du comportement obtenu binaire par l'exécution des tests sur le logiciel. La faisabilité de la comparaison fiable entre des nombres à virgule flottante binaire et décimaux représentés dans divers formats aux précisions différentes a fait l'objet de précédents travaux [14].

Dans le cadre de la génération automatique de tests pour des logiciels numériques, nous devons choisir entre ces deux modèles de gestion des erreurs de conversion et des erreurs de calculs.

2.3.2 Exemple numérique

Reprenons l'exemple de spécification détaillé dans la section 1.1.4 et simplifions les exigences décrivant le comportement numérique normal dans le but d'illustrer les problé-

matiques numériques pouvant survenir lors de la génération de tests, en suivant les étapes décrites dans la figure 2.4. Cette exigence décrit le calcul de la sortie $r \in \mathbb{R}$, tel que

$$r = 5 \cdot \frac{x_1 - \frac{x_3}{x_4}}{t_1}. \tag{2.15}$$

Une précision relative $|\varepsilon| = 10^{-7}$ est attachée au calcul du résultat exact r, de sorte que le comportement attendu valide est définit par un intervalle $[\underline{r}, \overline{r}]$, tel que

$$[\underline{r}, \overline{r}] = r \cdot (1 \pm \varepsilon). \tag{2.16}$$

Un test consiste à comparer un comportement obtenu par l'exécution des stimulations sur le logiciel sous test à cet intervalle de comportement attendu valide. L'oracle de test peut ainsi rendre le verdict `OK` ou `KO` si le comportement obtenu est estimé conforme ou non aux exigences, c'est-à-dire si le résultat obtenu par le logiciel ce trouve dans l'intervalle $[\underline{r}, \overline{r}]$.

Pour effectuer la comparaison entre le comportement obtenu par le logiciel, représenté dans un format à virgule flottante binaire, et l'intervalle $[\underline{r}, \overline{r}]$, il faut convertir cet intervalle dans un format à virgule flottante compatible pour la comparaison (binaire ou décimal puisque qu'on dispose de la comparaison binaire/décimale exacte [14]). Lorsque r est déterminé de façon exacte, on peut représenter l'intervalle du comportement attendu valide par $[\triangle(\underline{r}), \triangledown(\overline{r})]$ comme illustré dans la figure 2.5.

Dans cette figure, nous illustrons la détermination du verdict sur l'exemple numérique suivant :

$$\begin{aligned} r &= 5 \cdot \frac{49 \cdot 10^{-5} - \frac{1 \cdot 10^{-4}}{2 \cdot 10^{-1}}}{5 \cdot 10^{-7}} = -100, \\ [\underline{r}, \overline{r}] &= [-100.00001, -99.99999]. \end{aligned} \tag{2.17}$$

Appelons r_2 le résultat binaire obtenu après stimulation du logiciel. Le verdict du test est `OK` si on peut déterminer de façon certifiée que $r_2 \in [-100.00001, -99.99999]$. Si on assure que $r_2 \notin [-100.00001, -99.99999]$ alors le verdict est `KO`.

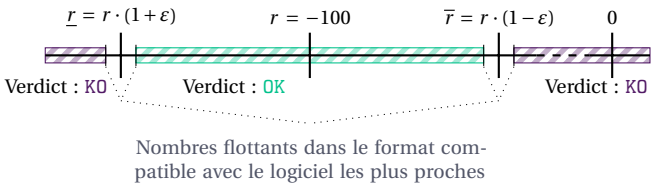


FIGURE 2.5 – Détermination du verdict pour un résultat de tests numériques lorsque r est exact

La génération automatique de tests consiste à déterminer d'un côté les stimulations et de l'autre les comportements attendus. Dans cet exemple, nous supposons que nous avons

déterminé les stimulations, telles que décrites dans l'équation (2.17), et nous cherchons à déterminer automatiquement une approximation des bornes de l'intervalle définissant le comportement attendu $[\underline{r}, \overline{r}]$, notées $[\underline{s}, \overline{s}]$ telles que

$$\underline{s} = \underline{r} \cdot (1 + \varepsilon_{\underline{s}}), \qquad \overline{s} = \overline{r} \cdot (1 + \varepsilon_{\overline{s}}). \tag{2.18}$$

Les calculs des bornes $\underline{r}$ et $\overline{r}$ sont donc soumis à des erreurs $\varepsilon_{\underline{s}}$, resp. $\varepsilon_{\overline{s}}$. Si ces erreurs sont bornées et leurs bornes sont connues, on peut construire un intervalle de confiance autour de $\underline{r}$ et $\overline{r}$, de sorte que $\underline{r} \in \left[\triangle\left(\underline{s} \cdot \frac{1}{1\pm\varepsilon_{\underline{s}}}\right), \triangledown\left(\underline{s} \cdot \frac{1}{1\pm\varepsilon_{\underline{s}}}\right)\right]$. Si cet intervalle n'est pas vide, c'est-à-dire qu'il existe au moins un nombre à virgule flottante dans le format choisi pour représenté le comportement attendu pour rendre le verdict, alors on ne sait pas dire avec certitude si le comportement décrit est valide ou non. Le verdict se décompose ainsi en trois résultats, tels qu'illustrés dans la figure 2.6 : les classiques `OK` et `KO`, ainsi qu'un troisième verdict `INCERTAIN`.

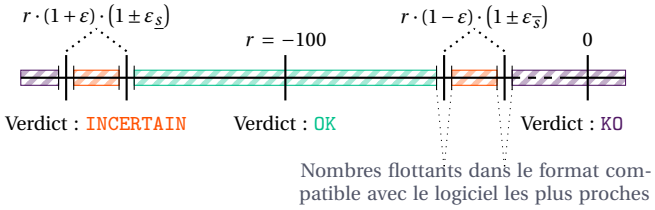


FIGURE 2.6 – Détermination du verdict pour un résultat de tests numériques lorsque $r \cdot (1+\varepsilon)$ est approché

Pour illustrer l'impact des erreurs relatives $\varepsilon_{\underline{s}}$ et $\varepsilon_{\overline{s}}$ sur la détermination du verdict, nous choisissons de calculer le comportement attendu à partir des stimulations de l'équation (2.17) à l'aide de quatre modèles arithmétiques différents :

- l'arithmétique à virgule flottante binaire aux variables représentées en `binary64`,
- l'arithmétique à virgule flottante décimale aux variables représentées en `decimal64`,
- l'arithmétique multi-précision binaire avec MPFR,
- l'arithmétique d'intervalles multi-précision binaire avec MPFI.

Nous supposons que le logiciel sous test calcule le comportement obtenu en `binary64`, et nous souhaitons représenter le résultat attendu dans ce même format pour la comparaison de l'oracle menant au verdict.

On calcule donc dans les quatre formats choisis

$$\underline{s} = \triangle\left(5 \cdot \frac{49 \cdot 10^{-5} - \frac{1 \cdot 10^{-4}}{2 \cdot 10^{-1}}}{5 \cdot 10^{-7}} \cdot \left(1 + 10^{-7}\right)\right) \quad \text{et} \quad \overline{s} = \triangledown\left(5 \cdot \frac{49 \cdot 10^{-5} - \frac{1 \cdot 10^{-4}}{2 \cdot 10^{-1}}}{5 \cdot 10^{-7}} \cdot \left(1 - 10^{-7}\right)\right). \tag{2.19}$$

Les résultat $\underline{s}$ et $\overline{s}$ diffèrent de $\underline{r}$, resp $\overline{r}$ de seulement quelques nombres à virgule flottante en `binary64`, nous représentons donc dans la table les erreurs relatives qui leur sont associées pour plus de lisibilité. Ces erreurs sont calculées par rapport au résultat attendu exact représenté dans le format de comparaison choisi, soit $\left[\triangle\left(\underline{r}\right), \triangledown\left(\overline{r}\right)\right]$ représenté en `binary64` déterminé à partir de l'équation (2.17) tel que

$$\left[\triangle\left(\underline{r}\right), \triangledown\left(\overline{r}\right)\right] = \left[-1\cdot 2^6\cdot 1.56250015625, -1\cdot 2^6\cdot 1.56249984375\right] \tag{2.20}$$

	$\lvert\varepsilon_{\underline{s}}\rvert$	$\lvert\varepsilon_{\overline{s}}\rvert$
`binary64`	2.56e-15	8.1e-15
`decimal64`	0	0
MPFR $p = 100$	1.4e-16	1.4e-16
MPFI $p = 100$	1.4e-16	1.4e-16

Tableau 2.2 – Erreurs relatives lors d'une simulation de génération de tests selon plusieurs modèles arithmétiques

Les résultats présentés dans cette table pourraient laisser penser que l'arithmétique à virgule flottante décimale est plus précise que toutes les variantes de l'arithmétique à virgule flottante binaire. Cependant selon les exigences traitées et les stimulations choisies, l'arithmétique binaire peut tout aussi bien être plus précise. De même, utiliser l'arithmétique multi-précision semble être une solution suffisante pour fournir une plus grande précision sur le résultat attendu, toutefois il se peut que ces derniers bits de précision soient difficiles à obtenir[6], impactant les performances des algorithmes de génération automatique de tests.

Dans le cas de la détermination du résultat avec l'arithmétique d'intervalles, l'erreur relative est déterminée par le rapport entre la largeur de l'intervalle obtenu pour $\underline{s}$, resp. $\overline{s}$, et le résultat exact attendu. C'est la seule solution avec laquelle on peut avoir une information sur l'erreur absolue effectuée lors de ce calcul sans avoir d'informations sur le résultat exact attendu. Dans le cas présent, les arrondis successifs impliquent qu'on se trouve dans le cas de la figure 2.6, avec un verdict ternaire `OK`, `KO`, ou `INCERTAIN`.

Écrire un outil de génération automatique de tests numériques n'est pas une tâche simple à cause de plusieurs facteurs, l'un d'eux étant l'utilisation de l'arithmétique en précision finie. Dans cette thèse, nous voulons réduire au maximum cet intervalle d'incertitude, permettant de rendre un verdict certifié le plus précis possible. Pour cela, nous souhaitons explorer deux pistes : d'un côté l'arithmétique d'intervalles multi-précision décimale, qui n'a jamais été implémentée à notre connaissance, et de l'autre côté la combinaison des arithmétiques à virgule flottante binaire et décimale.

6. C'est une instance de TMD

2.4 Conclusion

Nous sommes confrontés dans cette thèse à deux contextes arithmétiques : d'un côté les exigences, dans lesquelles les valeurs manipulées sont des valeurs décimales et où les contraintes de précision sont traitées à haut niveau, et de l'autre le logiciel dans lequel les calculs numériques sont effectués grâce à une arithmétique binaire en précision finie. Dans ce chapitre, nous avons présenté un aperçu des problématiques liées à l'arithmétique des ordinateurs. L'arithmétique à virgule flottante propose une représentation versatile, présentant des formats aux précisions variées aussi bien binaires que décimaux.

Cependant l'analyse de la propagation des diverses erreurs de calculs n'est pas une chose aisée lorsque les algorithmes étudiés sont des systèmes complexes comme on peut en trouver en aéronautique, rendant l'activité de validation et vérification fastidieuse. Les solutions comme l'arithmétique d'intervalles ou l'arithmétique multi-précision permettent d'augmenter notre confiance dans les résultats de l'activité de génération de tests. Néanmoins les potentielles erreurs de calculs pouvant survenir lors de l'étape de génération de tests numériques rendent la qualification de cette activité difficile. Une étude formelle de l'arithmétique utilisée dans cette partie du processus de validation et vérification est donc essentielle dans le but de fournir un outil de génération automatique candidat à la qualification.

CHAPITRE 3

GÉNÉRATION AUTOMATIQUE DE TESTS

Le test est une activité critique dans le cycle de développement d'un logiciel soumis à certification. Le test exhaustif de systèmes complexes n'étant pas envisageable, la conception de tests significatifs, pertinents et de qualité est une activité au coût de développement élevé [61]. Par conséquent, l'étude de la génération de tests et l'automatisation de cette activité sont des domaines de recherche très actifs. Le processus de certification d'un logiciel aéronautique apporte de fortes contraintes de sûreté et de sécurité [19] sur le cycle de développement du logiciel, ainsi les méthodes de génération de tests privilégiées sont les méthodes formelles et déterministes.

Dans ce chapitre, nous étudierons les contraintes liées à la génération de tests dans le contexte de certification aéronautique, en particulier pour les systèmes numériques, ainsi que leur compatibilité avec les méthodes de génération de tests actuellement disponibles. Nous articulerons notre analyse autour des aspects fondamentaux de la génération automatique de tests :

- le modèle d'exigences fonctionnelles,
- les objectifs de test formalisés et les critères d'évaluation de la qualité des tests,
- la détermination d'une séquence de stimulations permettant d'atteindre ces objectifs de test.

Nous analyserons notamment différentes méthodes permettant la génération automatique de tests, en présentant leurs caractéristiques et limitations. Nous verrons notamment lesquelles pourraient être des candidates potentielles pour la génération automatique de tests pour les logiciels numériques dans le contexte de la certification aéronautique.

3.1 Modèle d'exigences fonctionnelles

La génération automatique des tests dans le contexte de la certification aéronautique n'est pas effectuée à partir du logiciel mais à partir d'un ensemble d'exigences textuelles

décrivant le comportement fonctionnel du système. La traduction de ces exigences textuelles dans un certain formalisme comme entrée compatible à une méthode de génération automatique de tests n'est pas l'objet de cette thèse. Le choix du formalisme le plus adéquat pour représenter les exigences particulières qui nous intéresse est un point déterminant pour la qualité des tests générés à partir de celles-ci.

3.1.1 Description formelle du comportement fonctionnel du système

Les systèmes embarqués soumis à la certification présentent un large éventail de fonctionnalités. Nous avons présenté dans les sections 1.1.4 et 2.3.2 des exemples d'exigences simples permettant d'illustrer les problématiques liées à la génération de tests numériques. Ce qui forme la complexité d'un système embarqué est en fait la combinaison d'exigences aux caractéristiques différentes. Nous distinguons donc différents attributs d'une exigence :

- le type des variables : booléennes / numériques,
- la logique utilisée : logique combinatoire (linéaire ou non-linéaire) [62] / logique séquentielle [63],
- l'accessibilité : boîte blanche / boîte noire,
- le comportement décrit : comportement normal / robustesse (section 1.1.3).

Une exigence peut posséder une ou plusieurs de ces caractéristiques. Les exigences possédant des caractéristiques de *logique combinatoire* n'ont pas d'état interne, contrairement aux exigences de *logique séquentielle* qui décrivent différents états du système pour chaque cycle de temps (ex : signal d'horloge). Si les variables interne et la structure des exigences sont accessibles, on parle de *test en boîte blanche*, sinon il s'agit de *test en boîte noire*. Ces exigences sont aussi appelées exigences temporelles. Dans la section 1.1.4, nous décrivions donc une exigence numérique, de logique combinatoire non-linéaire, partiellement en boîte noire à cause de la présence d'une valeur déterminée par une table, décrivant les comportements normaux et de robustesse. Un autre exemple réaliste d'exigence pourrait être le suivant :

"La sortie augmente proportionnellement au gradient d'évolution. Pour ce faire, soit les entrées x et A, la sortie y doit valoir $y_0 = C_0$ au cycle $n = 0$, puis doit valoir $y_n = \left| C_1 C_2 x + y_{n-1} - \frac{C_2 y_{n-1}}{C_3} \right|$ au cycle n. Si un évènement A se produit (c'est-à-dire A passe à `true`*), alors la valeur y_n reste figée à la valeur y_{n-1}".*

Dans cette exigence, la valeur de la sortie dépend de l'état précédent du système et le changement d'état du système est commandé par un signal horloge : c'est une exigence temporelle synchrone. De plus l'exécution d'une séquence de stimulations pour atteindre un point du système dépend des entrées x et A, mais également de la composante temporelle n.

Remarque 5. *Notre but est d'identifier un ou plusieurs formalismes permettant de représenter le comportement fonctionnel décrit par ces exigences. Nous insistons sur le fait que ces exigences ne doivent pas décrire le design d'une solution particulière. Nous nous concentrerons sur notre objectif principal qui est la génération de tests pour les exigences numériques, tout en prenant en compte les autres caractéristiques associées à ces exigences. Il semble notamment raisonnable de se fixer comme premier objectif la formalisation des exigences numériques combinatoires avant de pouvoir considérer les exigences séquentielles.*

La formalisation d'exigences fonctionnelles peut s'effectuer à l'aide d'une syntaxe pouvant être textuelle ou graphique [64]. Les éléments descriptifs conformes à cette syntaxe suivent des règles sémantiques, c'est-à-dire que chacun de ces éléments est défini de façon précise et non ambiguë. Ainsi le bon formalisme de spécification fonctionnelle peut aider à l'écriture d'exigences non-ambiguës et cohérentes, permettant la détection automatique d'un certain nombre de contradictions, activités requises par la DO–178C [1]. Malgré cela, il peut être difficile de signaler automatiquement les exigences aux descriptions fonctionnelles équivalentes. Par exemple, considérons l'exigence formalisée sous la forme de l'équation suivante :

$$\text{Entrée} : x \in \left[\underline{x}, \overline{x}\right], \quad \text{sortie} : y, \qquad y \;=\; \sqrt{x}. \tag{3.1}$$

Cette exigence décrit la fonction racine carrée. Elle peut s'écrire de façon équivalente comme l'exigence suivante :

$$\text{Entrée} : x \in \left[\underline{x}, \overline{x}\right], \quad \text{sortie} : y, \qquad y^2 \;=\; x, \quad y \geq 0. \tag{3.2}$$

Ainsi un comportement fonctionnel unique peut être décrit de plusieurs façons équivalentes dans un certain formalisme. Un concepteur de test humain pourrait facilement considérer les deux exigences comme équivalentes et créer la même campagne de test pour ces deux exigences, afin d'éviter les tests redondants. En revanche, les campagnes de tests crées par un générateur automatique de tests pourraient être différentes pour ces deux exigences. Avoir des exigences équivalentes et des tests redondants fausse les métriques utilisées pour déterminer de la qualité d'une campagne de tests par rapport aux fonctionnalités décrites dans les exigences (présentées section 1.2.3). Ainsi l'estimation de la qualité d'un générateur de test est intrinsèquement liée au formalisme choisi pour décrire la spécification fonctionnelle [27]. Toutefois, il est suffisant de développer un générateur automatique de test ne décelant pas l'équivalence de ces exigences mais permettant de générer une campagne de test évaluée comme suffisamment qualitative pour permettre au logiciel de passer les activités de certification.

Un des critères motivant le choix d'une description formelle des exigences est sa capacité à capturer l'information fonctionnelle de façon concise et adaptée pour la méthode

de génération de tests utilisée. Il peut cependant être difficile d'écrire tous les types d'exigences dans un formalisme unique, car les caractéristiques selon lesquelles nous décrivons les exigences fonctionnelles font appel à des concepts distincts. En effet, une spécification décrivant un circuit électronique peut être décrit à l'aide d'un langage de logique séquentielle booléenne. Un algorithme de génération de tests peut être développé spécifiquement pour répondre aux problématiques inhérentes à ce formalisme. Cependant il ne sera pas forcément possible de le réutiliser ce formalisme pour décrire une spécification numérique combinatoire de façon adéquate en s'adaptant aux problématiques de génération de tests soulevées dans ce contexte précis. Le choix du formalisme dépend donc du type d'exigences que nous souhaitons représenter, et ainsi du type de tests à générer.

3.1.2 Aperçu des méthodes de formalisation des exigences

Le choix du formalisme dans lequel seront représentées les exigences est un point clef pour la génération automatique de tests. Nous souhaitons trouver une solution simple et outillée, c'est-à-dire que nous ne cherchons pas à implémenter un nouveau langage à partir de zéro, mais plutôt utiliser un outil déjà existant. De plus, la solution choisie doit être adaptée à la génération automatique de tests, notamment pour les tests numériques, en fournissant un moyen d'intégrer un procédé de certification de l'arithmétique, comme l'analyse automatique des erreurs ou l'arithmétique d'intervalles. Nous pouvons caractériser les différents langages de description formelle d'exigences fonctionnelles en plusieurs classes [64].

Langages de description d'états Les langages de modélisation basés sur des états décrivent à la fois les états dans lesquels le système peut se trouver, ainsi que les transitions entre ces états. On peut en distinguer deux types. D'un côté les langages basés sur des machines à états finis, comme par exemple les automates finis [65] ou les diagrammes états-transitions [66], permettent de décrire les états du système par un nombre fini de valeurs, ainsi que les transitions entre ces états. Ces langages sont souvent représentés de manière graphique, dans lesquels les transitions représentent des changements d'états souvent associés à des opérations. Ces langages de modélisations sont fournis dans les outils de Model Based Development comme SCADE [29], à partir duquel le code source est généré automatiquement. Le but de la description du logiciel par une spécification fonctionnelle indépendante est de multiplier les sources de vérification. Une description des exigences par un tel formalisme se rapproche donc plutôt de la description du design d'une solution (son architecture et ses LLR, donc les caractéristiques d'implémentation), plutôt que du comportement fonctionnel de haut niveau.

D'un autre côté les langages de description d'état peuvent se baser sur un modèle décrivant des machines à états infinis, comme le langage Z [67] ou B [68]. Dans ces lan-

gages, les éléments décrivant les états peuvent être représentés comme des ensembles, des relations, des séquences d'évènements ou des fonctions. Les opérations sont décrites par des prédicats donnés en termes de pré- et post-conditions.

Langages algébriques Les langages de spécification algébriques permettent de définir les types et structures de données de façon abstraite, de façon similaire à la définition des structures mathématiques de l'algèbre [69], ainsi que les opérations mathématiques sur ces sur ces types de données. Les données ou types de données représentent les fonctionnalités que l'on cherche à implémenter avec un logiciel par un ensemble d'équations ou d'axiomes. Un exemple de langage de spécification algébrique est le *Common Algebraic Specification Language* (CASL) [70], implémenté dans l'outil FERUS [71] qui permet le développement, le prototypage et l'exécution de spécifications décrites en CASL.

Langages déclaratifs Le concept de programmation déclarative réunit toutes les méthodes de description du comportement fonctionnel d'un système plutôt que de son design, c'est-à-dire sa réalisation selon une structure de contrôle (comme les états du système et les transitions entre ces états) [72]. Les exigences sont décrites comme des relations entre les variables en termes de fonctionnalités ou de règles d'inférence, et l'exécuteur du langage (compilateur ou interpréteur) applique sur ces relations un algorithme fixe permettant de renvoyer un résultat. Décrire un ensemble d'exigences HLR dans ce paradigme se distingue d'une solution logicielle dans le fait que le résultat n'est pas produit dans les conditions de la cible. Des exemples de langages déclaratifs sont les langages logiques comme le Prolog [73], les langages fonctionnels comme le Haskell [74] ou le Wolfram Language [75], les contraintes [76], ou encore les langages de flux de données comme le Lustre [77].

Pour écrire une description des fonctionnalités du système complète, précise et performante, le langage de spécification choisi doit pouvoir tirer parti des caractéristiques de la méthode choisie. Nous nous intéressons entre autres à la programmation par contraintes, qui permet notamment de manipuler des données numériques et qui offre une représentation adéquate des exigences combinatoires, ainsi qu'aux flux de données, qui permettent de maîtriser l'évolution du système au cours du temps. Certaines approches s'intéressent à la combinaison de ces méthodes pour tirer parti de leurs différentes caractéristiques [78].

3.2 Méthodologie de la génération de tests

Lors de la génération de tests selon une certaine stratégie pour la validation et la vérification d'un système [1], il faut se poser deux questions fondamentales : "Qu'est-ce qu'on veut

1. Ce qui se distingue du test exhaustif.

tester ?" et "Quand est-ce qu'on s'arrête de tester ?". C'est en répondant à ces questions que nous pourrons définir les critères selon lesquels on pourra attester de l'absence d'erreurs dans le logiciel par la génération de tests pertinents. Nous pourrons ainsi valider le comportement fonctionnel du système selon un certain niveau de confiance par l'évaluation de la qualité de la campagne de test. Dans cette section, nous détaillerons les activités de test présentées dans le chapitre 1, puis nous présenterons quelques méthodes de résolution de problèmes d'optimisation combinatoires, outils essentiels pour la génération automatique de tests.

3.2.1 Déterminer les points à vérifier à partir d'objectifs de test

Le test exhaustif sur toutes les entrées possible du système étant impossible, une solution raisonnable est d'effectuer une campagne de test contenant un ensemble de tests réduit mais significatif permettant de trouver le nombre maximum d'erreurs ou de comportements non-conformes aux exigences dans le logiciel [79]. Pour générer des tests significatifs, il faut définir clairement quelles erreurs on souhaite détecter dans le logiciel, c'est-à-dire définir des *objectifs de tests*. Des objectifs de tests classiques pour la certification de logiciels embarqués sont par exemple :

- chaque exigence doit être testée au moins une fois,
- chaque opération doit être testée pour des valeurs de fonctionnement normal, resp. anormal, au moins une fois,
- la campagne de test doit atteindre la couverture des exigences par les critères de couverture MCDC, etc.

Les objectifs de tests permettent de déterminer les points précis que nous voulons atteindre dans le système pour vérifier la présence ou l'absence d'erreurs, auxquels nous associons un *comportement attendu*, c'est l'oracle de test. Ces objectifs peuvent être plus ou moins précis selon le niveau de description des fonctionnalités et les contraintes relatives à la certification du logiciel. La validation du système est donc limitée à un nombre fini d'objectifs de tests motivés par la recherche d'erreurs potentielles présentes dans le logiciel. Cependant, cette méthodologie permet d'associer à un objectif de test défini de façon précise et non-ambiguë des liens de traçabilité forts entre un test généré et l'exigence qui en est la source.

3.2.2 Déterminer les stimulations pour atteindre des points du système

La génération de tests se découpe donc en deux parties ; nous avons d'abord étudié la détermination des points du système à valider par le test. Nous nous intéressons mainte-

nant au problème de la génération de stimulations numériques fiables à partir des comportement attendus identifiés précédemment. Déterminer une séquence de stimulations permettant d'atteindre un point spécifique du système revient à un problème de recherche de chemin dans un espace de configuration [80, 81]. C'est un problème difficile dû à la taille de cet espace, et à l'hétérogénéité des fonctionnalités décrites dans les exigences (booléennes/numériques, combinatoires/séquentielles, linéaires/non-linéaires) [82, 62].

Combiner une analyse de la propagation des erreurs de calcul pour pouvoir déterminer des stimulations numériques de manière fiable avec une analyse structurelle des états et transitions du système permettant d'atteindre le comportement attendu à observer dans le logiciel est un problème difficile [81], aussi bien pour des tests générés de façon automatique que manuelle. Le document de recommandations pour la certification de logiciels embarqués DO178–C [1] prévoit donc de remplacer la génération de certains tests par des activités de relecture et d'analyse (voir figure 1.3).

On peut donc envisager de développer un outil de génération automatique de tests exécutant plusieurs actions. À partir d'une spécification fonctionnelle formalisée traduire (à la main) des exigences textuelles et d'objectifs de tests décrits dans ce même formalisme, l'outil détermine les points du système correspondant, c'est-à-dire les comportements attendus. La deuxième étape est de générer les stimulations correspondantes permettant d'atteindre la configuration souhaitée dans le système pour vérifier les comportements attendus. L'outil peut s'arrêter avant d'avoir atteint le critère d'adéquation si l'une des deux étapes de génération de test est trop difficile à calculer, du moment qu'il informe l'utilisateur précisément sur les tests qui n'ont pas été générés.

3.2.3 Vérifier la qualité des tests

Cette dernière notion de fiabilité d'une campagne de test peut être quantifiée grâce au *critère d'adéquation* d'un ensemble de tests à une spécification fonctionnelle. Si on considère un ensemble d'exigences fonctionnelles E, on peut alors définir l'ensemble des programmes ψ non-équivalents implémentés à partir de E, ce qui signifie que ces programmes sont développés différemment et ne présentent potentiellement pas les mêmes fonctionnalités. Un ensemble de tests T est dit *adéquat* [83] pour le système d'exigences E relativement à ψ si pour chaque programme $P \in \psi$, P ne passe pas la campagne de test pour au moins un point de test, alors T a découvert une erreur dans P par rapport à E [83]. Si l'ensemble de programmes ψ implémentés à partir de E est fini, alors E possède un ensemble de tests adéquat.

Définition 5 (Critère d'adéquation des tests [79, 84])**.** *Un critère d'adéquation des tests est un prédicat permettant de déterminer si l'activité de test peut s'arrêter.*

En général, le critère d'adéquation des tests est un ensemble de règles dérivées des objectifs de test ; il est alors satisfait lorsque tous les objectifs de tests sont remplis par au moins un cas de test [2]. Dans la section 1.2.1 nous avons définis si des critères recommandés d'évaluation de la qualité d'une campagne de tests pour la certification. L'ensemble de ces critères, correspondent aux objectifs de tests, forment donc le critère d'adéquation des tests. Ainsi le but de la génération d'une campagne de test est de remplir le critère d'adéquation associé avec un nombre nécessaire et suffisant de tests, soit de remplir tous les objectifs de tests déterminés pour valider les fonctionnalités des exigences avec un minimum de tests. La génération de tests peut être apparentée à un problème d'optimisation [81].

3.2.4 Résoudre des problèmes d'optimisation combinatoire

Programmation par contraintes La programmation par contraintes [85] est un paradigme de programmation dans lequel les relations entre les inconnues d'un problème, c'est-à-dire les variables, sont décrites sous la forme de contraintes, c'est-à-dire de relations entre les variables limitant leurs domaines de définition. Le problème est de trouver des valeurs pour les variables satisfaisant la contrainte, ou toutes les contraintes du modèle. Un ensemble de valeurs associées aux variables sont dites *cohérentes* si elles forment une solution des contraintes. La recherche de solutions est effectuée par des solveurs de contraintes, dont la tâche consiste à éliminer les valeurs inconsistantes. La résolution de contraintes pour la génération de tests numériques est utilisée dans plusieurs outils de génération de tests pour la vérification et validation de logiciels critiques. On compte notamment l'outil GaTel [86], développé par la Commissariat à l'énergie atomique (CEA) se basant sur une représentation en Lustre d'une spécification du logiciel, et manipulant des variables booléennes et entières. On remarque également le cadre applicatif FPCS [87] qui propose de générer des tests pour des intervalles à partir d'un programme annoté. Les tests visent à vérifier les intervalles où le comportement du programme pourrait ne pas être conforme à un modèle manipulant des nombres réels.

Satisfaisabilité modulo théories (SMT) Un problème SAT (pour satisfaisabilité), s'exprimant grâce à une formule de logique propositionnelle, s'attache à déterminer s'il existe une configuration des variables propositionnelles satisfaisant ce problème, c'est-à-dire rendant la formule vraie [88]. Un problème SMT est une généralisation du problème SAT à la logique du premier ordre, étendue avec des symboles de fonction et de prédicat particuliers, formant des théories. Un problème SMT peut donc décrire, par exemple, la théorie de l'arithmétique linéaire sur les entiers (LIA) ou la théorie des tableaux (Array). La technique SMT repose sur des procédures de décision, soit des petits algorithmes de preuve

2. Un test peut remplir plusieurs objectifs de tests.

pour décider de la satisfaisabilité de formules pour les théories élémentaires données en exemple ci-dessus [89]. Combiné à la programmation par contrainte, un problème SMT peut également permettre une représentation formelle précise des nombres à virgule flottante [90]. Il existe tout un éventail de solveurs SMT permettant de montrer l'existence d'une configuration satisfaisant le problème, certains dédiés à des problèmes spécifiques et d'autres plus généralistes. Pour optimiser l'utilisation d'un solveur SMT, il est important de bien analyser le problème que l'on souhaite résoudre ainsi que la structure de données adéquate. Les problèmes SMT peuvent être utilisés dans le cadre de la preuve de circuits électroniques, ou pour les problèmes de coloration de graphes [91]. La complexité de ce genre de problème croît de façon exponentielle avec l'augmentation de la taille des données. La principale difficulté liée à la résolution de problèmes de décision à l'aide de solveurs SMT est la scalabilité [88].

3.3 Quelques méthodes de génération de tests

D'après les résultats de l'étude des problématiques soulevées par la génération de tests à partir d'exigences numériques formalisées dans un contexte de certification de logiciels critiques, nous nous intéressons à différentes méthodes associées aux activités de génération de tests et de qualification de la qualité de ces tests. Ces méthodes n'ont, à notre connaissance, pas encore d'application directe dans le contexte spécifique de cette thèse, mais certains principes associés à ces approches pourraient être réutilisés pour la création d'une campagne de tests numériques qualifiée à partir d'exigences formelles. Nous cherchons donc à étudier ces approches en les regroupant selon la stratégie de génération employée. Certaines approches utilisent la structure du logiciel comme base pour la génération de tests basée sur la couverture du logiciel tandis que d'autres méthodes s'attachent à générer des tests par injection de fautes, c'est-à-dire à partir d'erreurs potentielles présentes dans le logiciel.

Remarque sur le test aléatoire Le test aléatoire [92] est une méthode de génération de tests efficace ne demandant pas beaucoup de ressources et permettant de tester à moindre coût un large éventail de comportements décrits par les exigences en règle générale. Cependant, dans le contexte de la certification aéronautique, le test aléatoire n'est pas envisageable. En effet, une campagne de test générée de cette façon ne permet pas d'établir la pertinence des tests selon un objectif de tests, ou la traçabilité entre les tests et les exigences fonctionnelles, c'est-à-dire que les tests ne sont pas reliés à un objectif de test qui lui même n'est pas motivé par le besoin de tester un point particulier d'une exigence. De plus, même si on peut estimer un pourcentage des fonctionnalités qui n'ont pas été vérifiées par une campagne de tests générée aléatoirement, on ne peut pas estimer quel effort de test il

faudrait encore fournir par cette méthode pour valider les objectifs de test manquants, ce qui n'est pas acceptable pour la certification d'un logiciel aéronautique [1].

3.3.1 Générer des tests à partir de critères de couverture

Interprétation abstraite L'interprétation abstraite consiste en la représentation du système considéré, logiciel ou modèle, par des approximations de sa sémantique [93], notamment son flux de données. Ces abstractions sont définies par une sémantique abstraite construite sur des domaines abstraits déterminés selon un compromis entre la précision de la description et la rapidité de l'analyse [94]. Un domaine abstrait est une approximation d'un état concret du système considéré, ne prenant en compte que certaines propriétés spécifiques de cet état. Les abstractions numériques en particulier sont représentées par un ensemble infini de vecteurs numériques de dimension finie, comme par exemple les inégalités linéaires sur les intervalles [95] ou les zonotopes [96]. L'interprétation abstraite combinée à la programmation par contraintes permet l'élaboration d'algorithmes d'exploration de chemins dans un graphe de flux de données [94]. Lors de cette exploration, des approximations des domaines sont calculées aux embranchements pour chaque chemin du graphe de flux de données. Ces domaines peuvent correspondre aux classes d'équivalences (voir section 1.2.3) pour lesquelles le système sous test a un comportement uniforme. Une méthode de génération de test serait donc d'atteindre chacune de ces classes d'équivalence avec une ou plusieurs séquences de stimulations. Des outils comme Fluctuat [97] et Frama-C [98] permettent la vérification de programmes C par une analyse statique utilisant l'interprétation abstraite. Le problème de ces outils est qu'ils effectuent une analyse du code source pour fournir une preuve de présence d'erreurs. Les méthodes suivantes sont des exemples particuliers d'interprétation abstraite.

Exécution symbolique Cette méthode consiste à analyser un programme afin de déterminer quelle entrée induit l'exécution de quelle partie du programme de façon à le représenter sous forme de données symboliques. C'est-à-dire que le programme est représenté par des expressions symboliques dépendant de ses entrées. Chaque branche est donc représentée sous forme d'un chemin de contraintes, à savoir une formule booléenne dépendant des entrées symboliques, et de l'accumulation des contraintes précédentes [81, 99]. L'exécution symbolique est une méthode par boîte blanche, car elle nécessite l'analyse du code source et de l'architecture du logiciel pour former les données symboliques correspondantes [80]. Ces données peuvent être représentées notamment sous forme de contraintes en fonction des entrées ; la détermination de tests revient donc à un problème de résolution de contraintes pour trouver des entrées permettant d'atteindre chaque point de test du système [99]. Cependant, dans le cas général, sur des programmes de taille conséquente, la génération de test par exécution symbolique est soumis à obstacles. Le premier problème

est l'*explosion combinatoire* du nombre de chemins en fonction de la taille du programme. Pour permettre une terminaison de la recherche de test par exécution symbolique, on doit ainsi se contenter de l'exécution d'un sous-ensemble de chemins. Le second problème est la *divergence des chemins* entre le résultat de l'exécution de contraintes symboliques et le résultat attendu par le programme. Ceci survient lorsque la représentation symbolique du programme n'est pas suffisamment fidèle. Il est possible de n'utiliser que partiellement l'exécution symbolique et ainsi pallier à ces problèmes. Cette méthode s'appelle l'exécution symbolique dynamique. Il s'agit alors de remplacer les valeurs symboliques aux chemins trop complexes par des valeurs concrètes issues de l'exécution du programme.

Search-Based Software Testing Le Search Based Software Testing (SBST) [80, 100], est une approche qui reformule le problème de détermination d'une campagne de test en problème d'optimisation. C'est une méthode de test par boîte noire, qui va appliquer un algorithme de recherche sur le domaine de définition des entrées. Le SBST utilise des méthodes de recherche méta-heuristiques, permettant d'atteindre les objectifs de test à l'aide d'une *fonction objectif* pondérant l'espace de configuration en fonction de l'adéquation des fonctionnalités avec l'objectif de test. Ainsi, trouver les entrées, ou la combinaison d'entrées permettant de maximiser la pertinence de la recherche selon la fonction objectif revient à un problème d'optimisation. Cette méthode a pour avantage de considérer le domaine de définition des entrées dans son ensemble, quel que soit leur type. Elle est donc bien adaptée pour traiter les problèmes numériques prenant en entrée des nombres flottants [80]. Cependant, comme toute méthode utilisant une heuristique, elle peut renvoyer seulement un extremum local. La campagne de test qui en découlera ne permettra pas forcément de détecter toutes les erreurs recherchées.

3.3.2 Générer des tests pour trouver les erreurs du logiciel

Mutation Testing Le test par mutation [3, 8] est une méthode d'évaluation de la qualité d'une campagne de test par injection de fautes. Il consiste à apporter de petite modification, appelée *mutation* au système sous test, ou *mutant*, puis effectuer la campagne de test dont on souhaite évaluer la qualité sur cette version modifiée du système. Si la campagne de test a réussi à détecter l'erreur, et donc tuer le mutant, alors elle est considérée robuste aux erreurs de ce type. Dans le cas contraire, le mutant a survécu. On peut mesurer un critère d'*adéquation à un ensemble de mutations* comme le rapport entre le nombre de mutants tués sur le nombre de mutants générés au total. Ceci permet de mesurer le pourcentage de mutants rejetés pour déterminer l'efficacité de la campagne de test. De plus avec cette méthode on connaît spécifiquement quels mutants ont survécu à la campagne de test, chaque mutant étant lié à une mutation. L'étude de cette mutation, pouvant être en lien avec un objectif de test, peut ainsi donner une indication précise sur les erreurs présentes

dans le logiciel. Cette méthode est souvent décrite pour une application sur un logiciel, mais pourrait être facilement transposable à une spécification formelle. Elle peut être utilisée pour assister la conception de modèles ou de campagne de tests, mais n'a pas été appliquée, à notre connaissance, dans le contexte de la génération de tests à partir d'une spécification formelle. Le test par mutation est souvent couplé à la programmation par contraintes pour évaluer un ensemble de critères permettant d'attester de la qualité d'un logiciel [9, 101].

Model Checking Le Model Checking est un ensemble de méthodes pour la vérification automatique d'un système représenté comme un ensemble d'états fini à partir d'un langage de spécification muni d'une sémantique formelle. Le but de la validation par Model Checking est de mettre en évidence des erreurs dans la conception d'un système complexe qui ne sont pas détectables par les activités de simulation ou de test [24]. Ces erreurs sont détectées grâce à la violation de propriétés, démontrées par la recherche de contre-exemples, c'est-à-dire de séquences de stimulations permettant d'amener le système dans un état violant la propriété choisie. Si aucun contre-exemple n'a été trouvé pour une propriété dans le système fini, alors la propriété est démontrée correcte pour le système. Les méthodes de Model Checking ont déjà été utilisées comme générateur de tests, par la génération de contre- exemples pour certaines propriétés choisies en fonction des objectifs de test. Une fois que le point discriminant permettant d'invalider la propriété choisie, donc de déterminer si une erreur se trouve dans le logiciel, a été trouvé, les stimulations sont déterminées grâce à une méthode de résolution de contraintes temporelles [102]. Cette méthode est notamment efficace dans le cas d'une spécification logique à états finie, dans laquelle tous les états du système peuvent être visités de façon exhaustive. Il peut être plus difficile de trouver des contre-exemples suffisamment précis dans le cas d'exigences numériques.

3.4 Conclusion

Dans ce chapitre, nous avons décrit les différentes activités relatives à la génération de test à partir d'une spécification fonctionnelle. Nous avons donné un aperçu des différentes méthodes existantes permettant de répondre aux problématiques liées aux différents aspects de la génération automatique de test présentés en introduction. Tous ces éléments sont résumés dans la figure 3.1.

La génération de test consiste en la génération d'un oracle de test formé de comportement attendus, ainsi que des stimulations associées. De plus un critère d'évaluation de la qualité de la campagne de test est nécessaire comme critère d'arrêt à la génération de tests. Plusieurs méthodes ont été développées pour la génération de tests reposant sur la structure du logiciel. Leur but est en effet de générer des tests permettant de couvrir toutes

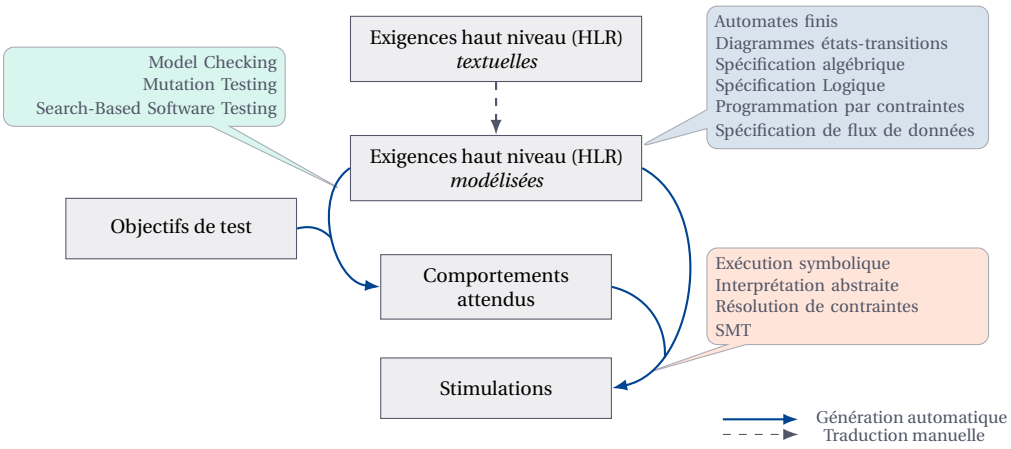


FIGURE 3.1 – Génération de tests basés sur la spécification fonctionnelle

les instructions, conditions ou branches du logiciel. Nous souhaitons nous inspirer de ces méthodes pour développer un générateur de tests à partir d'une spécification fonctionnelle formalisée, en concentrant principalement nos efforts sur les tests numériques pour la certification de logiciels embarqués.

DEUXIÈME PARTIE

ARITHMÉTIQUE CERTIFIÉE

CHAPITRE 4

ARITHMÉTIQUE D'INTERVALLES DÉCIMALE MULTI-PRÉCISION

Le cycle de vérification et validation de logiciels aéronautiques soumis à certification doit respecter des règles strictes. Nous avons vu dans le chapitre 2 quelques exemples de fautes pouvant survenir lors de la conception de tests, notamment des erreurs dues à des problèmes arithmétiques. Les concepteurs d'exigences, sur lesquelles sont basés les tests, ne sont en effet pas des informaticiens sensibilisés à l'arithmétique des ordinateurs à précision finie et les répercussions de son utilisation dans le système en cours de développement. C'est pourquoi nous proposons plusieurs solutions pour intégrer une arithmétique certifiée en binaire et en décimale dans le cycle de validation et de vérification. Ces travaux font suite au manque de solutions adéquates constaté pour répondre aux problématiques spécifiques soulevées dans ce contexte de certification aéronautique.

Dans ce chapitre, nous décrivons les mécanismes et l'implémentation de MPDI, une bibliothèque définissant des opérations arithmétiques d'intervalles décimales multi-précision. Le but d'une telle bibliothèque est de fournir des résultats décimaux fiables et précis. Cette bibliothèque contient les opérations arithmétiques de base rapides et fiables avec arrondi correct (pour une précision arbitraire décimale) ainsi que quelques fonctions élémentaires partageant les mêmes caractéristiques. De plus la représentation décimale définie est compatible avec la norme IEEE 754-2008 [38], et l'arithmétique d'intervalles est conforme à la norme IEEE 1788-2015 *Standard for Interval Arithmetic* [42].

Ces travaux ont mené à la publication d'un article dans le journal *Reliable Computing* en 2017 [12].

4.1 MPDI : conception et architecture

La conception d'une bibliothèque arithmétique d'intervalles décimale multi-précision ne s'effectue pas à partir de zéro. Nous souhaitons optimiser nos travaux en nous basant sur des bibliothèques existantes, éprouvées et performantes.

4.1.1 Choix de GNU MP et MPFR

Des études de faisabilité de l'implémentation de bibliothèques arithmétiques de calculs à virgule flottante décimaux ont montré que la méthode la plus simple et efficace d'implémenter des opérations décimales est d'utiliser leurs équivalents binaires [103]. De cette façon, il est possible de s'appuyer sur la base solide et l'efficacité de l'arithmétique binaire à virgule flottante et concentrer nos efforts d'implémentation sur certains points critiques.

Plusieurs critères sont à prendre en compte pour les choix d'implémentation de cette bibliothèque arithmétique d'intervalles décimale multi-précision. Ils dépendent des deux objectifs que l'on souhaite atteindre avec le développement de cette bibliothèque :

- l'arrondi correct en décimal pour une précision arbitraire, et
- l'efficacité et la rapidité des calculs.

Nous souhaitons implémenter une bibliothèque fiable qui remplisse les propriétés d'arrondi correct en décimal pour une précision arbitraire. Nous voulons également une bibliothèque rapide, qui, quand bien même ne parviendra pas à égaler les performances d'une bibliothèque arithmétique binaire très optimisée écrite en assembleur, puisse fournir des résultats décimaux fiables en un temps raisonnable. Comme nous décidons de nous reposer sur des fonctions binaires pour le calcul des fonctions arithmétiques complexes pour optimiser le coût d'implémentation, nous souhaitons que la bibliothèque binaire choisie soit rapide pour ne pas induire un surcoût trop élevé.

Plusieurs bibliothèques binaires de calculs en multi-précision existent, comme répertoriées dans le chapitre 2. Conformément à nos prérequis, nous choisissons de nous reposer sur GNU MP [49] et MPFR [11], bibliothèques fiables et performantes, fournissant les propriétés d'arrondi correct en multi-précision en binaire.

4.1.2 Représentation d'intervalles décimaux multi-précision

Dans ce chapitre, nous allons travailler avec les ensembles des nombres à virgule flottante binaires et décimaux représentables à une certaine précision binaire k_2 ou une certaine précision décimale k_{10}, présentés dans le chapitre 2. Nous adoptons la notation suivante :

- $\mathbf{x}, \mathbf{y}, \mathbf{z} \in \mathbb{R}$ représentent les nombres réels,
- $x, y, z \in \mathbb{Z}$: les entiers signés,
- $x_2, y_2, z_2 \in \mathbb{F}_{k_2}^2$: les nombres à virgule flottante binaires à la précision k_2,
- $x_{10}, y_{10}, z_{10} \in \mathbb{F}_{k_{10}}^{10}$: les nombres à virgule flottante décimaux à la précision k_{10},

- $\varepsilon, \eta, \theta$: les erreurs.

Au début de ces travaux il n'existait pas, à notre connaissance, de bibliothèque d'arithmétique décimale multi-précision bas niveau. Les bibliothèques disponibles comme BigDecimal [55] [1] étant des bibliothèques de haut niveau, elles ne peuvent pas servir de base pour une bibliothèque d'arithmétique d'intervalles décimale répondant à des besoins de performances. De plus BigDecimal, bien que proposant les arrondis dirigés, ne donne aucune indication sur sa conformité avec l'arrondi correct, ce qui est problématique pour l'utilisation dans le cadre de la certification.

Il existe la bibliothèque *mpdecimal* [56] [2] qui repose uniquement sur GNU MP et MPFR, de la même façon que celle que nous voulons développer et garantissant l'arrondi correct. Un des points clefs de notre développement est de pouvoir fournir des opérations de conversion entre les formats multi-précision binaire et décimal. La bibliothèque *mpdecimal* est limitée dans son interface avec les formats à virgule flottante binaire et ne propose pas d'opérations trigonométriques. De plus la précision courante est attachée à un contexte, et non aux variables. La gestion d'un algorithme multi-précision où chaque variable a une précision courante différente est donc complexe. Nous souhaitons effectuer une comparaison avec cette bibliothèque pour s'assurer de l'exactitude de nos résultats, ainsi que confronter leurs performances temporelles.

Nous souhaitons donc commencer nos travaux par définir une bibliothèque décimale multi-précision qui proposera un nombre intéressant d'opérations arithmétiques sur laquelle reposera notre bibliothèque d'intervalles.

Le format d'un nombre décimal multi-précision est le suivant :

$$x_{10} \coloneqq n \cdot 10^F, \qquad x_{10} \in \mathbb{F}^{10}_{k_{10}}, \tag{4.1}$$

tel que

- $n \in \mathbb{Z}$ la mantisse, un entier signé GMP,
- $F \in \mathbb{Z}$ l'exposant, un entier signé 64 bits,
- une information additionnelle à propos de la classe de x_{10} (NaN, $\pm\infty$, ± 0 ou un nombre décimal)
- $k_{10} \geq 2$ la précision décimale en chiffres décimaux après la virgule, la bibliothèque va garantir que $1 \leq |n| \leq 10^{k_{10}} - 1$ en toutes circonstances.

1. `https://docs.oracle.com/javase/7/docs/api/java/math/BigDecimal.html`
2. `http://www.bytereef.org/mpdecimal/index.html`

Dans notre bibliothèque, nous représentons les intervalles avec deux nombres décimaux multi-précision de la façon suivante :

$$\left[\underline{x}_{10}, \overline{x}_{10}\right] := \left\{\mathbf{x} \in \mathbb{R} \mid \underline{x}_{10} \leq \mathbf{x} \leq \overline{x}_{10} \mid \underline{x}_{10}, \overline{x}_{10} \in \mathbb{F}_{k_{10}}^{10}\right\}. \tag{4.2}$$

La norme IEEE 754-2008 [38] définit un format de nombres à virgule flottante décimaux. Grâce à la normalisation des exposants, un nombre à virgule flottante binaire n'a qu'une représentation unique. Ce n'est pas le cas pour les nombres à virgule flottante décimaux, qui de par leur définition peuvent avoir plusieurs représentations pour une même valeur. Ces différentes représentations sont appelées une cohorte.

Le *quantum* d'un nombre représenté dans le format à virgule flottante décimal en précision finie est la valeur de l'unité en dernière position de sa mantisse. Le quantum est utilisé pour distinguer deux nombres d'une même cohorte, c'est-à-dire deux représentation d'un même nombre à virgule flottante décimal.

Le format décimal que nous définissons est compatible avec le format à virgule flottante décimal définit dans la norme IEEE 754-2008, mais n'introduit pas la notion de quantum et n'est donc pas conforme à cette norme.

4.1.3 Architecture haut niveau

Pour implémenter notre bibliothèque d'arithmétique d'intervalles décimale multi-précision, nous avons choisi de nous reposer sur la bibliothèque GNU MP pour la définition de notre format décimal. En supposant l'arrondi correct des opérations de conversion entre le format binaire et décimal, nous voulons utiliser des fonctions binaires pour l'implémentation de notre bibliothèque. Pour calculer les fonctions binaires en multi-précision nous avons choisi d'utiliser la bibliothèque MPFR.

La bibliothèque Multi-Precision Decimal with Intervals (MPDI), est construite sur la bibliothèque Multi-Precision Decimal (MPD), de la même façon que MPFI se base sur la bibliothèque MPFR, comme présenté dans la figure 4.1. Un intervalle est représenté par ses deux bornes décimales qui sont des nombres MPD. Le calcul de ces bornes est réalisée par l'appel des fonctions décimales MPD avec des arrondis dirigés.

La figure 4.2 illustre les opérations multi-précision décimales supportées par la bibliothèque MPD, ainsi que leurs dépendances aux fonctions GMP et MPFR. Au moins une opération binaire est appelée dans chaque opération décimale. L'exactitude des résultats données en multi-précision décimale repose sur l'exactitude et l'arrondi correct des deux algorithmes de conversion.

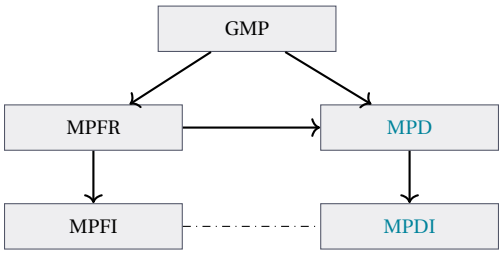


FIGURE 4.1 – Architecture haut niveau, conception des bibliothèques MPD et MPDI

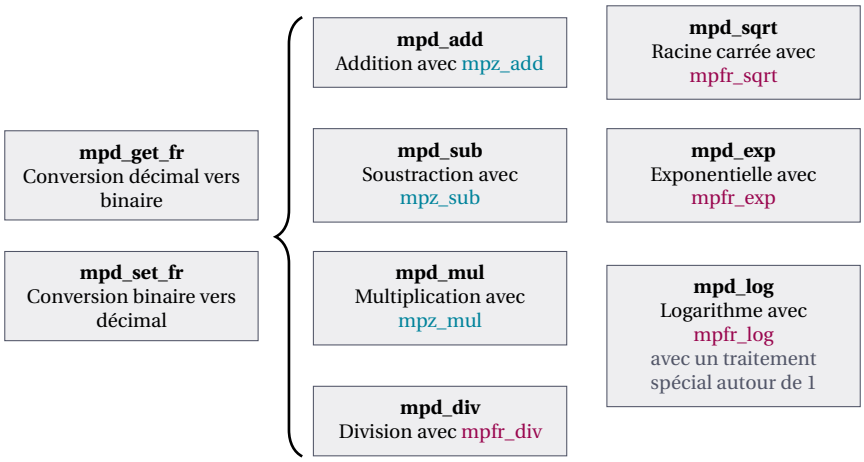


FIGURE 4.2 – Architecture globale de la bibliothèque MPD

4.2 Opérations arithmétiques de base et conversion

Les opérations arithmétiques de base sont l'addition, la soustraction, la multiplication et la division. Dans notre architecture, chacune de ces opérations repose sur la conversion correctement arrondie entre les formats binaire et décimal multi-précision.

4.2.1 Multiplication

Il peut paraître étrange de commencer l'étude de la multiplication au lieu de l'addition, mais il se trouve que la multiplication est l'opération arithmétique la plus simple à implémenter dans notre format décimal. En effet, cet algorithme consiste à calculer z_{10} avec $x_{10}, y_{10} \in \mathbb{F}_{k_{10}}^{10}$ de la façon suivante :

$$\begin{aligned} z_{10} &= \circ_{k_{10}}\left(x_{10} \times y_{10}\right), \\ z_{10} &= \circ_{k_{10}}\left(n_{x_{10}} \cdot 10^{F_{x_{10}}} \times n_{y_{10}} \cdot 10^{F_{y_{10}}}\right). \end{aligned} \tag{4.3}$$

L'algorithme de multiplication se décompose selon les étapes suivantes :

1. Calculer le résultat de l'exposant $F_{z_{10}}$ par l'addition des deux exposants de x_{10} et y_{10}, tel que $F_{z_{10}} = F_{x_{10}} + F_{y_{10}}$, ceci revient donc à calculer l'addition de deux nombres entiers signés de 64 bits,
2. Calculer une mantisse temporaire $n'_{z_{10}}$ exacte par la multiplication des deux mantisses avec la fonction de multiplication de GMP [49], telle que $n'_{z_{10}} = n_{x_{10}} \times n_{y_{10}}$ avec `gmp_mul` $\left(n'_{z_{10}}, n_{x_{10}}, n_{y_{10}}\right)$,
3. Arrondir le résultat intermédiaire $n'_{z_{10}}$ à la précision requise pour le résultat z_{10} et ajuster les exposants en conséquence.

La fonction d'arrondi `mpz_round` consiste à vérifier le mode d'arrondi choisi pour l'opération de multiplication. Dans le cas des arrondis vers $+\infty$, $-\infty$, ou zéro, une simple troncature est suffisante par décalage des mantisses. Dans le cas de l'arrondi au plus proche, une boucle est effectuée à partir du dernier chiffre significatif à arrondir. Ceci permet de déterminer si l'arrondi est évident, c'est-à-dire si le chiffre suivant est différent de 4, ou s'il faut boucler sur les chiffres significatifs précédents afin de désambiguïser l'arrondi. La boucle prend alors en compte au maximum tous les chiffres significatifs disponibles, et au minimum seulement un chiffre significatif.

La troisième étape est possible grâce à notre algorithme de conversion, décrit ci-après, qui sert également à arrondir le résultat correctement lorsque la taille du nombre dépasse la précision attendue en sortie.

4.2.1.1 Addition et soustraction

Les algorithmes d'addition et de soustraction sont plus complexes que l'algorithme de multiplication, car ils nécessitent de manipuler les mantisses pour les aligner de sorte à effectuer l'addition, resp. soustraction. En binaire, cette opération de décalage est peu coûteuse car elle revient à effectuer une manipulation des nombres en mémoire. Pour effectuer le même genre d'alignement en décimal, il faudrait diviser la mantisse du nombre le plus grand en valeur absolue. La division est une opération plus complexe et coûteuse qu'un décalage de bits en binaires. Elle fait appel à l'algorithme de conversion du format décimal vers binaire (voir section 4.2.1.2) et fait appel à la fonction de division `mpfr_div`, respectant le raisonnement décrit dans la section 4.4.2.1.

En revanche, comme décrit précédemment, la multiplication est l'opération la plus simple et peu coûteuse en arithmétique multi-précision décimale. Dans le pire des cas, la multiplication fait appel à une addition de nombres entiers signés de 64 bits, une multiplication à l'aide de `gmp_mul`, et une opération d'arrondi à l'aide de `mpd_round`. Malgré le fait que cette opération est plus coûteuse qu'un simple décalage de bits, qui serait utilisé pour aligner les mantisses en binaire, nous utilisons cette opération pour aligner les mantisses

décimales. On va donc utiliser la multiplication par une puissance de dix pour aligner les entrées au lieu de la division. L'addition, resp. soustraction, peut ensuite être effectuée de façon exacte, c'est-à-dire sans aucune élimination, en appelant la fonction GMP `gmp_add`, resp `gmp_sub`, sur les mantisses entières, et le résultat ensuite arrondi grâce à l'opération `mpd_round` à la précision k_{10} exigée en sortie.

4.2.1.2 Algorithme de conversion

Décrivons maintenant les grandes lignes de l'algorithme de conversion entre les formats binaire et décimal. Nous présentons l'algorithme de conversion du format binaire vers décimal, mais les mêmes principes s'appliquent pour la conversion inverse.

L'algorithme 4.1 correspond à la fonction `mpd_set_fr`, c'est-à-dire la fonction qui prend un nombre à virgule flottante binaire en précision arbitraire MPFR et le convertit en nombre décimal MPD. Nous voulons convertir le nombre binaire $x_2 = m \cdot 2^E, x_2 \in \mathbb{F}_{k_2}^2$ vers le nombre décimal $x_{10} = n \cdot 10^F, x_{10} \in \mathbb{F}_{k_{10}}^{10}$ tel que

$$x_{10} = x_2 \cdot (1+\varepsilon), \qquad |\varepsilon| \leq 10^{-k_{10}}. \tag{4.4}$$

Les précisions k_2 et k_{10} sont représentées sur des entiers non signés de 64 bits, et sont ainsi bornés tels que $2 \leq k_2 \leq 2^{63}-1$ et $2 \leq k_{10} \leq 2^{63}-1$. Les exposants E et F sont représentés par des entiers signés de 64 bits, et sont donc bornés en conséquences.

Le principe de cet algorithme est de calculer l'exposant F, puis de calculer une approximation r_2 de la mantisse n sous la forme d'un nombre à virgule flottante binaire. Nous cherchons, à partir de cette approximation, à trouver l'arrondi correct à une précision donnée k_{10}, c'est-à-dire donner le même dernier chiffre que l'arrondi du résultat exact. Calculer l'arrondi correct peut être difficile, car on ne peut pas savoir a priori combien de chiffres supplémentaires sont nécessaires pour y parvenir. C'est le Dilemme du Fabriquant de Table (TMD) [36].

Pour surmonter le TMD, nous utilisons une boucle de Ziv [104]. Cela consiste à calculer le résultat avec quelques chiffres de précision supplémentaire, et de le remplacer par un intervalle encadrant ce résultat. Si cet intervalle contient le point milieu entre deux nombres à virgules flottantes, alors ses deux bornes ne peuvent pas s'arrondir au même nombre à virgule flottante selon le mode d'arrondi choisi. Il est donc impossible de déterminer quelle est la valeur de l'arrondi correct à partir de ces bornes. On calcule donc de nouveau l'approximation r_2 de la mantisse à une précision plus grande. La boucle se termine lorsque les deux bornes s'arrondissent au même nombre à virgule flottante.

Par définition, l'exposant décimal est déterminé tel que

$$F = \lfloor \log_{10}(|x_2|) \rfloor - k_{10} + 1, \tag{4.5}$$

Algorithme 4.1 : Conversion binaire MPFR vers décimal MPD (`mpd_set_fr`), $c = 4$

Données : x_2 le nombre binaire MPFR tel que $x_2 = m \cdot 2^E$,
k_{10} la précision décimale attendue en sortie.
Résultat : x_{10} le nombre décimal MPD tel que $x_{10} = n \cdot 10^F$

1 $t_{2_{66}} \leftarrow \nabla 66(\log_{10}(|x_2|))$
2 $F \leftarrow \lfloor t_{2_{66}} - (k_{10} - 1) \rfloor$ ▷ Exposant entier F
3 $k'_2 \leftarrow \lceil k_{10} \cdot \log_2(10) + c \rceil$ ▷ Précision binaire temporaire k'_2
4 $r_2 \leftarrow \circ_{k'_2}(x_2 \times \circ_{k'_2}(10^{-F}))$ ▷ Mantisse r_2 en binaire MPFR
5 **si** r_2 *est* `inexact` **alors**
6 $\quad \underline{r}_2 \leftarrow \left(\left(r_2^-\right)^-\right)^-$ ▷ On note $r_2^- =$ `mpfr_nextbelow(r2)`
7 $\quad \overline{r}_2 = \left(\left(r_2^+\right)^+\right)^+$ ▷ On note $r_2^+ =$ `mpfr_nextabove(r2)`
8 $\quad r_2 \in [\underline{r}_2, \overline{r}_2]$ avec $\underline{r}_2, \overline{r}_2 \in \mathbb{F}^2_{k'_2}$ ▷ Dilemme du fabriquant de tables
9 $\quad [\underline{n}, \overline{n}] \leftarrow [\lfloor \underline{r}_2 \rfloor, \lceil \overline{r}_2 \rceil]$ ▷ Conversions en entiers GMP
10 $\quad$ **si** $\underline{n} \neq \overline{n}$ **alors**
11 $\quad\quad k'_2 \leftarrow 1.5 * k'_2$ ▷ Augmenter la précision binaire
12 $\quad\quad$ Aller à la ligne 4 ▷ Recalculer r_2 à la nouvelle précision k'_2
13 $\quad$ **sinon**
14 $\quad\quad n \leftarrow \underline{n}$ ▷ Arrondi correct de n
15 $\quad$ **fin**
16 **sinon**
17 $\quad n \leftarrow$ `mpfr_get_z`(n, r_2) ▷ r_2 est exact, conversion en entier GMP
18 **fin**
19 **si** $n = 10^{k_{10}}$ **alors**
20 $\quad F \leftarrow F + 1$ ▷ Correction de l'exposant
21 **fin**

modulo une retouche en cas d'arrondi qui fait changer de décade. On calcule ainsi en binaire la valeur $t_{2_{66}} = \lfloor \nabla 66 \left(\log_{10}(|x_2|)\right) \rfloor$ pour déterminer F. Nous choisissons de calculer cette valeur avec 66 bits de précision car nous savons que les exposants sont représentés sur des entiers signés de 64 bits, et ainsi $F \leq 2^{63} - 1$. La valeur $t_{2_{66}}$ est donc calculée avec 2 bits de garde. Le problème est donc de déterminer si $t_{2_{66}} = \lfloor \log_{10}(|x_2|) \rfloor$, c'est-à-dire s'il peut y avoir un entier L tel que

$$\nabla 66 \left(\log_{10}(|x_2|)\right) < L \leq \log_{10}(|x_2|). \tag{4.6}$$

Comme l'exposant F de la conversion est représenté sur un entier signé de 64 bits, on a $F \leq 2^{63} - 1$, soit $|x_2| \leq 10^{2^{63}-1}$, et donc $\log_{10}(|x_2|) \leq 2^{63} - 1$. Ceci entraîne que $\lfloor \log_{10}(|x_2|) \rfloor$ est un nombre à virgule flottante dès que $k_2 \geq 63$. L'équation (4.6) est donc impossible, et on a bien que $t_{2_{66}} = \lfloor \log_{10}(|x_2|) \rfloor$.

Pour calculer l'approximation de la mantisse r_2, on estime d'abord la précision binaire temporaire k'_2. Elle correspond en premier lieu à la précision binaire temporaire nécessaire pour calculer un nombre décimal à la précision k_{10} dans le cas général, telle que $k'_2 = k_{10} \cdot \log_2(10) + c$. On ajoute une constante c à k'_2 de façon à légèrement surestimer l'équivalent binaire de la précision k_{10}. Dans la plupart des cas cette précision est suffisante, dans le cas du TMD plus de précision est nécessaire. En pratique, on choisit $c = 4$.

On peut alors calculer une première approximation de la mantisse entière recherchée n, sous la forme d'un nombre à virgule flottante binaire r_2 à la précision k'_2 tel que $r_2 = \circ_{k'_2}(x_2 \times \circ_{k'_2}(10^{-F}))$ avec les fonctions MPFR correspondantes. Ces fonctions renvoient un drapeau `inexact` lorsque le résultat d'une opération ne peut pas être représenté exactement et est arrondi. Lorsqu'aucun arrondi a lieu, le résultat r_2 peut être converti en entier GMP sans problème.

Sinon on se trouve dans le cas du TMD, on entre dans la boucle de Ziv. On calcule $[\underline{r}_2, \overline{r}_2]$ l'intervalle encadrant r_2. Pour se faire, on utilise les fonctions binaires `mpfr_nextabove` et `mpfr_nextbelow`. On introduit les notations suivantes pour plus de lisibilité :

- $r_2^+ =$ `mpfr_nextabove`(r_2) est le premier nombre à virgule flottante supérieur à r_2,
- $r_2^- =$ `mpfr_nextbelow`(r_2) est le premier nombre à virgule flottante inférieur à r_2.

L'idée est de prendre l'intervalle le plus grand possible pour assurer l'arrondi correct et contenir la solution exacte, mais le plus petit possible pour minimiser le temps de calculs.

Lemme 1. *Si on a* $\underline{r}_2 = \left(\left(r_2^-\right)^-\right)^-$ *et* $\overline{r}_2 = \left(\left(r_2^+\right)^+\right)^+$*, alors la solution exacte* $x_2 \cdot 10^{-F}$ *est toujours comprise dans l'intervalle* $\left[\underline{r}_2, \overline{r}_2\right]$*, telle que*

$$\underline{r}_2 \leq x_2 \cdot 10^{-F} \leq \overline{r}_2. \tag{4.7}$$

Démonstration. Sachant que $r_2 = \circ_{k'_2}(x_2 \times \circ_{k'_2}(10^{-F})) = \left(x_2 \times (10^{-F} \cdot (1+\eta_F))\right) \cdot (1+\eta_{x_2})$, on peut écrire

$$r_2 = x_2 \cdot 10^{-F} \cdot (1+\eta), \qquad |\eta| \;\leq\; |\eta_{x_2}| + |\eta_F| + |\eta_{x_2}| \cdot |\eta_F|. \tag{4.8}$$

Comme ces opérations sont effectuées avec des fonctions MPFR, on peut assurer que $|\eta_{x_2}| \leq 2^{-k'_2}$ et $|\eta_F| < 2^{-k'_2}$. On a donc que $|\eta| \leq 2^{-k'_2} \cdot \left(2 + 2^{-k'_2}\right)$. D'après sa définition, la variable k'_2 est bornée par c telle que $k'_2 \geq c$. Rappelons que nous avons choisi $c = 4$, ce qui implique qu'on peut borner $2^{-k'_2}$ tel que $2^{-k'_2} \leq 2^{-4}$. On peut donc borner η de la façon suivante

$$|\eta| \;\leq\; \left(2 + 2^{-4}\right) \cdot 2^{-k'_2} \;\leq\; 2.5 \cdot 2^{-k'_2}. \tag{4.9}$$

Montrons dans un premier temps que $x_2 \cdot 10^{-F} \leq \overline{r}_2$, pour $x_2 \cdot 10^{-F}$ positif.
La fonction `mpfr_nextabove` renvoie le premier nombre à virgule flottante binaire supérieur à r_2, ce qui revient à renvoyer un ulp supplémentaire à r_2 de sorte que $r_2^+ = r_2 + \text{ulp}_{k_2'}(r_2)$, où k_2' est la précision de r_2. Cela signifie qu'on peut réécrire $\overline{r}_2$ tel que

$$\overline{r}_2 = r_2^+ + \text{ulp}_{k_2'}\left(r_2^+\right) + \text{ulp}_{k_2'}\left(r_2^+ + \text{ulp}_{k_2'}\left(r_2^+\right)\right). \tag{4.10}$$

On peut simplifier cette équation telle que

$$\overline{r}_2 = r_2 \cdot (1+\overline{\theta}), \qquad \overline{\theta} = \frac{1}{r_2} \cdot \left(\text{ulp}_{k_2'}(r_2) + \text{ulp}_{k_2'}\left(r_2^+\right) + \text{ulp}_{k_2'}\left(r_2^+ + \text{ulp}_{k_2'}\left(r_2^+\right)\right)\right). \tag{4.11}$$

La fonction ulp étant monotone et strictement croissante, alors $r_2 < r_2^+$ et on a que $\text{ulp}_{k_2'}(r_2) \leq \text{ulp}_{k_2'}\left(r_2^+\right) \leq \text{ulp}_{k_2'}\left(r_2^+ + \text{ulp}_{k_2'}\left(r_2^+\right)\right)$. Cela revient à dire que

$$\frac{1}{r_2} \cdot 3 \cdot \text{ulp}_{k_2'}(r_2) \leq \frac{1}{r_2} \cdot \left(\text{ulp}_{k_2'}(r_2) + \text{ulp}_{k_2'}\left(r_2^+\right) + \text{ulp}_{k_2'}\left(r_2^+ + \text{ulp}_{k_2'}\left(r_2^+\right)\right)\right) = \overline{\theta}. \tag{4.12}$$

D'après la définition de l'ulp, définition 4 donnée dans le chapitre 2, on peut réécrire cette borne inférieur sur $\overline{\theta}$ telle que $3 \cdot \frac{1}{r_2} \cdot \text{ulp}_{k_2'}(r_2) = 3 \cdot 2^{-\log_2(|r_2|)} \cdot 2^{\lfloor \log_2(|r_2|) \rfloor - k_2' + 1}$. D'après la définition de la partie entière inférieure, on a $2^{-1} \leq 2^{\lfloor \log_2(|r_2|) \rfloor - \log_2(|r_2|)} < 2^0$. D'où $3 \cdot 2^{-k_2'} \leq 3 \cdot \frac{1}{r_2} \cdot \text{ulp}_{k_2'}(r_2) < 3 \cdot 2^{-k_2'+1}$. Finalement on borne $\overline{\theta}$ tel que

$$\overline{\theta} \geq 3 \cdot 2^{-k_2'}. \tag{4.13}$$

En injectant l'équation (4.8) dans l'équation (4.11), on a donc

$$\overline{r}_2 = x_2 \cdot 10^{-F} \cdot (1+\eta) \cdot (1+\overline{\theta}). \tag{4.14}$$

La combinaison des équations (4.13) et (4.9) donne la borne inférieure suivante $1+\eta+\overline{\theta} \cdot (1+\eta) \geq 1 - 2.5 \cdot 2^{-k_2'} + 3 \cdot 2^{-k_2'} \cdot \left(1 - 2.5 \cdot 2^{-k_2'}\right)$, soit $1+\eta+\overline{\theta} \cdot (1+\eta) \geq 1 + 0.5 \cdot 2^{-k_2'} - 7.5 \cdot 2^{-2 \cdot k_2'}$. Or $7.5 \cdot 2^{-2 \cdot k_2'} < 0.5 \cdot 2^{-k_2'}$ pour tout $k_2' \geq 4$, ce qui est assuré par la constante c. Ceci implique que $1 + 0.5 \cdot 2^{-k_2'} - 7.5 \cdot 2^{-2 \cdot k_2'} \geq 1$, et finalement on a

$$\begin{aligned} 1 &\leq 1+\eta+\overline{\theta} \cdot (1+\eta), \\ x_2 \cdot 10^{-F} &\leq x_2 \cdot 10^{-F} \cdot (1+\eta) \cdot (1+\overline{\theta}) = \overline{r}_2. \end{aligned} \tag{4.15}$$

Par symétrie, le raisonnement est similaire pour montrer que $\underline{r}_2 \leq x_2 \cdot 10^{-F}$ lorsque $x_2 \cdot 10^{-F}$ est négatif. Il convient cependant de remplacer `mpfr_nextabove` par `mpfr_nextbelow`.

Montrons maintenant que $\underline{r}_2 \leq x_2 \cdot 10^{-F}$, pour $x_2 \cdot 10^{-F}$ positif.
Dans ce cas, on utilise la fonction `mpfr_nextbelow` pour le calcul de $\underline{r}_2$, qui renvoie le

premier nombre à virgule flottante binaire inférieur à r_2. Ainsi, pour s'assurer que l'approximation $\underline{r}_2$ est bien inférieure à la solution exacte $x_2 \cdot 10^{-F}$, on doit distinguer deux cas.

Soit un des trois appels à `mpfr_nextbelow` fait intervenir un changement de binade. Cela signifie que l'on passe d'un nombre à virgule flottante représenté dans un intervalle $[2^i\,;\,2^{i+1}[$ à un nombre à virgule flottante représenté dans la binade du dessous $[2^{i-1}\,;\,2^i[$. Ce phénomène n'intervient que lorsqu'on a $r_2 = 2^i$. La taille de l'ulp passe alors de $\mathrm{ulp}_k(2^i) = 2^{i-k+1}$ à $\mathrm{ulp}_k(2^{i-1}) = 2^{i-k} = \frac{1}{2} \cdot \mathrm{ulp}_k(2^{i-1})$. Comme $k_2' \geq c$ on assure que le nombre d'ulps par binade est supérieur ou égal à $2^{k_2'-1}$. Dans notre cas on a $c = 4$, donc le nombre d'ulps par binade est supérieur ou égal à 8. Cela signifie qu'en faisant trois appels à `mpfr_nextbelow`, on effectue au plus un seul changement de binade.

Soit les trois appels à `mpfr_nextbelow` s'effectuent dans la même binade, ce qui n'influe pas sur la taille de l'ulp.

On peut alors définir le résultat de la fonction `mpfr_nextbelow` tel que, il existe un $i \in \mathbb{Z}$ pour lequel

$$r_2^- = \begin{cases} r_2 - \mathrm{ulp}_{k_2'}(r_2) & \text{si } r \neq 2^i, \\ r_2 - \dfrac{1}{2}\,\mathrm{ulp}_{k_2'}(r_2) & \text{si } r = 2^i. \end{cases} \tag{4.16}$$

Soient $\alpha_j \in \left\{\frac{1}{2}, 1\right\}$ avec $j \in \{1,2,3\}$. En notant $\left(r_2^-\right)_{\alpha_1} = r_2 - \alpha_1 \cdot \mathrm{ulp}_{k_2'}(r_2)$, on définit $\underline{r}_2$ tel que

$$\underline{r}_2 = \left(r_2^-\right)_{\alpha_1} - \alpha_2 \cdot \mathrm{ulp}_{k_2'}\left(\left(r_2^-\right)_{\alpha_1}\right) - \alpha_3 \cdot \mathrm{ulp}_{k_2'}\left(\left(r_2^-\right)_{\alpha_1} - \alpha_2 \cdot \mathrm{ulp}_{k_2'}\left(\left(r_2^-\right)_{\alpha_1}\right)\right). \tag{4.17}$$

avec $\alpha_1 + \alpha_2 + \alpha_3 \geq 2.5$. Comme la fonction ulp est positive, alors $\left(r_2^-\right)_{\alpha_1} > r_2$. Elle est de plus monotone et strictement croissante, on a donc que $-\alpha_j \cdot \mathrm{ulp}_{k_2'}\left(\left(r_2^-\right)_{\alpha_1}\right) \geq -\alpha_j \cdot \mathrm{ulp}_{k_2'}(r_2)$. Ceci implique que

$$\underline{r}_2 \;\geq\; r_2 - (\alpha_1 + \alpha_2 + \alpha_3) \cdot \mathrm{ulp}_{k_2'}(r_2) \;\geq\; r_2 - 2.5 \cdot \mathrm{ulp}_{k_2'}(r_2). \tag{4.18}$$

On peut donc réécrire $\underline{r}_2$ comme $\underline{r}_2 = r_2 \cdot (1 + \underline{\theta})$, avec $\underline{\theta} = \frac{1}{r_2} \cdot 2.5 \cdot \mathrm{ulp}_{k_2'}(r_2)$. De la même façon que nous avons borné $\overline{\theta}$ dans l'équation (4.11), on peut borner $\underline{\theta}$ par $\underline{\theta} \leq -2.5 \cdot 2^{-k_2'}$

Ainsi, en suivant le raisonnement du cas précédent, on détermine que $\underline{r}_2 = x_2 \cdot 10^{-F} \cdot (1+\eta) \cdot (1+\underline{\theta})$. On prend une nouvelle borne plus large sur η tel que $|\eta| \leq \left(2 + 2^{-4}\right) \cdot 2^{-k_2'} \leq 3 \cdot 2^{-k_2'}$. Ceci implique que $(1+\eta) \cdot (1+\underline{\theta}) \geq 1$. On a donc $\underline{r}_2 \geq x_2 \cdot 10^{-F}$.

Le raisonnement est similaire par symétrie pour montrer que $\underline{r}_2 \leq x_2 \cdot 10^{-F}$ lorsque $x_2 \cdot 10^{-F}$ est négatif. Il convient cependant de remplacer la fonction `mpfr_nextabove` par `mpfr_nextbelow`. □

Quelle que soit la précision k_2', le lemme 1 assure que la solution exacte se trouve dans l'intervalle $[\underline{r}_2, \overline{r}_2]$. Ces bornes sont ensuite converties en nombres entiers de sorte que

$\underline{n} = \lfloor \overline{r}_2 \rfloor$ et $\overline{n} = \lceil \underline{r}_2 \rceil$. Si $\underline{n} = \overline{n}$, alors la mantisse $x_2 \cdot 10^{-F}$ est égale à cette valeur. Sinon, les calculs de r_2 et de ces bornes n'étaient pas assez précis. On augmente donc la précision k_2' et on recommence l'algorithme à partir du calcul de r_2.

L'algorithme se termine, 10 étant divisible par 2, la conversion d'un format binaire vers un format décimal devient exacte à une certaine précision finie. Ainsi, augmenter la précision k_2' permet d'ajouter de nouvelles informations, et permet d'atteindre la précision pour laquelle $\underline{n} = \overline{n}$.

La conversion d'un nombre décimal x_{10} en un nombre binaire x_2 suit essentiellement les mêmes principes. On calcule d'abord le nombre MPFR à une précision temporaire k_2' tel que

$$x_2 = \circ_{k_2'}\left(x_{10} \times \circ_{k_2'}\left(10^F\right)\right). \tag{4.19}$$

On représente alors le nombre x_2 par un intervalle et on essaye de l'arrondir à la précision de sortie désirée. Si le calcul est inexact ou si cet arrondi n'est pas possible, alors on effectue une boucle de Ziv en augmentant la précision temporaire k_2' et en recalculant x_2.

4.3 Fonctions élémentaires

Une fois l'arithmétique décimale multi-précision de base construite, on peut s'en servir pour développer les fonctions élémentaires et enrichir notre environnement. Nous allons d'abord présenter une méthode générale d'étude de la propagation d'erreur qui pourra être applicable pour le développement de plusieurs fonctions, pour ensuite étudier ces fonctions au cas par cas.

4.3.1 Discussions à propos de la propagation d'erreurs

Avant d'aller plus loin dans la construction de cette bibliothèque arithmétique, prenons un moment pour analyser la propagation d'erreurs. En effet, le calcul de fonctions transcendantes telle que l'exponentielle ou le logarithme demande plusieurs conversions entre nombres décimaux et binaires, introduisant ainsi une erreur qui se propagera tout au long du calcul.

Il y a trois différents types d'erreurs pouvant apparaître dans le cas général du calcul d'une fonction décimale $D\colon \mathbb{F}_{k_{10}}^{10} \to \mathbb{F}_{k_{10}}^{10}$, approximation d'une fonction réelle $f\colon \mathbb{R} \to \mathbb{R}$. Nous supposons pour le développement de notre bibliothèque qu'il existe déjà une fonction binaire multi-précision donnant une approximation de f, notée $B\colon \mathbb{F}_{k_2}^{2} \to \mathbb{F}_{k_2}^{2}$.

La première est l'erreur de conversion ε du nombre décimal $x_{10} \in \mathbb{F}_{k_{10}}^{10}$ vers le nombre binaire $x_2 \in \mathbb{F}_{k_2}^{2}$, telle que

$$x_2 \;=\; \texttt{mpd\_get\_fr}(x_{10}) \;=\; \circ_{k_2}(x_{10}) \;=\; x_{10}\cdot(1+\varepsilon), \qquad |\,\varepsilon\,| \le 2^{-k_2}, \quad \varepsilon \in \mathbb{R}. \tag{4.20}$$

La seconde est l'erreur de calcul θ survenant lors de l'appel de la fonction B telle que

$$y_2 \;=\; B(x_2) \;=\; f(x_2)\cdot(1+\theta). \tag{4.21}$$

Si la fonction B fournit un arrondi correct vers le plus proche, on a alors $|\theta| \leq 2^{-k_2'}$, $\theta \in \mathbb{R}$. De la même façon que dans l'algorithme 4.1, nous passons par un intermédiaire binaire très précis afin d'assurer l'arrondi correct du résultat en décimal à la précision k_{10}. Ainsi la précision k_2' dans un premier temps telle que $k_2' = k_2 + c$, avec $c = 4$, et une boucle de Ziv est effectuée sur le résultat y_2 qui est réévalué avec une précision k_2' plus grande si nécessaire, de la même façon que dans l'algorithme de conversion. Par conséquent, la troisième est l'erreur de conversion η du nombre binaire calculé vers le résultat décimal telle que

$$y_{10} \;=\; \circ_{k_{10}}\left(y_2\right) \;=\; y_2\cdot\left(1+\eta\right), \qquad |\eta| \leq 10^{-k_{10}}, \quad \eta \in \mathbb{R}. \tag{4.22}$$

En résumé, on peut décrire la fonction décimale D telle que

$$\begin{aligned} D(x_{10}) \;&=\; \circ_{k_{10}}\left(B\left(\circ_{k_2}(x_{10})\right)\right) \;=\; B(x_{10}\cdot(1+\varepsilon))\cdot\left(1+\eta\right) \\ D(x_{10}) \;&=\; f(x_{10}\cdot(1+\varepsilon))\cdot\left(1+\eta\right)\cdot(1+\theta). \end{aligned} \tag{4.23}$$

Passer par une précision intermédiaire permettant d'assurer l'arrondi correct du résultat y_2 à la précision décimale k_{10} implique que, selon les hypothèses d'arrondi correct de la fonction intermédiaire B, l'erreur qui aura le plus d'impact sur le résultat y_{10} est la première erreur de conversion ε. On réécrit alors D telle que $D(x_{10}) \;=\; f(x_{10}\cdot(1+\varepsilon))$.

On peut approcher la fonction $D(x_{10})$ par $f(x_{10}\cdot(1+\varepsilon)) = f(x_{10}+x_{10}\varepsilon)$. Soit un entier $n \in \mathbb{N}$, si f est n fois dérivable en le point $x_{10} \in \mathbb{F}_{k_{10}}^{10} \subset \mathbb{R}$, alors par l'application du théorème de Taylor-Lagrange, il existe un nombre réel ξ entre x_{10} et $x_{10}\cdot(1+\varepsilon)$ tel que

$$f(x_{10}+x_{10}\varepsilon) \;=\; f(x_{10}) + f'(\xi)\cdot x_{10}\varepsilon \;=\; f(x_{10})\cdot\left(1+\frac{x_{10}f'(\xi)}{f(x_{10})}\varepsilon\right). \tag{4.24}$$

Sous cette forme, on voit facilement qu'il est suffisant de calculer le premier terme car les termes d'ordre supérieur sont négligeable pour l'approximation de $D(x_{10})$. Dans ce cas, la stabilité de l'erreur est liée au conditionnement de la fonction, qui quantifie à quel point la valeur de sortie est affectée par de petites variations en entrée. Si le conditionnement est borné, on peut assurer que la propagation de l'erreur aura un impact négligeable dans l'algorithme. Cependant, si ce n'est pas le cas, nous devrons trouver une méthode pour évaluer la fonction d'une autre façon.

4.3.2 Exponentielle

Étudions maintenant le cas de la fonction exponentielle. Nous voulons d'abord borner l'erreur de conversion comme expliqué dans le protocole précédent décrit section 4.3.1.

$$\begin{aligned} \exp(x_2) &= \exp(x_{10}\cdot(1+\varepsilon)) \\ &= \exp(x_{10})\cdot\exp(x_{10}\varepsilon), \\ \exp(x_2) &= \exp(x_{10})\cdot\left(1+\varepsilon'\right), \qquad \varepsilon' = \exp(x_{10}\varepsilon)-1. \end{aligned} \tag{4.25}$$

On peut donc réécrire l'erreur ε telle que $\varepsilon = \frac{\log(1+\varepsilon')}{x_{10}}$. Sous cette forme, on voit qu'il est facile de borner cette erreur comme $|\varepsilon| \leq 10^{-k_{10}}$ pour tout $x_{10} \in \mathbb{F}^{10}_{k_{10}}$. Ceci implique qu'on peut écrire la fonction exponentielle décimale en utilisant la méthode de conversion du nombre décimal pour utiliser la fonction exponentielle binaire, et convertir le résultat de nouveau en décimal. Si l'erreur croît avec l'exponentielle, elle est toujours négligeable par rapport à la précision du résultat attendu.

4.3.3 Logarithme

Le cas du logarithme est un peu plus complexe. On peut le réécrire de la façon suivante

$$\begin{aligned} \log(x_2) &= \log(x_{10}\cdot(1+\varepsilon)) \\ \log(x_2) &= \log(x_{10})\cdot\left(1+\frac{1}{\log(x_{10})}\cdot\frac{\log(1+\varepsilon)}{\varepsilon}\cdot\varepsilon\right) \end{aligned} \tag{4.26}$$

On peut voir sous cette forme que l'erreur ε est directement liée au terme $\frac{1}{\log(x_{10})}$. Cela signifie que lorsque cette quantité n'est pas bornée, l'erreur croît considérablement.

L'évaluation du logarithme en le point $x_{10} = 1$ peut être définie manuellement telle que $\log(x_{10}) = 0$. Cependant autour de ce point, l'erreur produite par la conversion du nombre décimal en binaire est amplifiée par le calcul du logarithme. Dans ce cas, atteindre un arrondi correct avec notre algorithme de conversion sur la valeur en sortie peut nécessiter un grand nombre d'itérations, répétant ainsi les appels à l'opération de logarithme binaire en MPFR. Le temps d'exécution de l'algorithme n'est alors plus compétitif pour une bibliothèque arithmétique multi-précision décimale rapide.

On peut s'affranchir de ce problème facilement en utilisant judicieusement les fonctions de MPFR à notre disposition, notamment en appelant la fonction `mpfr_log1p` autour de 1. Pour se faire, on calcule d'abord $t_{10} = x_{10} - 1$, avec x_{10} autour de 1, c'est-à-dire que nous bornons x_{10} tel que $\frac{1}{2} \leq x_{10} \leq 2$. Alors, par l'application du lemme 2, ou lemme de Sterbenz, cette soustraction décimale est exacte.

Lemme 2 (Sterbenz [105])**.** *Soit un système de nombres à virgules flottantes en base β avec des nombres dénormalisés, si x et y sont deux nombres à virgules flottantes finis tels que*

$$\frac{y}{2} \leq x \leq 2y, \tag{4.27}$$

alors $x - y$ est représentable exactement.

La variable t_{10} est ensuite convertie en binaire à une précision k_2 telle que $t_2 \leftarrow$ `mpd_get_fr`(t_2, t_{10}). On appelle `mpfr_log1p` sur cette variable t_2, de sorte que le résultat calculé est l'arrondi de $\log(1+t_2) = \log(1+(x_{10}-1)(1+\varepsilon)) = \log(x_{10}(1+\varepsilon))$. Finalement, nous calculons bien le logarithme souhaité correctement arrondi autour de 1.

Traiter ponctuellement les cas où l'erreur de conversion est amplifiée dans l'algorithme n'est pas toujours possible, notamment lorsqu'il y a un grand nombre de points difficiles, comme dans le cas des fonctions trigonométriques telles que sin et cos.

4.4 Résultats expérimentaux

Ces études théoriques ont mené par application expérimentale au développement d'une bibliothèque d'arithmétique décimale multi-précision et d'une bibliothèque d'arithmétique d'intervalles décimale multi-précision. Ces développements ont été soigneusement testés pour vérifier leur validité ainsi que leurs performances temporelles.

4.4.1 Test de validité

La validité des résultats de notre bibliothèque d'arithmétique MPDI repose sur l'arrondi correct de notre bibliothèque MPD. Pour tester cette dernière, nous avons besoin de fournir en entrée un nombre décimal avec un mode d'arrondi et une précision, et d'attendre en sortie un nombre décimal à une certaine précision. Nous devons avoir une certaine confiance dans ce nombre en sortie pour valider la vérification de l'exactitude de l'arrondi correct de notre bibliothèque arithmétique. Nous souhaitons également vérifier nos algorithmes sur un grand nombre d'entrées, choisies avec soin permettant de maximiser la couverture des différents comportements de nos fonctions.

Nous souhaitons éprouver nos développements en comparant nos résultats à ceux de la bibliothèque *mpdecimal*. Elle fournit un framework de test complet que nous pouvons réutiliser tel quel pour valider nos algorithmes décimaux. Ces tests passent des variables décimales converties depuis des chaînes de caractères aux fonctions arithmétiques sous tests, ainsi qu'un résultat attendu, avec une précision et un mode d'arrondi.

Nous souhaitons également effectuer une seconde campagne de test en nous inspirant de la campagne de test développée pour la bibliothèque MPFR. Elle contient notamment

des tests ayant pour but de débusquer des erreurs d'arithmétique multi-précision binaire bien connus. Nous souhaitons éprouver les bibliothèques décimales quant à ces erreurs. L'utilisation de ces tests demande un effort important pour trier les erreurs dues aux conversions entre binaire et décimal des réelles erreurs d'implémentation présente dans notre bibliothèque.

Notre bibliothèque MPD s'est montrée robuste à ces erreurs en passant ces deux campagnes de test. L'application de cette seconde campagne de test sur *mpdecimal* a cependant permis de faire remontrer quelques erreurs. Par exemple le test de l'addition suivant

```
check16_add("2.9928048191891653800e+272", "5.34637717585790933424e
    +271", "3.5274425367757074714 24e272", MPD_RNDN);
```

teste l'addition $2.992804819189917 \cdot 10^{272} + 5.346377175857909 \cdot 10^{271}$ à la précision décimale $k = 16$ arrondie au plus proche a le résultat attendu arrondi $3.527442536775708 \cdot 10^{272}$. Dans le cas de l'addition de *mpdecimal*, ce résultat est bien calculé, mais une erreur est soulevée car l'arrondi de la chaîne de caractères représentant la solution attendue donne le résultat $3.527442536775707 \cdot 10^{272}$. De ce fait, plusieurs erreurs d'arrondi ont été relevées dans les tests de *mpdecimal*.

Le test de la bibliothèque MPDI reprend le principe de la campagne de test de MPFI, son développement reprenant les mêmes principes. Les mêmes considérations qu'avec MPD et MPFR sont à prendre en compte dans le processus de validation des résultats.

4.4.2 Tests de performances

Les tests ont été effectués sur un processeur Intel ® Core™ i7-8650U CPU @ 1.90GHz × 8, sous un système d'exploitation Ubuntu 16.04.5 LTS 64 bits. Les versions des bibliothèques utilisées pour l'implémentation sont GMP 6.1.2 et MPFR 4.0.1. La comparaison des performances temporelle est faite avec les bibliothèques MPFR 4.0.1, MPFI 1.5.3 et *mpdecimal* 2.4.2. Tous les tests sont compilés avec gcc version 5.4.0.

4.4.2.1 Détails d'implémentation

En plus des fonctions arithmétiques décrites dans la section 4.2 et des fonctions élémentaires de la section, un certain nombre d'autres fonctions de bases ont été implémentées dans nos bibliothèques MPD et MPDI. Parmi elles se trouve la comparaison, implémentée à la fois en MPD et pour les intervalles décimaux. Ces fonctions utilisent également l'algorithme de conversion.

Cet algorithme de conversion est utilisé dans approximativement toutes les fonctions des bibliothèques MPD et MPDI. Cet algorithme fait appel à la fois à des fonctions MPFR et

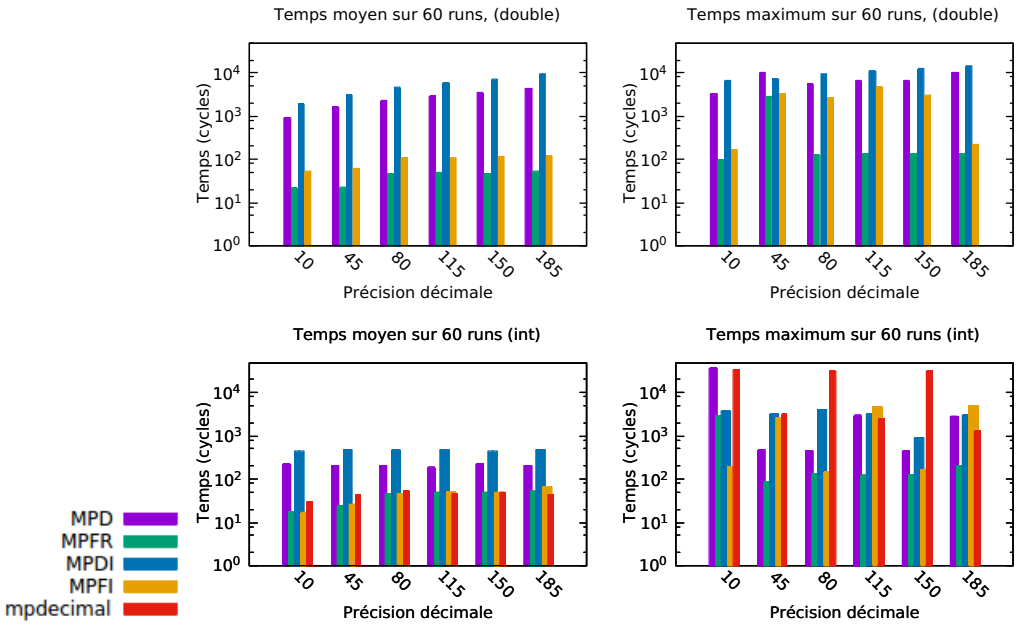


FIGURE 4.3 – Temps d'exécution de l'addition

GMP pour calculer le nombre converti en décimal ou en binaire. Lorsque la précision pour arrondir ce nombre calculé n'est pas suffisante, l'algorithme est itéré un certain nombre de fois pour atteindre la précision attendue. On s'attend donc à ce que les performances de la bibliothèque MPD soient liées à la vitesse d'exécution de la bibliothèque MPFR.

La bibliothèque MPDI se comporte de la même façon que la bibliothèque MPFI, calculant deux fois le même algorithme pour déterminer les bornes de l'intervalle. Ceci implique qu'on s'attend à ce que la bibliothèque MPDI soit environ deux fois plus lente que la bibliothèque MPD.

4.4.2.2 Performances temporelles

Pour illustrer les performances temporelles de nos bibliothèques MPD et MPDI, nous choisissons de comparer les résultats de l'algorithme d'addition, de l'exponentielle et du logarithme avec leurs contreparties binaires en MPFR et MPFI mais également avec les algorithmes de la bibliothèque *mpdecimal*.

Les performances sont calculées pour un échantillon d'entrées aléatoires données à partir d'entiers `int64` et de nombres à virgule flottante binaire `double`. La bibliothèque *mpdecimal* ne proposant pas de conversion à partir `double`, nous n'avons pas tenté de tester son comportement dans un contexte similaire. Nous déduisons le temps d'appel

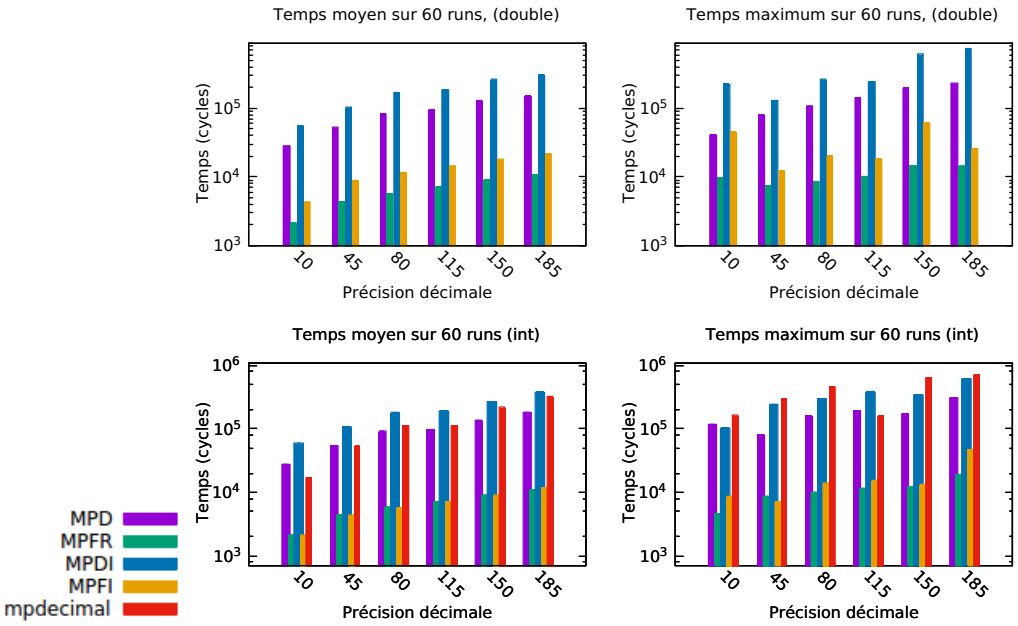


FIGURE 4.4 – Temps d'exécution de l'exponentielle

pour une fonction vide ayant le même prototype, et nous effectuons les mesures temporelles 60 fois, après avoir préchauffer les caches, pour lisser les effets dus aux processeurs superscalaires modernes.

Dans ces conditions, nous pouvons faire plusieurs observations, dans un premier temps sur l'algorithme d'addition représenté figure 4.3. Dans toutes les figures, le nombre de cycle est représenté en échelle logarithmique pour faciliter la lisibilité. On voit donc pour l'addition les performances moyennes des algorithmes MPD et MPDI sont jusqu'à 5 fois plus lents que leurs contreparties binaires, aussi bien avec des variables entières qu'à virgule flottante. L'algorithme d'addition de *mpdecimal* a des performances comparables à MPFR et MPFI. Notre bibliothèque MPD n'est donc pas très compétitive dans ce contexte, car le calcul des derniers nombres de la mantisse pour fournir l'arrondi correct est très gourmand. Cette observation peut également être faite pour la soustraction et la multiplication.

Cependant, lorsque l'opération étudiée repose sur l'opération binaire MPFR, notre bibliothèque MPD commence à être compétitive. On peut voir en effet dans les figures 4.4 et 4.4 illustrant les performances des différentes bibliothèques pour l'exponentielle et le logarithme respectivement, que même si, comme attendu, les bibliothèques décimales sont plus lentes que MPFR et MPFI, les performances de MPD sont comparables à celles de *mpdecimal* à partir d'une précision décimale de 45 chiffres pour l'exponentielle, et dès 10 chiffres pour le logarithme.

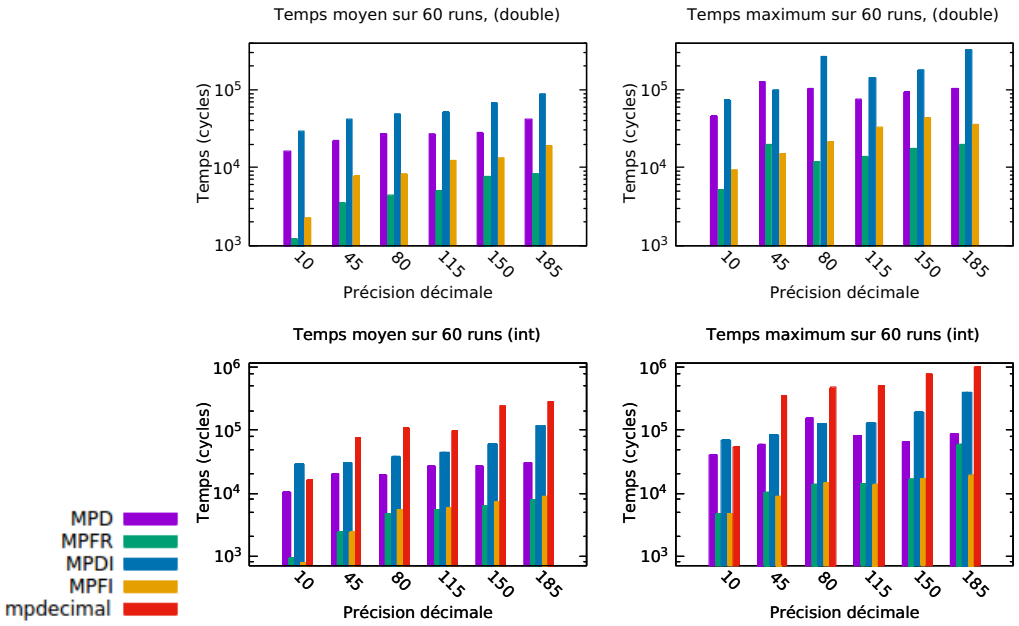


FIGURE 4.5 – Temps d'exécution du logarithme

4.5 Conclusion

Dans cette partie nous avons présenté deux bibliothèques arithmétiques, multi-précision décimale, MPD, et d'intervalles multi-précision décimale, MPDI. Ces bibliothèques proposent des algorithmes d'arithmétique de base correctement arrondis pour une précision décimale arbitraire. Nous avons montré notre approche en fournissant une preuve écrite de notre algorithme de conversion d'un format binaire vers le format décimal. Notre implémentation a été amplement testée et comparée aux résultats d'une bibliothèque décimale multi-précision existante, *mpdecimal*. De plus, notre représentation décimale est compatible avec la norme IEEE 754-2008, et l'arithmétique d'intervalle est conforme avec la norme IEEE 1788-2015 *Standard for Interval Arithmetic*.

Avec ces bibliothèques, nous répondons à un besoin d'arithmétique décimale certifiée pour permettre la génération de tests à partir d'un modèle d'exigences, introduisant le moins d'erreurs possibles, ou tout du moins une erreur décimale maîtrisée. Notre implémentation a été pensée pour fournir l'arrondi correct en se basant sur les bibliothèques binaires éprouvées GMP et MPFR. Quand bien même nos algorithmes peuvent être jusqu'à 12 fois plus lents que leurs contreparties binaires, ces performances sont acceptables dans le cadre de l'utilisation pour la génération automatique de tests pour un logiciel cible. L'assurance apportée par l'arrondi correct décimal contre des performances plus lentes

qu'en binaire est un compromis acceptable. De plus, nous parvenons tout de même à être plus efficace que les bibliothèques décimales existantes tout en ayant une plus grande confiance dans nos résultats.

CHAPITRE 5

VERS L'ARITHMÉTIQUE IEEE 754 EN BASES MIXTES

La norme IEEE 754-2008 [38] décrit l'arithmétique à virgule flottante pour des formats binaires et décimaux. Elle spécifie les opérations de conversion entre la base 2 et 10, mais ne fournit aucune indication sur la combinaison de formats à virgule flottante de bases différentes dans une même opération. En revanche, la norme décrit les opérations mélangeant dans une opération différents formats dans une même base.

Faire le lien entre l'arithmétique à virgule flottante binaire et son équivalent décimal est pourtant crucial dans le contexte de la certification aéronautique. En effet, comme vu précédemment, la spécification étant écrite dans un format décimal et le code en arithmétique à précision finie binaire, faire le lien entre les deux de manière certifiée peut s'avérer délicat. L'arithmétique multi-précision et l'arithmétique d'intervalles peuvent être utilisées pour la validation et vérification par le test, mais la solution proposée dans le chapitre 4 contraint l'utilisateur a choisir d'effectuer ses calculs uniquement en décimal.

Nous proposons donc une solution plus flexible, afin d'étendre la norme en spécifiant toutes les opérations arithmétiques de base entre tous les formats disponibles. Dans ce chapitre, nous fournissons un algorithme avec arrondi correct pour un Fused-Multiply-Add (FMA) en bases mixtes. Sur cet algorithme repose l'arithmétique en bases mixtes pour toutes les opérations arithmétiques élémentaires $+, -, \times, \div, \sqrt{}$. Cet algorithme prend en entrée une combinaison de nombres IEEE 754 dans les formats binary64, binary128, decimal64 et decimal128, et renvoie un résultat IEEE 754 binary64, binary128, decimal64 ou decimal128, arrondi correctement selon les cinq modes d'arrondis IEEE 754. Notre implémentation ne demande aucune allocation de mémoire dynamique et son temps d'exécution peut être borné statiquement. Nous comparons notre implémentation à une implémentation d'un FMA en bases mixtes simple, développé grâce à la bibliothèque GMP [49].

Ces travaux s'inscrivent dans la continuité d'une étude ayant mené à la publication et présentation d'un article lors de la conférence *IEEE 25th Symposium on Computer Arithmetic (ARITH)* [15] en 2018.

5.1 Combiner les formats IEEE 754 binaires et décimaux

5.1.1 Problématique

Lors du développement d'un logiciel utilisant l'arithmétique à virgule flottante, l'utilisateur est confronté au choix entre deux bases : l'arithmétique à virgule flottante binaire (en base 2), ou l'arithmétique à virgule flottante décimale (en base 10). La norme IEEE 754-2008 pour l'arithmétique à virgule flottant [38] définit les formats et les opérations pour les deux bases, spécifiant qu'un *environnement de programmation peut être conforme à cette norme pour une de ces bases ou pour les deux*. Un certain nombre d'études sur l'arithmétique à virgule flottante décimale ont été entreprises dans un effort pour fournir des opérations décimales performantes et précises [103], ainsi que pour les opérations de conversion [106]. Ces algorithmes assurent des résultats avec arrondi fidèle, ou de précision comparable aux opérations binaires.

Nous souhaitons aller plus loin et effectuer des opérations à virgule flottante en "bases mixtes" correctement arrondies, soit que les opérandes de ces opérations soient représentées dans un format à virgule flottante binaire ou décimal.

Ce genre de situation peut survenir lorsque l'utilisateur est obligé de se servir de deux bibliothèques scientifiques différentes, l'une renvoyant un résultat binaire et l'autre un résultat décimal. Pour effectuer des opérations en bases mixtes, l'utilisateur pourrait être tenté de combiner les formats à virgule flottante binaires et décimaux comme dans le code C suivant :

```
double a=2000.0, r;
_Decimal64 b=0.001D;
r = a * b;
```

La compilation de ce code, par exemple avec gcc 6.3.0, échoue et le compilateur émet un message d'erreur précisant que les différents types des variables ne sont pas compatibles et ne peuvent être mélangés.

Une solution simple pour effectuer ce calcul serait de convertir toutes les variables dans un même type par convertion de type, forçant ainsi le compilateur a effectuer l'opération. Cependant ces conversions pouvant introduire des erreurs, pouvant s'accumuler et avoir un impact significatif sur le reste de l'algorithme par amplification, il est fortement déconseillé d'utiliser cette méthode. Elle peut fonctionner dans certains cas, mais nous souhaitons fournir une solution méthodique permettant de garantir le résultat à l'utilisateur.

Nous pouvons étendre notre exemple pour illustrer cet exemple dangereux, tel que :

```
double a=2000.0, c=-2.0, r;
_Decimal64 b=0.001D;
r = __builtin_fma(a,(double)b,c);
```

Ce code compile, cependant la valeur affectée à la variable `r` est $4.16333634 \cdot 10^{-17}$, ce qui ne correspond pas au résultat correct attendu, soit $2000 \times 0.001 - 2 = 0$. Cette erreur émane de la conversion de la variable `b` du format decimal64 vers le format à virgule flottante binaire.

Toutes ces raisons motivent le besoin pour l'étude et la conception d'un framework spécifique pour l'arithmétique en bases mixtes. Jusqu'à présent, de précédentes études se sont penchées sur la faisabilité d'une comparaison exacte entre différents formats à virgule flottante en bases mixtes [14], dans une démarche d'enrichir l'environnement arithmétique à virgule flottante et de façon à rendre les logiciels numériques plus sûrs. De plus, l'application de l'arithmétique en bases mixtes aux opérations calculatoires de base définies par la norme IEEE 754 $+, -, \times, \div, \sqrt{}$ et FMA a été étudiée pour les formats à virgule flottante binary64 et decimal64 dans de précédents travaux [15]. L'opération de FMA, ajoutée dans la version 2008 de la norme, effectue le calcul $a \times b + c$ avec un seul arrondi correct, au lieu de deux (un pour la multiplication et l'autre pour l'addition).

Nous souhaitons étendre ces résultats, en fournissant des algorithmes pour le développement de toutes les opérations calculatoires en bases mixtes pour une combinaison de tous les formats à virgule flottant, binaires et décimaux, décrits dans la norme, et assurer un résultat binaire ou décimal, arrondi correctement.

On peut facilement construire un exemple en bases mixtes pour lequel le résultat d'une opération est difficile à arrondir correctement dans le format de sortie. Il n'existe cependant pas de méthode simple et systématique permettant de détecter ces cas où l'arrondi est difficile, d'où le besoin d'une étude dédiée des opérations en bases mixtes et notamment du FMA. Ces travaux sont basés sur une précédente étude [107] tentant de calculer ces cas d'arrondi difficile sans pour autant y parvenir.

La norme IEEE 754 définit plusieurs formats binaire et décimaux à virgule flottante à des précision standards, tels que les binary32, binary64 et les decimal64 ou decimal128. Nous souhaitons fournir des opérations en bases mixtes, telles qu'une addition *binary64 plus decimal128 égal binary128* devienne possible. Pour implémenter toutes les opérations arithmétiques de base $+, -, \times, \div, \sqrt{}$ et le FMA avec toutes les combinaisons d'opérandes possibles de façon exhaustive, il faudrait écrire environ 500 fonctions différentes. Nous proposons d'automatiser cette étape en proposant un générateur de code, produisant des fonctions arithmétiques en bases mixtes correctement arrondies, reposant sur un seul FMA en bases mixtes correctement arrondi.

5.1.2 Notations

Pour commencer, nous adoptons une notation unifiée pour les précisions binaires et décimales de chaque format IEEE 754. Appelons k_n une telle précision décrivant aussi bien

la précision binaire k_2 que la précision décimale k_{10}. Les valeurs prises par ces précisions pour chaque format sont décrites dans le tableau 5.1. L'erreur relative équivalente au formats binaires est $u_2 = 2^{-k_2}$ et l'erreur relative décimale peut être estimée telle que $u_{10} = 1/2 \times 10^{-k_{10}+1}$.

	binary64	decimal64	binary128	decimal128
précision k_n	53	16	113	34
$e_{\min}$	-1022	-383	-16382	-6143
$e_{\max}$	1023	384	16383	6144

Tableau 5.1 – Description des formats IEEE 754 à virgule flottante utilisés

Nous supposons que les formats décimaux IEEE 754, tel que le format decimal64, sont encodés dans le Binary Integer Decimal (BID) format [106], c'est-à-dire que l'accès à la représentation binaire de la mantisse décimale est direct. Étant donné ces paramètres, les entrées et sortie de nos algorithmes en bases mixtes sont notées de la façon suivante :

- des entrées binaires tels que $2^E \cdot m$, avec un exposant binaire signé et borné $E \in \mathbb{Z}$, et une mantisse entière signée $m \in \mathbb{Z}$, satisfaisant $2^{k_2-1} \leq |m| \leq 2^{k_2} - 1$, où k_2 est la précision binaire,
- ou des entrées décimales tels que$10^F \cdot n$, avec un exposant binaire signé et borné $F \in \mathbb{Z}$, et une mantisse entière signée $n \in \mathbb{Z}$, satisfaisant $1 \leq |n| \leq 10^{k_{10}} - 1$, où k_{10} est la précision décimale.
- un résultat binaire $2^G \cdot p$, avec l'exposant $G \in \mathbb{Z}$ convenablement borné et une mantisse entière signée et bornée $p \in \mathbb{Z}$,
- ou un résultat décimal $10^H \cdot q$ avec l'exposant $H \in \mathbb{Z}$ convenablement borné et une mantisse entière signée et bornée $q \in \mathbb{Z}$.

La norme IEEE 754 définit la notion de "quantum" pour les opérations décimales [38]. Cependant, lorsque l'entrée d'une opération est un nombre binaire (nombre à virgule flottante ou entier), le quantum est défini de façon ponctuelle. Nous n'aurons donc pas de considérations de quantum dans nos algorithmes.

5.2 Généralités sur les opérations en bases mixtes

5.2.1 Opérations de bases à partir du FMA

Étant donné un FMA en bases mixtes correctement arrondi tel que, soient a, b, et c trois nombres à virgule flottante binaires ou décimaux, $\circ_{k_n}(a \times b + c)$, on peut facilement

calculer la multiplication et l'addition en bases mixtes correctement arrondies. Pour la multiplication, il suffit d'affecter l'opérande c à zéro, et pour l'addition/soustraction, d'affecter la variable a ou b à un.

Il est également possible de générer facilement les opérations correctement arrondies en bases mixtes de division et de racine carrée à partir d'un seul FMA en bases mixtes correctement arrondi. Il faut pour cela supposer que le FMA permet d'utiliser une précision légèrement plus grande : étant donné les précisions binaire k_2 et décimale k_{10}, nous notons f le point milieu, soit le point se trouvant à équidistance entre deux nombres à virgule flottante consécutifs. Cela signifie qu'on peut noter le point milieu pour un format binaire tel que $2^{k_2-1} \leq \left|\frac{1}{2}\cdot m\right| < 2^{k_2}$, et pour un format décimal tel que $1 \leq \left|\frac{1}{2}\cdot n\right| \leq 10^{k_{10}}-1$.

Afin de pouvoir réaliser une analyse générale combinant l'analyse des différents formats binaires et décimaux, nous souhaitons introduire une notation unique permettant d'harmoniser tous ces formats. Comme 10 est divisible par 2, il est possible de représenter le point milieu décimal avec un exposant de deux et un exposant de cinq. Nous choisissons de représenter les mantisses des formats binaires et décimaux en binaire, ce qui signifie que le point milieu binaire a une mantisse m' telle que $2^{k'_2-1} \leq \left|m'\right| < 2^{k'_2}$ et le point milieu décimal a une mantisse n' telle que $2^{k'_2-1} \leq |n'| < 2^{k'_2}$ avec $k'_2 = \max\left(k_2+1, \left\lceil \log_2\left(10^{k_{10}}-1\right)\right\rceil+1\right)$. Les valeurs prises par k'_2 selon les différents formats sont détaillées dans le tableau 5.2. Les intervalles des exposants sont adaptés comme expliqué ci-dessous.

	binary64, decimal64	binary128, decimal128
précision k'_2	55	114
précision k_b	64	128

Tableau 5.2 – Équivalence des précisions pour les variables internes

L'arrondi du résultat de la division en bases mixtes $\circ_{k_n}\left(\frac{x}{y}\right)$ de précision k_2, resp. k_{10}, peut être déterminé grâce à l'aide du point milieu le plus porche du résultat exact $\frac{x}{y}$ [108]. Ce point milieu f peut être représenté sous la forme d'un point à virgule flottante de précision légèrement plus grande k'_2. Ainsi, la décision de l'arrondi par rapport à ce point milieu revient à décider entre les trois cas ci-contre.

- $\frac{x}{y} < f \iff y\times f - x > 0$,
- $\frac{x}{y} > f \iff y\times f - x < 0$,
- ou $\frac{x}{y} = f \iff y\times f - x = 0$.

Cela revient à dire qu'identifier l'arrondi du résultat de la division $\frac{x}{y}$ et ainsi calculer l'arrondi correct de l'opération de division en bases mixtes dans un format de sortie de précision k_n est équivalent au calcul du FMA en bases mixtes $y\times f - x$, permettant l'utilisation

d'opérandes ayant une précision légèrement plus grande k_2' que le format de sortie et de comparer le résultat à zéro.

De manière similaire, le calcul d'une racine carrée en bases mixtes avec arrondi correct à une précision k_n revient à déterminer de quel côté du point milieu le plus proche f se trouve le résultat. Ainsi pour déterminer si $\sqrt{x} < f$, nous pouvons établir si $f \times f - x > 0$ à l'aide d'un FMA permettant d'utiliser des opérandes de précision k_2' légèrement plus grande.

- $\sqrt{x} < f \iff f \times f - x > 0$,
- $\sqrt{x} > f \iff f \times f - x < 0$,
- ou $\sqrt{x} = f \iff f \times f - x = 0$.

5.2.2 Bornes du format interne

Si on veut représenter nos entrées avec la nouvelle précision k_2', nous avons besoin d'adapter les formats binaires et décimaux et les bornes de leurs exposants.

Les formats binaires de précision initiale k_2 permettent de représenter tous les nombres flottants normalisés et dénormalisés dans un certain intervalle. Nous savons que la mantisse est normalisée telle que $2^{k_2'-1} \leq |m| < 2^{k_2'}$. La plus petite valeur est le point milieu entre le plus petit dénormalisé et zéro, c'est-à-dire $2^{E^{\min}-k_2}$. Ainsi on assure que $2^{E'} \cdot m \geq 2^{E^{\min}}$, et on peut définir la borne inférieure de l'exposant telle que $E' \geq E^{\min} - k_2 - k_2'$. La plus grande valeur est $2^{E^{\max}-(k_2-1)} \cdot \left(2^{k_2}-1\right)$. Ainsi on assure que $2^{E'} \cdot m < 2^{E^{\max}+1}$, et on peut définir la borne supérieure de l'exposant telle que $E' \leq E^{\max} - k_2' + 1$. On peut ainsi redéfinir les formats binaires de précision initiale k_2, en incluant les nombres dénormalisés, tels que, étant donné $m, E' \in \mathbb{Z}$, on a

$$\begin{aligned} &2^{E'} \cdot m, \qquad 2^{k_2'-1} \leq |m| < 2^{k_2'}, \\ &E^{\min} - k_2 - k_2' \leq E' \leq E^{\max} - k_2' + 1. \end{aligned} \tag{5.1}$$

Dans le cas des formats décimaux de précision k_{10}, nous cherchons à représenter le nombre $10^F \cdot n$ borné tel que

$$\begin{aligned} &10^{F^{\min}} \cdot 10^{1-j_{10}} \leq 10^F \cdot |n| < 10^{F^{\max}+1}, \\ &1 \leq |n| \leq 10^{k_{10}} - 1, \end{aligned} \tag{5.2}$$

pour obtenir une représentation en bases mixtes $2^J \cdot 5^K \cdot r$ bornée telle que

$$\begin{aligned} &10^{F^{\min}} \cdot 10^{1-j_{10}} \leq 2^J \cdot 5^K \cdot |r| < 10^{F^{\max}+1}, \\ &2^{k_2'-1} \leq |r| < 2^{k_2'}. \end{aligned} \tag{5.3}$$

Cela se fait en posant $K = F$, et en ajustant la mantisse $|n|$ par décalage afin de respecter les contraintes des bornes de la mantisse $|r|$. L'exposant binaire J est corrigé en conséquence.

Ainsi, $10^F \cdot |\, n \,| \;=\; 2^J \cdot 5^K \cdot |\, r \,|$ avec $K = F$ nous donne

$$2^F \cdot |\, n \,| \;=\; 2^J \cdot |\, r \,|. \tag{5.4}$$

Les bornes de la mantisse $|\, n \,|$ décrites dans l'équation (5.2) impliquent que $1 \;\leq\; |\, n \,| \;\leq\; 2^{\lceil \log_2(10^{k_{10}-1}) \rceil}$. Les bornes de la mantisse $|\, r \,|$ décrites dans l'équation (5.3) impliquent que $2^{\lceil \log_2(10^{k_{10}-1}) \rceil} \;\leq\; |\, r \,| \;\leq\; 2^{\lceil \log_2(10^{k_{10}-1}) \rceil + 1}$. On en déduit que $|\, n \,| \;\leq\; |\, r \,| \;\leq\; 2^{\lceil \log_2(10^{k_{10}-1}) \rceil + 1} \cdot |\, n \,|$. Ainsi, d'après l'équation (5.4), on a que

$$-\left\lceil \log_2\left(10^{k_{10}-1}\right) \right\rceil - 1 + F \;\leq\; J \;\leq\; F \tag{5.5}$$

Ainsi, si $F^{\min} \;\leq\; F \;\leq\; F^{\max}$, on a que $F^{\min} - \lceil k_{10} \cdot \log_2(10) \rceil - 1 \;\leq\; J \;\leq\; F^{\max}$, et $F^{\min} \;\leq\; K \;\leq\; F^{\max}$.

Les formats décimaux de précision initiale k_{10} se traduisent en base mixte tels que, étant donné $r, J, K \in \mathbb{Z}$, on a

$$\begin{aligned} 10^F \cdot n \;&=\; 2^J \cdot 5^K \cdot r, \qquad 2^{k_2'-1} \;\leq\; |\, r \,| \;<\; 2^{k_2'}, \\ F^{\min} - \lceil k_{10} \cdot \log_2(10) \rceil - 1 \;&\leq\; J \;\leq\; F^{\max}, \\ F^{\min} \;&\leq\; K \;\leq\; F^{\max}. \end{aligned} \tag{5.6}$$

5.2.3 Défis du FMA

Générer les opérations arithmétiques de base en bases mixtes ne repose cependant pas sur un seul FMA correctement arrondi en bases mixtes, mais sur plusieurs dépendant de la combinaison de formats donnés comme entrées et sorties. En effet, le FMA comprend trois opérandes en entrées, *a*, *b*, et *c*, et renvoie une sortie, chacune de ces quatre variables étant représentable en quatre formats différents : binary64, decimal64, binary128 ou decimal128. Nous pouvons naïvement considérer qu'il faut ainsi développer $4^4 \;=\; 256$ versions du FMA en bases mixtes. Toutefois, comme nous étudions l'arithmétique en bases mixtes, nous pouvons écarter les FMA ayant des opérandes uniquement binaires ou décimales. De plus, la multiplication étant commutative, nous pouvons considérer par exemple qu'une multiplication *decimal64-par-binary64* est équivalente à la multiplication *binary64-by-decimal64*, peu importe la sortie désirée. Ainsi, nous pouvons exclure $2^4 \;=\; 16$ versions supplémentaires du FMA. Il reste ainsi 236 version différentes du FMA à considérer pour l'implémentation d'une arithmétique en bases mixtes se basant sur les formats binary64, decimal64, binary128 et decimal128.

Nous n'avons cependant pas l'intention de développer chacune de ces 236 fonctions à la main, mais au contraire d'analyser l'algorithme du FMA en bases mixtes de façon à formaliser les variables clefs et factoriser les parties communes. De cette façon, nous

pourrons alors implémenter un générateur de code, renvoyant un FMA en fonction de sa borne d'erreur et le type de ses variables.

Le vrai défi des opérations en bases mixtes repose sur le calcul de l'addition ou de la soustraction à l'intérieur du FMA. Comme la soustraction de variables à virgules flottantes est simplement une addition dont le signe du terme soustractif a été modifié, nous allons seulement continuer cette discussion pour "l'addition", la soustraction étant implicite. De façon générale, pour l'addition de variables à virgules flottantes, on peut distinguer deux cas : d'un côté, lorsque les deux opérandes sont suffisamment proches en termes d'exposants, de sorte que la soustraction de leurs mantisses alignées mène à une élimination, et d'un autre côté, lorsque les opérandes sont suffisamment éloignées selon la magnitude de leurs exposants, de sorte que l'ordre de magnitude du résultat soit essentiellement celui de la plus grande opérande. Le premier cas est appelé couramment l'addition *near path*, tandis que le second est référé comme l'addition *far path* [36].

L'addition à virgule flottante correctement arrondie en base uniforme est basée, pour l'algorithme near path, sur l'observation que, lorsqu'une élimination se produit, l'opération à virgule flottante devient alors naturellement exacte, c'est-à-dire qu'aucun arrondi spécifique n'est nécessaire [36]. Ce cas correspond au cas d'application du lemme de Sterbenz [105], lemme 2. Pour l'addition en bases mixtes, rien n'indique que cette élimination peut devenir exacte : par exemple, lorsque les deux opérandes sont des variables décimales à virgule flottante, alors 10 et -9.9 sont représentables exactement. Leur somme cependant doit être égale à 0.1, qui n'est clairement pas représentable en format à virgule flottante binaire.

Nous ne pouvons donc pas pré-supposer qu'aucun arrondi n'est nécessaire lors d'une addition en bases mixtes de deux nombres décimaux renvoyant un nombre à virgule flottante binaire. De plus, lorsque les deux opérandes d'une addition en bases mixtes ne sont pas représentées dans la même base, une sorte de conversion de l'opérande binaire vers une variable décimale ou inversement sera nécessaire. Alors que l'élimination dans le cas near path permet à l'addition de deux opérandes de formats uniformes de devenir exacte, cette même élimination dans le cas de l'addition en bases mixtes amplifie cette erreur de conversion :

$$a + b \cdot (1+\varepsilon) \;=\; (a+b) \cdot \left(1 + \frac{b}{a+b} \cdot \varepsilon\right), \qquad |a+b| \;\ll\; |b|\,. \tag{5.7}$$

Quand bien même serions-nous capable de trouver une méthode pour calculer une borne inférieure sur le résultat de cette élimination catastrophique, c'est-à-dire trouver un moyen de rendre la conversion suffisamment précise de sorte que même si la élimination amplifie l'erreur, le résultat garde un certain nombre de bits corrects, le résultat (approché) de cette addition en bases mixtes peut lui-même se trouver près d'une borne d'arrondi dans la base de sortie. Ainsi l'arrondi correct du résultat sera tout de même difficile à obtenir.

Le cas de l'addition en bases mixtes far path n'est pas moins simple, une absorption pouvant survenir et mener aux mêmes problèmes d'arrondi. Même en construisant un exemple illustrant un cas d'arrondi difficile d'addition en bases mixtes, il n'existe pas de méthode simple et systématique permettant d'identifier les entrées engendrant un tel arrondi difficile pour l'arrondi au plus proche ou les arrondis dirigés. Nous proposons donc une approche pour développer un FMA en bases mixtes avec arrondi correct, sans pré-calculer cette borne inférieure.

5.3 Algorithme du FMA en bases mixtes

Les entrées de notre algorithme de FMA en bases mixtes sont soit des nombres à virgule flottante binaires ou décimaux, avec la possibilité de représenter les points milieux à une précision k_n donnée par une précision légèrement supérieure k_2' de la mantisse. Deux formats adapté ont été définis, un binaire dans l'équation (5.1) et un format décimal dans l'équation (5.6). Ces deux formats peuvent être harmonisés dans une même représentation, appelée format unifié en bases mixtes, avec des bornes pessimistes sur les exposants. Étant donné a une entrée du FMA en bases mixtes, sa représentation dans le format unifié en bases mixtes est donc, soient $t_a, N_a, P_a \in \mathbb{Z}$ et soient les bornes des exposants E', J et K détaillées dans les équation (5.6) :

$$\begin{aligned} a &= 2^{N_a} \cdot 5^{P_a} \cdot t_a, \qquad 2^{k_2'-1} \leq |\, t_a \,| < 2^{k_2'}, \\ \min\left(E^{\min\prime}, J^{\min}\right) &\leq N_a \leq \max\left(E^{\max\prime}, J^{\max}\right), \\ K^{\min} &\leq P_a \leq K^{\max}. \end{aligned} \tag{5.8}$$

Nous proposons de développer un FMA en bases mixtes, rapide et correctement arrondi, comme défini dans l'algorithme 5.1.

La première étape est la décomposition des entrées représentées en binary64, binary128, decimal64 et decimal128 dans le format unifié en bases mixtes, en définissant le signe, l'exposant et la mantisse de façon explicite. Tout au long de cet algorithme, nous analyserons les bornes de ces variables de façon à assurer le nombre de mots machines nécessaires pour représenter chacune d'elles. L'exposant est représenté par un entier signé de 32 bits, une taille suffisante pour représenter tous les exposants des formats considérés. La mantisse est stockée sur un ou plusieurs mots machine entiers de 64 bits.

Avec les entrées représentées dans ce format unifié en bases mixtes, il est possible de calculer la multiplication en bases mixtes de façon exacte. L'étape suivante consiste à choisir l'algorithme d'addition/soustraction à utiliser. Pour cela, nous effectuons un test, appelé le test de rapport, décrit plus en détail ci-après. Selon le résultat de ce test, le résultat de la multiplication en bases mixtes et l'entrée c sont convertis en format binaire, pour ensuite effectuer soit l'addition/soustraction *far path* ou bien la soustraction *near path*.

Algorithme 5.1 : FMA en bases mixtes avec arrondi correct

Données : a, b et c, nombres à virgule flottante binaires ou décimaux
Résultat : $R = 2^G \cdot p$ en binaire ou $R = 10^H \cdot q$ en décimal

$\psi \leftarrow a \times b$
si $\left| \frac{\psi}{c} \right| \notin \left[\frac{1}{2} ; 2 \right]$ **alors**
  $T_1 \leftarrow 2^{\gamma_1} \cdot \upsilon_1 = \psi \cdot \left(1 + \varepsilon_{T1_{\mathrm{FP}}}\right)$
  $T_2 \leftarrow 2^{\gamma_2} \cdot \upsilon_2 = c \cdot \left(1 + \varepsilon_{T2_{\mathrm{FP}}}\right)$
  $\phi \leftarrow (T_1 \pm T_2)\left(1 + \varepsilon_{\phi_{\mathrm{FP}}}\right)$ ▷ Addition/soustraction *"far-path"*
sinon
  $T_1 \leftarrow 2^{\Gamma_1} \cdot w_1 = \psi \cdot \left(1 + \varepsilon_{T1_{\mathrm{NP}}}\right)$
  $T_2 \leftarrow 2^{\Gamma_2} \cdot w_2 = c \cdot \left(1 + \varepsilon_{T2_{\mathrm{NP}}}\right)$
  $\phi \leftarrow (T_1 - T_2)\left(1 + \varepsilon_{\phi_{\mathrm{NP}}}\right)$ ▷ Soustraction *"near-path"*
fin
Définir $\rho = (a \times b + c)\left(1 + \varepsilon_\rho\right)$ tel que
$\rho \leftarrow 2^C \cdot g = \phi$ ▷ en binaire
$\rho \leftarrow 10^H \cdot 2^{-k_b + k_2' - 1} \cdot q = \phi \cdot \left(1 + \varepsilon_{\rho_D}\right)$ ▷ en décimal
si ρ *peut être correctement arrondi* **alors**
  $R \leftarrow \rho$
sinon
  Calculer la mantisse entière du point milieu le plus proche f tel que
  $f_2 : 2^C \cdot 2^{k_b - k_2' + 1} \cdot 1/2 \cdot f$ ▷ en binaire
  $f_{10} : 10^H \cdot 2^{-k_b + k_2' - 1} \cdot 2^{k_b - k_2' + 1} \cdot 1/2 \cdot f$ ▷ en décimal
  Calculer $\alpha \in \mathbb{Z}$ tel que
  $\alpha \leftarrow 2^{L - Z_{\min}} \cdot 5^{M - F_{\min}} \cdot (a \times b) + 2^{N_c - Z_{\min}} \cdot 5^{P_c - F_{\min}} \cdot c - 2^{F_T - Z_{\min}} \cdot 5^{F_F - F_{\min}} \cdot f$
  Corriger ρ en utilisant f et le signe de α
  $R \leftarrow \rho$
fin

Le résultat de cette addition/soustraction est ensuite testé pour déterminer si l'arrondi rapide de ce résultat est possible dans le format de sortie du FMA. Si le résultat est proche d'un point milieu f, décider de son arrondi vers une valeur à virgule flottante dans le format de sortie en fonction du mode d'arrondi choisi n'est pas possible [108]. Dans ce cas, on peut s'en sortir avec une étape de calculs supplémentaire, appelée la phase de récupération, permettant d'obtenir les quelques derniers bits de précision nécessaires pour décider de l'arrondi du résultat.

Dans la suite de ce chapitre, nous allons détailler point par point les mécanismes de l'algorithme de FMA en bases mixtes.

5.3.1 Multiplication et test de rapport

La première opération exécutée dans l'algorithme de FMA est assez simple : la multiplication peut être calculée exactement dans le format unifié en bases mixtes. Étant donné la définition des entrées a et b en format unifié en bases mixtes décrite dans l'équation (5.8), on peut calculer le résultat de la multiplication tel que, soient $s, L, M \in \mathbb{Z}$:

$$\begin{gathered} \psi \;=\; a \times b \;=\; 2^{L} \cdot 5^{M} \cdot s, \qquad 2^{2 \cdot k_2' - 1} \;\leq\; |\, s \,| \;<\; 2^{2 \cdot k_2'}, \\ \min\left(2 \cdot E^{\min\prime},\, E^{\min\prime} + J^{\min},\, 2 \cdot J^{\min}\right) - 1 \;\leq\; L \;\leq\; \max\left(2 \cdot E^{\max\prime},\, E^{\max\prime} + J^{\max},\, 2 \cdot J^{\max}\right), \\ 2 \cdot K^{\min} \;\leq\; M \;\leq\; 2 \cdot K^{\max}. \end{gathered} \tag{5.9}$$

Nous voulons ensuite effectuer l'addition/soustraction entre ψ et c, la troisième opérande donnée en entrée du FMA représentée dans le format décrit par l'équation (5.8). Nous avons détaillé précédemment dans la section 5.2.3 deux cas pour l'addition de deux variables à virgules flottantes, selon leur distance en termes d'exposants. Ces deux cas sont les algorithmes d'addition far path et near path.

Un test simple peut permettre de déterminer si les opérandes sont proches ou non en termes d'exposants pour identifier lequel des deux algorithmes d'addition utiliser. Si le rapport $\left| \frac{\psi}{c} \right|$ est clairement en dehors de l'intervalle $\left[\frac{1}{2} ; 2 \right]$, les opérandes sont suffisamment éloignées pour calculer l'addition avec l'algorithme le plus rapide : far path. Dans le cas contraire, l'algorithme plus précis near path est nécessaire. Ce test permet d'assurer que l'algorithme far path est utilisé seulement lorsque le rapport est en dehors de l'intervalle $\left[\frac{1}{2} ; 2 \right]$. Il se peut cependant que l'algorithme near path soit appelé sur des valeurs n'ayant pas besoin de précision supplémentaire, mais il sera forcément appelé pour les valeur qui en ont besoin.

Montrer que le rapport est assurément en dehors de l'intervalle $\left[\frac{1}{2} ; 2 \right]$ revient à montrer que, étant donné les définitions de ψ équation (5.9) et la définition de c équation (5.8), on a soit $2^{L-N_c} \cdot 5^{M-P_c} \cdot \frac{s}{t_c} \;<\; \frac{1}{2}$, soit $2^{L-N_c} \cdot 5^{M-P_c} \cdot \frac{s}{t_c} \;>\; 2$. Ceci revient à montrer que

$$\begin{gathered} (L - N_c) + \left\lfloor (M - P_c) \cdot \log_2(5) \right\rfloor + 1 + k_2' + 1 \;<\; -1, \\ \text{ou} \quad (L - N_c) + \left\lfloor (M - P_c) \cdot \log_2(5) \right\rfloor + k_2' - 1 \;>\; 1. \end{gathered} \tag{5.10}$$

Étant donné la valeur pré-calculée $\log_2(5)$, et soit $A \in \mathbb{Z}$ dans un petit domaine de définition, alors $\left\lfloor A \cdot \log_2(5) \right\rfloor$ peut être estimé correctement tel que $\left\lfloor A \cdot \left\lfloor \log_2(5) \cdot 2^{\xi} \right\rfloor \cdot 2^{-\xi} \right\rfloor$. Il existe une valeur minimum $\xi = \xi_{\min}$ pour laquelle cette estimation n'a pas de dépassement de capacité vers le haut (overflow) et peut être calculée exactement sur une variable entière signée de 64 bits [14].

Pour nous en convaincre, on peut appliquer cette méthode pour le calcul du pire cas d'élimination pour lequel toute précision est perdue (voir section 5.3.3.1 pour plus de

détails), considérant des entrées représentées comme des nombres à virgules flottantes décimaux ou binaires 128 bits. On a alors A borné tel que $-18430 \leq A \leq 18430$ et on peut estimer $\xi_{min} = 30$. Mis à part le calcul de $A \cdot \lfloor \log_2(5) \cdot 2^{\xi} \rfloor$ qui nécessite une variable entière signée d'au moins 47 bits, toutes les variables tiennent sur des entiers signés de 32 bits. Ainsi la multiplication de variables de 32 bits renvoie un résultat qui peut être exactement représenté sur un entier signé de 64 bits. Nous pouvons ainsi calculer les deux valeurs données dans l'équation (5.10) sur des entiers signés de 64 bits et effectuer le test de rapport qui permet de déterminer les deux cas de l'addition/soustraction.

5.3.2 Aucune élimination n'a lieu (far path)

Lorsque le test de rapport assure que le résultat est clairement en dehors de l'intervalle $\left[\, \frac{1}{2} \,;\, 2 \,\right]$, aucune élimination n'a lieu lors du calcul de l'addition/soustraction. On peut donc utiliser l'algorithme d'addition/soustraction *far path*. Il est plus simple de développer l'algorithme sur des variables à virgules flottantes binaire puis effectuer une analyse d'erreur plutôt que d'essayer d'additionner les variables dans le format unifié en bases mixtes. Le résultat binaire de l'addition/soustraction sera ensuite converti dans le format de sortie, cette conversion et l'arrondi final étant abordés dans la section 5.3.4.

5.3.2.1 Conversion en binaire pour l'addition/soustraction far path

Ainsi les opérandes ψ, resp. c, sont représentées dans le format unifié en bases mixtes avec des mantisses à la précision $2 \cdot k_2'$, resp. k_2', comme détaillé dans les équations (5.9), resp. (5.8). Nous souhaitons convertir ces opérandes dans un format binaire à la précision k_b. Cette étape correspond aux lignes 3 et 4 de l'algorithme 5.1.

Convertir ces opérandes revient à représenter la mantisse v stockée sur un entier non signé de k_b bits telle que $2^{k_b-1} \leq |v| \leq 2^{k_b} - 1$. Les valeurs prises par la précision k_b, par rapport aux différentes précisions k_n des formats binaires et décimaux, sont détaillées dans le tableau 5.2.

Les précisions des mantisses des opérandes ψ et c sont différentes, mais nous n'allons pas développer un algorithme de conversion du format unifié en bases mixtes vers le format binaire à la précision k_b pour chaque précision. L'idée est au contraire d'harmoniser en premier lieu les opérandes dans un format unifié en bases mixtes de précision k_b, telles que, soient $\omega, \lambda, \mu \in \mathbb{Z}$:

$$\begin{gathered} 2^{\lambda} \cdot 5^{\mu} \cdot \omega, \qquad 2^{k_b-1} \leq |\omega| \leq 2^{k_b} - 1, \\ \min\left(N^{\min} + N^{\text{shift}},\, L^{\min} + L^{\text{shift}}\right) \leq \lambda \leq \max\left(N^{\max} + N^{\text{shift}},\, L^{\max} + L^{\text{shift}}\right), \\ M^{\min} \leq \mu \leq M^{\max}, \end{gathered} \tag{5.11}$$

avec $N^{\text{shift}} = -k_b + k'_2$ et $L^{\text{shift}} = -k_b + 2 \cdot k'_2$. On peut ensuite appliquer l'algorithme de conversion du format unifie en bases mixtes à la précision k_b vers le format binaire à la précision k_b. Ainsi, soit une entrée conforme au format décrit par l'équation (5.11), nous calculons l'exposant γ et la mantisse v, tels que $\gamma = \lambda + \lfloor \mu \cdot \log_2(5) \rfloor + 1$ et $v = \lfloor 2^{-k_b} \cdot \omega \cdot \tau_0(\mu) \rfloor$, à l'aide de la table pré-calculée $\tau_0(\mu) = \left\lfloor 5^{\mu} \cdot 2^{-\lfloor \mu \cdot \log_2(5) \rfloor + k_b - 1} \right\rfloor$. Cette table a 1534 entrées pour un FMA en 64 bits, et 24574 pour un FMA de 128 bits. En appliquant cette formule, on remarque que la mantisse v est bornée telle que $2^{k_b-2} \leq |v| \leq 2^{k_b} - 1$. Lorsque $v < 2^{k_b-1}$, on effectue une correction sur cette mantisse afin d'assurer la contrainte de sortie $2^{k_b-1} \leq |v| \leq 2^{k_b} - 1$. L'exposant est adapté en conséquence. La mantisse v de la variable T est représentées par un entier non signé à la précision k_b, tel que, soient $v, \gamma \in \mathbb{Z}$:

$$\begin{aligned} &T = 2^{\gamma} \cdot v, \qquad 2^{k_b-1} \leq |v| \leq 2^{k_b} - 1, \\ &\lambda^{\min} + \lfloor \mu^{\min} \cdot \log_2(5) \rfloor - 1 \leq \gamma \leq \lambda^{\max} + \lfloor \mu^{\max} \cdot \log_2(5) \rfloor. \end{aligned} \tag{5.12}$$

5.3.2.2 Algorithme d'addition/soustraction far path

L'application de cet algorithme de conversion sur ψ et c nous donne les deux variables $T_1 = 2^{\gamma_1} \cdot v_1$ et $T_2 = 2^{\gamma_2} \cdot v_2$, définies comme dans l'équation (5.12). Ce sont les entrées de l'algorithme 5.2, l'addition/soustraction far path.

L'idée générale derrière cet algorithme est que lorsqu'il n'y a pas d'absorption, on aligne les deux variables puis on les additionne. Le résultat arrondi donne la mantisse binaire et on estime l'exposant associé.

Algorithme 5.2 : Addition/soustraction far path

Données : $T_1 = 2^{\gamma_1} \cdot v_1$ et $T_2 = 2^{\gamma_2} \cdot v_2$
Résultat : $\phi = (T_1 \pm T_2)\left(1 + \varepsilon_{\phi_{\text{FP}}}\right) = 2^C \cdot g$

1 Ordonner T_1 et T_2 de sorte que $\gamma'_1 \geq \gamma'_2$
2 **si** $\gamma'_1 \geq \gamma'_2 + k_b$ **alors**
3 $\quad g \leftarrow v'_1$ ▷ Absorption de T'_2
4 $\quad C \leftarrow \gamma'_1$
5 **sinon**
6 $\quad V_1 \leftarrow 2^{\gamma'_1 - \gamma'_2} \cdot v'_1$
7 $\quad V_2 \leftarrow v'_2$
8 $\quad V \leftarrow V_1 \pm V_2$
9 $\quad \sigma_V \leftarrow \texttt{lzc}(V)$
10 $\quad \pi \leftarrow 2^{\sigma_V} \cdot V$
11 $\quad g \leftarrow \lfloor 2^{-k_b} \cdot \pi \rfloor$ ▷ Mantisse g
12 $\quad C \leftarrow \gamma'_2 + k_b - \sigma_V$ ▷ Exposant C
13 **fin**

Lorsqu'aucune des deux opérandes n'est zéro, alors on peut ordonner T_1 et T_2 selon leurs exposants, de sorte que $\gamma'_1 \geq \gamma'_2$. Le cas de l'absorption revient à dire que, lorsque $\gamma'_1 \geq \gamma'_2 + k_b$, alors on a $T'_1 \geq 2^{\gamma'_2+2\cdot k_b-1}$ et $T'_2 \geq 2^{\gamma'_2+k_b-1}$ ou $-T'_2 \geq -2^{\gamma'_2+k_b}$. Cela signifie que le résultat de l'addition est borné par $T'_1 + T'_2 \geq 2^{\gamma'_2} \cdot \left(2^{2\cdot k_b-1} + 2^{k_b-1}\right)$ ou de la soustraction, par $T'_1 - T'_2 \geq 2^{\gamma'_2} \cdot \left(2^{2\cdot k_b-1} - 2^{k_b}\right)$. On voit que dans les deux cas, le résultat de l'addition/soustraction far path après absorption est $2^{\gamma'_1} \cdot \nu'_1$.

Dans le cas contraire, pour effectuer l'addition des deux mantisses entières ν'_1 et ν'_2 de précision k_b bits, on crée deux variables entières non signées V_1 et V_2 de taille $2 \cdot k_b$ bits. On décale la mantisse ν'_1 vers la gauche de sorte que $V_1 = 2^{\gamma'_1-\gamma'_2} \cdot \nu'_1$. On peut ainsi effectuer l'addition ou soustraction sur une variable entière non signée de taille $2 \cdot k_b$ bits, telle que $V = V_1 \pm V_2$. La fonction $\sigma_V = \texttt{lzc}(V)$ (leading zero count) permet de compter le nombre de zéros présents en tête des bits significatifs de la mantisse. Cette dernière est normalisée par ce nombre, puis tronquée à la précision k_b dans la variable entière non signée g. L'exposant C est calculé en conséquence, en fonction de σ_V.

La variable σ_V est bornée telle que $0 \leq \sigma_V \leq k_b + 2$, dépend du plus petit nombre de chiffres significatifs présents dans la variable V. Ces bornes dépendent de la borne d'erreur sur la soustraction far path, preuve exprimée en détail dans la section 5.3.2.4 suivante.

Le résultat de l'algorithme d'addition *far path* est donc renvoyé dans le format binaire suivant, soient $g, C \in \mathbb{Z}$:

$$\begin{aligned} &2^C \cdot g, \qquad 2^{k_b-1} \leq \left|\, g \,\right| \leq 2^{k_b} - 1, \\ &\gamma^{\min} + k_b - \sigma_V^{\max} \leq C \leq \gamma^{\max} + k_b. \end{aligned} \tag{5.13}$$

5.3.2.3 Analyse d'erreur : conversion en binaire pour l'addition/soustraction far path

Nous allons analyser l'erreur relative globale de l'algorithme d'addition/soustraction far path, depuis la conversion des entrées ψ et c jusqu'à l'obtention du résultat $\phi = 2^C \cdot g$. Nous noterons cette erreur globale $\varepsilon_{\phi_{\text{FP}}}$.

Lors de la normalisation des variables à une même précision en bases mixtes, comme décrit par l'équation (5.11), la précision de c augmente, ne produisant ainsi pas d'erreur. En revanche, la précision de la variable ψ décroît de $2 \cdot k'_2$ bits à k_b. L'erreur ε_ψ survient, telle que $2^{\lambda_\psi} \cdot 5^{\mu_\psi} \cdot \omega_\psi = 2^L \cdot 5^M \cdot s \cdot \left(1 + \varepsilon_\psi\right)$, et peut être bornée par

$$\left|\, \varepsilon_\psi \,\right| \leq 2^{L^{\text{shift}} - \left(2\cdot k'_2 - 1\right)} = 2^{-k_b+1}. \tag{5.14}$$

On rappelle que l'exposant γ satisfait $\gamma = \lambda + \lfloor \mu \cdot \log_2(5) \rfloor + 1$ et la mantisse ν se calcule comme $\nu = \left\lfloor 2^{-k_b} \cdot \omega \cdot \tau_0(\mu) \right\rfloor$. La table $\tau_0(\mu) = \left\lfloor 5^\mu \cdot 2^{-\lfloor \mu \cdot \log_2(5) \rfloor + k_b - 1} \right\rfloor$ est représentée par un tableau de variables entières non signées de k_b bits. Cela signifie que pour les valeurs k_b considérées, décrite dans le tableau 5.2, une valeur du tableau $\tau_0(\mu)$ est stockée soit sur un entier non signé de 64 bits, soit sur deux entiers non signés de 128 bits.

L'erreur relative se produisant lors du calcul de la table peut être bornée en remplaçant la partie entière inférieure dans l'exposant de 2 par la variable $\delta_{\exp\tau_0}$, avec $-1 < \delta_{\exp\tau_0} \leq 0$, telle que $\left|\varepsilon_{\tau_0(\mu)}\right| \leq \left(5^{\mu}\cdot 2^{-\mu\cdot\log_2(5)-\delta_{\exp\tau_0}+k_b-1}\right)^{-1}$. Cela veut dire que $1 \leq 2^{\delta_{\exp\tau_0}} < 2$, et donc $\left|\varepsilon_{\tau_0(\mu)}\right| \leq \left(5^{\mu}\cdot 5^{-\mu}2^{-\delta_{\exp\tau_0}+k_b-1}\right)^{-1}$. Au final, l'erreur relative associée au calcul de la table $\tau_0(\mu)$ est bornée telle que

$$\left|\varepsilon_{\tau_0(\mu)}\right| \leq 2^{-k_b+1}. \tag{5.15}$$

Enfin, la dernière erreur survenant lors de la conversion est liée au calcul de la mantisse binaire v. Cette erreur relative est la combinaison de l'erreur de troncature et de l'erreur associée au calcul de la table $\tau_0(\mu)$, de sorte que $\varepsilon_v = \left(2^{-k_b}\cdot\tau_0(\mu)\cdot\omega\right)^{-1}$, c'est-à-dire que $\varepsilon_v = \left(2^{-k_b}\cdot 2^{k_b-1}\cdot 2^{k_b-1}\right)^{-1}$. Elle est bornée telle que

$$|\varepsilon_v| \leq 2^{-k_b+2}. \tag{5.16}$$

De façon pessimiste, l'erreur relative globale $\varepsilon_{T_{i\mathrm{FP}}}$ survenant lors du calcul des deux variables binaires T_1 et T_2 de précision k_b bits est donc une combinaison de ces trois erreurs : l'erreur de conversion ε_ψ, l'erreur sur la table pré-calculée $\varepsilon_{\tau_0(\mu)}$ et l'erreur d'arrondi ε_v. On a donc que $2^{\gamma}\cdot v = 2^{\lambda}\cdot 5^{\mu}\cdot\omega\cdot\left(1+\varepsilon_{T_{i\mathrm{FP}}}\right)$ avec $\left(1+\varepsilon_{T_{i\mathrm{FP}}}\right) = (1+\varepsilon_v)\cdot\left(1+\varepsilon_{\tau_0(\mu)}\right)\cdot\left(1+\varepsilon_\psi\right)$. Une borne pessimiste sur l'erreur globale des entrées T_1 et T_2 de l'algorithme d'addition/-soustraction far path est donc

$$\left|\varepsilon_{T_{i\mathrm{FP}}}\right| \leq 2^{-k_b+3.5}. \tag{5.17}$$

5.3.2.4 Analyse d'erreur : addition/soustraction far path

Étant donné les entrées binaires T_1 et T_2, nous souhaitons borner l'erreur relative $\varepsilon_{\phi_{\mathrm{FP}}}$ telle que $\phi = (T_1 \pm T_2)\left(1+\varepsilon_{\phi_{\mathrm{FP}}}\right)$.

En premier lieu, l'algorithme teste si une absorption se produit. Dans ce cas, c'est-à-dire si $\gamma'_1 \geq \gamma'_2 + k_b$, la sortie est $\phi = 2^{\gamma'_1}\cdot v'_1\cdot\left(1+\varepsilon_{\phi_{\mathrm{FP}}}\right)$ avec

$$\left|\varepsilon_{\phi_{\mathrm{FP}}}\right| \leq 2^{-k_b+1}. \tag{5.18}$$

Dans le cas où il n'y a pas d'absorption, c'est-à-dire si $\gamma'_1 < \gamma'_2 + k_b$, alors puisque $\gamma'_1, \gamma'_2 \in \mathbb{Z}$, on a $\gamma'_2 \leq \gamma'_1 \leq \gamma'_2 + k_b - 1$, soit $0 \leq \gamma'_1 - \gamma'_2 \leq k_b - 1$. Ainsi, on peut déduire des bornes sur les variables V_1 et V_2 telles que définies dans l'algorithme 5.2, de sorte que $2^{k_b-1} \leq V_2 < 2^{k_b}$ et $2^{k_b-1} \leq V_1 < 2^{2\cdot k_b-1}$. Pour borner l'erreur relative $\varepsilon_{\phi_{\mathrm{FP}}}$, nous voulons borner la variable $V = V_1 \pm V_2$. Si c'est une addition, on peut borner V tel que $2^{k_b-1}+2^{k_b-1} \leq V_1+V_2 < 2^{2\cdot k_b-1}+2^{k_b}$, qui se simplifie de la façon suivante $2^{k_b} \leq V_1+V_2 < 2^{2\cdot k_b-1}\left(1+2^{-k_b+1}\right) < 2^{2\cdot k_b}$.

Si on effectue une soustraction, alors la borne supérieure de v est facile à déterminer, soit $V_1 - V_2 < 2^{2\cdot k_b - 1} - 2^{k_b} < 2^{2\cdot k_b}$. La borne inférieure est un peu plus complexe à établir. Remarquons d'abord que le résultat du test de rapport pour l'addition/soustraction far path est

$$2 < \frac{2^L \cdot 5^M \cdot s}{2^{N_c} \cdot 5^{P_c} \cdot t_c}. \tag{5.19}$$

Étant donné la définition des entrées T_1 et T_2 et de leurs bornes d'erreurs, $T_1 = 2^{\gamma_1} \cdot v_1 = 2^L \cdot 5^M \cdot s \cdot \left(1 + \varepsilon_{T_{1\text{FP}}}\right)$ and $T_2 = 2^{\gamma_2} \cdot v_2 = 2^{N_c} \cdot 5^{P_c} \cdot t_c \cdot \left(1 + \varepsilon_{T_{2\text{FP}}}\right)$, avec $\left| \varepsilon_{T_{i\text{FP}}} \right| \leq 2^{-k_b + 3.5}$, on peut en déduire que

$$2 < \frac{2^{\gamma_1 - \gamma_2} \cdot v_1 \cdot (1 + \varepsilon_{T_{2\text{FP}}})}{v_2 \cdot (1 + \varepsilon_{T_{1\text{FP}}})} \qquad \Rightarrow \qquad 2 \cdot \frac{1 + \varepsilon_{T_{1\text{FP}}}}{1 + \varepsilon_{T_{2\text{FP}}}} < \frac{V_1}{V_2}. \tag{5.20}$$

Pour parvenir à définir une borne inférieure pour le résultat de la soustraction, définissons le lemme suivant.

Lemme 3. *Soient $\varepsilon_1, \varepsilon_2 \in \mathbb{R}$ bornés tel que $|\varepsilon_1| \leq \bar{\varepsilon}$ et $|\varepsilon_2| \leq \bar{\varepsilon}$, avec $\bar{\varepsilon} \in \mathbb{R}^+$. Si $\bar{\varepsilon}$ est bornée par $\bar{\varepsilon} \leq \frac{1}{8}$, alors on a*

$$\left| \frac{1 + \varepsilon_1}{1 + \varepsilon_2} - 1 \right| \leq 2^{1.20} \cdot \bar{\varepsilon}. \tag{5.21}$$

Démonstration. Nous voulons borner $\left| \frac{1+\varepsilon_1}{1+\varepsilon_2} - 1 \right|$, en commençant d'abord par réécrire cette quantité de la façon suivante :

$$\begin{aligned}
\frac{1 + \varepsilon_1}{1 + \varepsilon_2} - 1 &= (1 + \varepsilon_1)\left(\frac{1}{1 + \varepsilon_2} - 1\right) + (1 + \varepsilon_1) - 1, \\
&= (1 + \varepsilon_1)\left(\frac{1}{1 + \varepsilon_2} - 1 + \varepsilon_2 - \varepsilon_2\right) + \varepsilon_1, \\
&= \varepsilon_1 - \varepsilon_2 (1 + \varepsilon_1) + (1 + \varepsilon_1)\left(\frac{1}{1 + \varepsilon_2} - 1 + \varepsilon_2\right), \\
&= \varepsilon_1 - \varepsilon_2 - \varepsilon_1 \varepsilon_2 + (1 + \varepsilon_1)\left(\frac{1 - 1 - \varepsilon_2 + \varepsilon_2 (1 + \varepsilon_2)}{1 + \varepsilon_2}\right), \\
\frac{1 + \varepsilon_1}{1 + \varepsilon_2} - 1 &= \varepsilon_1 - \varepsilon_2 - \varepsilon_1 \varepsilon_2 + \varepsilon_2^2 \left(\frac{1 + \varepsilon_1}{1 + \varepsilon_2}\right).
\end{aligned} \tag{5.22}$$

Ainsi, la valeur absolue est bornée telle que

$$\left| \frac{1 + \varepsilon_1}{1 + \varepsilon_2} - 1 \right| \leq |\varepsilon_1| + |\varepsilon_2| + |\varepsilon_1 \varepsilon_2| + \varepsilon_2^2 \cdot \left| \frac{1 + \varepsilon_1}{1 + \varepsilon_2} \right|. \tag{5.23}$$

Nous supposons que ces variables sont bornées telles que $|\varepsilon_1| \leq \bar{\varepsilon} \leq \frac{1}{8}$ et $|\varepsilon_2| \leq \bar{\varepsilon} \leq \frac{1}{8}$. On peut donc borner la quantité suivante :

$$\left| \frac{1 + \varepsilon_1}{1 + \varepsilon_2} \right| \leq \left| \frac{1 + \frac{1}{8}}{1 - \frac{1}{8}} \right| = \frac{9}{7}. \tag{5.24}$$

On peut donc utiliser cette borne et remplacer les variables de l'équation (5.23) telles que

$$\begin{aligned} \left| \frac{1+\varepsilon_1}{1+\varepsilon_2} - 1 \right| &\leq 2\bar{\varepsilon} + \bar{\varepsilon}^2 + \frac{9}{7}\bar{\varepsilon}^2 = \bar{\varepsilon} \cdot \left(2 + \bar{\varepsilon} + \frac{9}{7}\bar{\varepsilon}\right), \\ \left| \frac{1+\varepsilon_1}{1+\varepsilon_2} - 1 \right| &\leq \bar{\varepsilon} \cdot \left(2 + \frac{1}{8} + \frac{9}{7} \cdot \frac{1}{8}\right) = \frac{16}{7}\bar{\varepsilon}. \end{aligned} \tag{5.25}$$

Au final, on a bien montré que $\left| \frac{1+\varepsilon_1}{1+\varepsilon_2} - 1 \right| \leq 2^{1.20} \cdot \bar{\varepsilon}$. □

Dans le cas de la soustraction, les erreurs relatives sur les entrées T_1 et T_2 sont bornées telles que $\left| \varepsilon_{T_{i\text{FP}}} \right| \leq 2^{-k_b+3.5}$. Ainsi, d'après le lemme 3, étant donné $\bar{\varepsilon} = 2^{-k_b+3.5} \leq \frac{1}{8}$ pour toutes les valeurs de k_b, on a

$$\begin{aligned} \left| \frac{1+\varepsilon_{T_{1\text{FP}}}}{1+\varepsilon_{T_{2\text{FP}}}} - 1 \right| &\leq 2^{1.2} \cdot 2^{-k_b+3.5} = 2^{-k_b+4.7}, \\ 1 - 2^{-k_b+4.7} &\leq \frac{1+\varepsilon_{T_{1\text{FP}}}}{1+\varepsilon_{T_{2\text{FP}}}}. \end{aligned} \tag{5.26}$$

En injectant la borne décrite dans l'équation (5.26) dans l'équation (5.20), on trouve que les variables V_1 et V_2 sont bornées par $2 \cdot (1 - 2^{-k_b+4.7}) < \frac{V_1}{V_2}$. Étant donné que $V_1 - V_2 = \left(\frac{V_1}{V_2} - 1\right) V_2$, la borne inférieure de la soustraction est $2^{k_b-1} - 2^{4.7} < V_1 - V_2$.

On peut simplifier cette borne en remarquant que $2^{k_b-1} - 2^{4.7} = 2^{k_b-2} \cdot (2 - 2^{-k_b+6.7})$. Puisque $1 \leq 2 - 2^{-k_b+6.7}$ pour toutes les valeurs de k_b considérées, on peut borner $2^{k_b-1} - 2^{4.7}$ tel que $2^{k_b-2} < 2^{k_b-1} - 2^{4.7}$, et on a donc $2^{k_b-2} < V_1 - V_2$. Au final, les bornes de la soustraction far path sont $2^{k_b-2} < V_1 - V_2 < 2^{2 \cdot k_b}$.

Une fois le résultat de l'addition/soustraction borné, on peut facilement borner σ_V, représentant le nombre de zéros précédent les bits significatifs de $V = V_1 \pm V_2$. Cette valeur dépend donc de la borne minimale sur la taille de V, ainsi, on peut définir les bornes de σ_V telles que $0 \leq \sigma_V \leq 2 \cdot k_b - (k_b - 2) = k_b + 2$.

La dernière étape de l'algorithme consiste à tronquer le résultat V pour déduire la mantisse de la sortie $\phi = 2^C \cdot g$. Cette opération est le calcul de la variable π, telle que $\pi = 2^{\sigma_V} \cdot V$ bornée par $2^{2 \cdot k_b - 1} \leq \pi < 2^{2 \cdot k_b}$. Finalement, on a $g = \left\lfloor 2^{-k_b} \cdot \pi \right\rfloor = 2^{-k_b} \cdot \pi \cdot \left(1 + \varepsilon_{\phi_{\text{FP}}}\right)$. La borne d'erreur sur l'addition/soustraction far path est donc

$$\left| \varepsilon_{\phi_{\text{FP}}} \right| \leq 2^{-k_b+1}. \tag{5.27}$$

5.3.3 Une élimination a lieu (near path)

Lorsque le test de rapport établit que le résultat de la soustraction des deux entrées pourrait être à l'intérieur de l'intervalle $\left[\frac{1}{2} ; 2 \right]$, alors le calcul de cette soustraction peut être le sujet d'une élimination. On utilise donc l'algorithme de soustraction *near path*.

Une solution pour effectuer cet algorithme de soustraction near path sur des variables en bases mixtes est d'utiliser la même méthode que pour l'algorithme far path, et d'effectuer la soustraction sur des opérandes à virgules flottantes binaires. Pour se faire, nous devons déterminer la précision nécessaire des entrées binaires pour calculer la soustraction near path dans le pire cas d'élimination en évitant de perdre toute la précision de sortie.

5.3.3.1 Pire cas d'élimination

Le pire cas d'élimination se produit lorsque le résultat de la soustraction des entrées en bases mixtes ψ et c telle que $2^L \cdot 5^M \cdot s - 2^{N_c} \cdot 5^{P_c} \cdot t_c$ est relativement petite, sans être zéro. Ceci équivaut à dire que

$$\begin{aligned} 2^L \cdot 5^M \cdot s &\approx 2^{N_c} \cdot 5^{P_c} \cdot t, \\ \frac{s}{t} &\approx 2^{N_c - L} \cdot 5^{P_c - M}, \end{aligned} \tag{5.28}$$

avec les mantisses t et s bornées telles que $2^{k'_2 - 1} \leq |t| < 2^{k'_2}$ et $2^{2 \cdot k'_2 - 1} \leq |s| < 2^{2 \cdot k'_2}$.

Définissons deux variables X et Y, telles que $X = N_c - L$ et $Y = P_c - M$, $X, Y \in \mathbb{Z}$. Étant donné la définition de X, Y, ainsi que les bornes sur les exposants M, L, P_c et N_c donnés dans les équations (5.8) et (5.9), on peut définir des bornes sur X et Y. On se sert de ces valeurs pour définir le pire cas d'élimination.

Lemme 4. *Pour tous $X, Y \in \mathbb{Z}$ bornés tels que $N^{\min} - L^{\max} \leq X \leq N^{\max} - L^{\min}$ et $P^{\min} - M^{\max} \leq Y \leq P^{\max} - M^{\min}$, et étant donné $s, t \in \mathbb{Z}$ bornés tels que $2^{k'_2 - 1} \leq |t| < 2^{k'_2}$ et $2^{2 \cdot k'_2 - 1} \leq |s| < 2^{2 \cdot k'_2}$, le pire cas d'élimination est borné tel que*

$$\left| \frac{s}{t} - 2^X \cdot 5^Y \right| \geq \eta. \tag{5.29}$$

Démonstration. Application de l'algorithme décrit dans [14]. □

Les valeurs de la borne sur le pire cas d'élimination η sont détaillées dans le tableau 5.3. Elles ont été calculées à l'aide de l'outil Sollya [58] [1] en un temps raisonnable (environ 15 minutes pour un ensemble d'entrées et une sortie dans le format 64 bits, et 10 heures pour le format 128 bits).

5.3.3.2 Conversion en binaire pour l'addition/soustraction far path

On peut ainsi déduire le nombre de bits de précision des mantisses des opérandes binaires pour effectuer la soustraction near path avec au moins $k_b - 4$ bits de précision,

1. Le programme est disponible ici : `https://hal.archives-ouvertes.fr/hal-01893801`

	binary64, decimal64	binary128, decimal128
η	$2^{-177.61}$	$2^{-355.07}$
précision k_{NP}	256	448
bits de garde f_{NP}	18	32

Tableau 5.3 – Pire cas d'élimination et soustraction near path

précision équivalente à l'addition/soustraction far path qui est jugée suffisante pour effectuer l'arrondi final dans le format de sortie de précision k_n. Nous devons donc représenter ces mantisses avec une précision d'au moins $\lceil -\log_2(\eta) \rceil + k_b - 4$ bits.

Nous souhaitons représenter ces mantisses sur des mots machine de 64 bits. Ainsi cette représentation requiert plusieurs entiers non signés de 64 bits de façon à former des entiers non signés de k_{NP} bits, tel que $k_{\text{NP}} = 64 \cdot \left\lceil \frac{\lceil -\log_2(\eta) \rceil + k_b - 4}{64} \right\rceil$, dont les valeurs sont décrites dans le tableau 5.3. Cette représentation surestime le nombre de bits nécessaires pour effectuer la soustraction near path à la précision ciblée, de sorte qu'il reste f_{NP} bits "gratuits", tel que $f_{\text{NP}} = 64 - \left(\lceil -\log_2(\eta) \rceil + k_b - 4 - 64 \cdot (k_{\text{NP}} - 1) \right)$.

En pratique, cela signifie que dans le cas d'un FMA en format 64 bits, la mantisse des opérandes de la soustraction est représentée sur 256 bits, soit quatre entiers non signés de 64 bits, ce qui laisse 18 bits "gratuits" utilisés comme bits de garde. Dans le cas du format 128 bits, la mantisse est représentée sur 448 bits, soit sept mots de 64 bits, laissant 32 bits supplémentaires, utilisés comme bits de garde.

Comme pour l'algorithme far path, les opérandes ϕ et c sont converties du format unifié en bases mixtes vers un format binaire, cette fois de précision k_{NP} bits. On souhaite effectuer une seule conversion du format unifié en bases mixtes vers le format binaire. Pour cela, nous harmonisons les formats en bases mixtes, en représentant c comme une variable en bases mixtes de précision $2 \cdot k_2'$ bits. Cela revient à adapter l'exposant et la mantisse de c en augmentant sa précision, tels que, soient $L', M', s' \in \mathbb{Z}$ on a

$$\begin{aligned} 2^{L'} \cdot 5^{M'} \cdot s', \qquad 2^{2 \cdot k_2' - 1} &\leq \left| s' \right| \leq 2^{2 \cdot k_2'} - 1, \\ N^{\min} - k_2' &\leq L' \leq N^{\max} - k_2', \\ M^{\min} &\leq M' \leq M^{\max}. \end{aligned} \tag{5.30}$$

On peut ensuite appliquer la conversion sur les variables c et ψ maintenant représentées à la même précision. Cette conversion s'effectue à partir d'une entrée en format unifié en bases mixtes à la précision $2 \cdot k_2'$ vers le format binaire à la précision k_{NP}. Le nouvel exposant est $\Gamma = L' + \lfloor M' \log_2(5) \rfloor + 2k_2' + 1 - k_{\text{NP}} - \sigma_\Gamma$, avec $\sigma_\Gamma \in \mathbb{N}$ la constante de normalisation de la mantisse telle que $0 \leq \sigma_\Gamma \leq 1$. La nouvelle mantisse binaire est $w = \left\lfloor 2^{-2 \cdot k_2'} \cdot \tau_{0-3}(M') \cdot s \right\rfloor \cdot 2^{\sigma_\Gamma}$, étant donné la table $\tau_{0-3}(M) = \left\lfloor 5^{M'} \cdot 2^{k_{\text{NP}} - 1 - \lfloor M' \cdot \log_2(5) \rfloor} \right\rfloor$.

Cette variable binaire est bornée telle que

$$T = 2^{\Gamma} \cdot w, \qquad 2^{k_{\mathrm{NP}}-1} \leq |w| \leq 2^{k_{\mathrm{NP}}} - 1,$$
$$L^{\min\prime} + \left\lfloor M^{\min} \log_2(5) \right\rfloor + 2k_2' + 1 - k_{\mathrm{NP}} \leq \Gamma \leq L^{\max\prime} + \left\lfloor M^{\max} \log_2(5) \right\rfloor + 2k_2' + 2 - k_{\mathrm{NP}}. \tag{5.31}$$

5.3.3.3 Algorithme de soustraction near path

L'application de cet algorithme de conversion sur ψ et c nous donne les deux variables $T_1 = 2^{\Gamma_1} \cdot w_1$ et $T_2 = 2^{\Gamma_2} \cdot w_2$, définies comme dans l'équation (5.31). Ce sont les entrées de l'algorithme 5.3 de soustraction near path.

Algorithme 5.3 : Addition/soustraction near path

Données : $T_i = 2^{\Gamma} \cdot w = \left(2^{L'} \cdot 5^{M'} \cdot s'\right)_i \cdot \left(1 + \varepsilon_{T_{i\mathrm{NP}}}\right)$, avec $i = 2$

Résultat : $\phi = \left(\left(2^{L'} \cdot 5^{M'} \cdot s'\right)_1 - \left(2^{L'} \cdot 5^{M'} \cdot s'\right)_2\right)\left(1 + \varepsilon_{\phi_{\mathrm{NP}}}\right) = 2^{C} \cdot g$

1 Ordonner T_1 et T_2 de sorte que $\Gamma_1' \geq \Gamma_2'$
2 $z_1 \leftarrow w_1'$
3 $z_2 \leftarrow \left\lfloor 2^{\Gamma_2' - \Gamma_1'} \cdot w_2' \right\rfloor$ ▷ Alignement des opérandes
4 Ordonner z_1 et z_2 de sorte que $z_1' \geq z_2'$
5 $d \leftarrow z_1' - z_2'$
6 $\sigma_d \leftarrow \mathtt{lzc}(d)$
7 $g \leftarrow \left\lfloor d \cdot 2^{\sigma_d} \cdot 2^{(k_{\mathrm{NP}} - k_b)} \right\rfloor$ ▷ Mantisse g
8 $C \leftarrow \Gamma_1' - \sigma_d + (k_{\mathrm{NP}} - k_b)$ ▷ Exposant C

L'algorithme de soustraction near path est similaire à celui de l'addition/soustraction far path. Comme ce dernier, la première étape de l'algorithme de soustraction near path est d'ordonner les opérandes telles que $\Gamma_1' > \Gamma_2'$. Il s'en distingue cependant en alignant les opérandes non pas par rapport à la plus petite des deux opérandes en valeur absolue, mais par rapport à la plus grande, de sorte que $z_2 = \left\lfloor 2^{\Gamma_2' - \Gamma_1'} \cdot w_2' \right\rfloor$. Cette opération produit une erreur, mais, étant donné que la taille des opérandes dépend de la borne sur le pire cas d'élimination, la précision de sortie de l'algorithme near path reste acceptable pour le reste de l'algorithme de FMA, comme nous le verrons dans la section 5.3.3.5.

Les variables z_1 et z_2 sont ensuite ordonnées telles que $z_1' > z_2'$, et on effectue la soustraction $d = z_1' - z_2'$. Comme dans l'algorithme far path, le résultat final dépendra sur nombre de chiffres significatifs de d, estimé grâce à $\sigma_d = \mathtt{lzc}(d)$, le nombre de zéros en tête des chiffres significatifs de d. Ainsi la mantisse est normalisée avec σ_d et rétablie dans la précision de sortie telle que $g = \left\lfloor d \cdot 2^{\sigma_d} \cdot 2^{(k_{\mathrm{NP}} - k_b)} \right\rfloor$ et l'exposant est ajusté en conséquence tel que $C = \Gamma_1' - \sigma_d + (k_{\mathrm{NP}} - k_b)$.

Le résultat de la soustraction *near path* prend une forme similaire au résultat de l'addition/soustraction *far path* défini dans l'équation (5.13), à l'exception des bornes de l'exposant C, définies telles que, soient $g, C \in \mathbb{Z}$ on a

$$\begin{gathered} 2^C \cdot g, \qquad 2^{k_b-1} \;\leq\; \left| g \right| \;\leq\; 2^{k_b}-1, \\ \left\lfloor \log_2\left(2^{N^{\min}} 5^{P^{\min}} t^{\min} \frac{\eta\left(1-\varepsilon_{\phi_{\mathrm{NP}}}\right)}{g^{\max}}\right)\right\rfloor \;\leq\; C \;\leq\; \left\lfloor \log_2\left(\max\left(\left|c^{\max}\right|, \left|\psi^{\max}\right|\right) \cdot \frac{\left(1+\varepsilon_{\phi_{\mathrm{NP}}}\right)}{g^{\min}}\right)\right\rfloor + 1. \end{gathered} \tag{5.32}$$

5.3.3.4 Analyse d'erreur : conversion en binaire pour la soustraction near path

La première erreur se produisant dans le cas de la soustraction near path est due à la conversion des opérandes du format unifié en bases mixtes vers le format binaire en précision k_{NP} bits. Une seule conversion est effectuée pour les deux variables ψ et c, leurs précisions sont donc d'abord unifiées. On augmente la précision de l'opérande c à la précision de ψ, ce qui ne produit aucune erreur.

On rappelle que la mantisse est $w \;=\; \left\lfloor 2^{-2\cdot k_2'} \cdot \tau_{0-3}\left(M'\right) \cdot s \right\rfloor \cdot 2^{\sigma_\Gamma}$, étant donné la table $\tau_{0-3}\left(M'\right) \;=\; \left\lfloor 5^{M'} \cdot 2^{k_{\mathrm{NP}-1}-\lfloor M' \cdot \log_2(5) \rfloor} \right\rfloor$. Cette table est la une extension de la table $\tau_0(\mu)$ utilisée précédemment dans l'algorithme far path, étendue à une précision de k_{NP} bits. On peut montrer qu'on a exactement $\tau_0(M) \;=\; \left\lfloor 2^{k_b-k_{\mathrm{NP}}} \cdot \tau_{0-3}(M) \right\rfloor$. Ce résultat nous permet de calculer et de stocker la table $\tau_{0-3}(M)$ seulement, et l'adapter pour les calculs utilisant la table $\tau_0(\mu)$. La table $\tau_{0-3}(M)$ peut être représentée par quatre k_b variables entières non signées, chacune sur un ou deux mots de 64-bits. On peut borner l'erreur relative découlant du calcul de cette table de sorte que

$$\left| \varepsilon_{\tau_{0-3}} \right| \;\leq\; 2^{-k_{\mathrm{NP}}+1}. \tag{5.33}$$

L'erreur relative globale de la conversion est notée $\varepsilon_{T_{i\mathrm{NP}}}$ et définie telle que $T_i \;=\; 2^\Gamma \cdot w \;=\; 2^{L'} \cdot 5^{M'} \cdot s \cdot \left(1+\varepsilon_{T_{i\mathrm{NP}}}\right)$. Elle dépend de l'erreur relative de la table $\varepsilon_{\tau_{0-3}}$, ainsi que de l'erreur ε_w produite lors du calcul de la mantisse w. Cette erreur survient lors de la multiplication de la mantisse s' exacte et de la table $\tau_{0-3}(M)$, elle est bornée par $|\,\varepsilon_w\,| \;\leq\; \left(2^{-2\cdot k_2'} \cdot 2^{k_{\mathrm{NP}}-1} \cdot 2^{2\cdot k_2'-1} \cdot 1\right)^{-1}$, c'est-à-dire

$$|\,\varepsilon_w\,| \;\leq\; 2^{-k_{\mathrm{NP}}+2}. \tag{5.34}$$

Au final, l'erreur relative de la conversion en binaire pour l'algorithme near path est donc bornée par $\left(1+2^{-k_{\mathrm{NP}}+1}\right) \cdot \left(1+2^{-k_{\mathrm{NP}}+2}\right) - 1$, soit

$$\left| \varepsilon_{T_{i\mathrm{NP}}} \right| \;\leq\; 2^{-k_{\mathrm{NP}}+3}. \tag{5.35}$$

5.3.3.5 Analyse d'erreur : soustraction near path

Le calcul du résultat ϕ de la soustraction far path est sujet à des erreurs ; l'erreur globale étant définie telle que $\phi = (T_1 - T_2)(1+\varepsilon_{\phi_{\text{NP}}})$. On cherche donc à déterminer une borne sur l'erreur relative globale de la soustraction near path telle que, si $2^L5^M s - 2^{N_c}5^{P_c}t_c \neq 0$:

$$\begin{aligned} 2^C \cdot g &= \left(2^L5^M s - 2^{N_c}5^{P_c}t_c\right)\cdot\left(1+\varepsilon_{\phi_{\text{NP}}}\right), \\ 2^C \cdot g &= \left(2^L5^M s\cdot(1+\varepsilon_{V_{1\text{NP}}}) - 2^{N_c}5^{P_c}t_c\cdot(1+\varepsilon_{V_{2\text{NP}}})\right)\cdot\left(1+\varepsilon_g\right), \end{aligned} \tag{5.36}$$

avec

$$\left(1+\varepsilon_{\phi_{\text{NP}}}\right) = \left(1 + \frac{2^L5^M s}{2^L5^M s - 2^{N_c}5^{P_c}t_c}\varepsilon_{V_{1\text{NP}}} - \frac{2^{N_c}5^{P_c}t_c}{2^L5^M s - 2^{N_c}5^{P_c}t_c}\varepsilon_{V_{2\text{NP}}}\right)\left(1+\varepsilon_g\right), \tag{5.37}$$

où ε_g est l'erreur produite lors du calcul de la mantisse g et $\varepsilon_{V_{i\text{NP}}}$ sont les erreurs globales se produisant lors des transformations des opérandes. Cette dernière erreur est donc une combinaison de l'erreur de conversion, et l'erreur d'alignement des opérandes, telle que $\varepsilon_{V_{i\text{NP}}} = (1+\varepsilon_{T_{i\text{NP}}})\cdot(1+\varepsilon_{z_2}) - 1$.

La première erreur se produisant dans l'algorithme near path a lieu lors de la normalisation de la plus petite des deux opérandes en valeur absolue et de son alignement avec l'autre opérande, telle que $z_2 = \left(2^{\Gamma'_2-\Gamma'_1}\cdot w'_2\right)\cdot\left(1+\varepsilon_{z_2}\right)$. Comme nous sommes dans le cas de la soustraction near path, le test de rapport assure que

$$\frac{1}{2} \leq \frac{2^L5^M s}{2^{N_c}5^{P_c}t_c} \leq 2 \quad \Rightarrow \quad \frac{1}{2}\cdot\frac{1+\varepsilon_{T_{1\text{NP}}}}{1+\varepsilon_{T_{2\text{NP}}}}\cdot\frac{w_2}{w_1} \leq 2^{\Gamma_1-\Gamma_2} \leq 2\cdot\frac{1+\varepsilon_{T_{1\text{NP}}}}{1+\varepsilon_{T_{2\text{NP}}}}\cdot\frac{w_2}{w_1}. \tag{5.38}$$

D'après la définition de la borne d'erreur de $\varepsilon_{T_{i\text{NP}}}$ donnée dans l'équation (5.35), on a que $\frac{1}{2} < \frac{1+\varepsilon_{T_{1\text{NP}}}}{1+\varepsilon_{T_{2\text{NP}}}} < 2$, et d'après la définitions des bornes de la mantisse w détaillée équation (5.31), on a que $\frac{1}{2} < \frac{w_2}{w_1} < 2$. On peut ainsi affiner les bornes de l'équation (5.38) telles que $\frac{1}{8} < 2^{\Gamma_1-\Gamma_2} < 8$, c'est-à-dire que $0 \leq |\Gamma_1-\Gamma_2| \leq \lfloor\log_2(8)\rfloor - 1 = 2$. Comment Γ_1 et Γ'_2 sont ordonnés tels que $\Gamma'_1 \geq \Gamma'_2$, on a que $-2 \leq \Gamma'_2-\Gamma'_1 \leq 0$. D'après les bornes de la mantisse w, qui sont $2^{k_{\text{NP}}-1} \leq |w| < 2^{k_{\text{NP}}}$, on peut borner z_2 tel que $2^{k_{\text{NP}}-3} \leq 2^{\Gamma'_2-\Gamma'_1}\cdot w'_2$. Ainsi, la borne d'erreur sur la normalisation est

$$\left|\varepsilon_{z_2}\right| \leq 2^{-k_{\text{NP}}+3}. \tag{5.39}$$

On combine cette borne d'erreur avec l'erreur de conversion du format en bases mixtes vers le format binaire pour intégrer toutes les erreurs ayant lieu jusqu'à ce point. L'erreur relative combinée, notée $\varepsilon_{V_{i\text{NP}}}$ définie plus haut, est bornée telle que

$$\left|\varepsilon_{V_{i\text{NP}}}\right| \leq 2^{-k_{\text{NP}}+5}. \tag{5.40}$$

D'après l'équation (5.29) du lemme 4, si on a $2^L5^M s \neq 0$ et $2^L5^M s - 2^{N_c}5^{P_c}t_c \neq 0$, alors on peut écrire que

$$\left|\frac{2^L5^M s - 2^{N_c}5^{P_c}t_c}{2^L5^M s}\right| \geq \eta, \quad \Rightarrow \quad \begin{cases} \left|\frac{2^L5^M s}{2^L5^M s - 2^{N_c}5^{P_c}t_c}\right| & \leq \frac{1}{\eta}, \\ \left|\frac{2^{N_c}5^{P_c}t_c}{2^L5^M s - 2^{N_c}5^{P_c}t_c}\right| & \leq \frac{1}{\eta} + 1. \end{cases} \tag{5.41}$$

En utilisant ces valeurs pour borner l'erreur $\varepsilon_{\phi_{\text{NP}}}$ comme décrite dans l'équation (5.37), on a que

$$\left|\,\varepsilon_{\phi_{\text{NP}}}\,\right| \leq \left(1 + \frac{1}{\eta}\cdot\varepsilon_{V_{1\text{NP}}} + \left(\frac{1}{\eta}+1\right)\cdot\varepsilon_{V_{2\text{NP}}}\right)\cdot\left(1+\varepsilon_g\right) - 1. \tag{5.42}$$

En remplaçant l'erreur combinée de conversion et d'alignement $\varepsilon_{V_{i\text{NP}}}$ par sa borne donnée équation (5.40), on peut donc dire que $\frac{1}{\eta}\cdot\varepsilon_{V_{1\text{NP}}} \leq \frac{2^{-k_{\text{NP}}+5}}{\eta}$. De plus en remarquant que, pour les valeurs de η considérées, on a $\frac{1}{\eta}+1 \leq \frac{1}{\eta}\cdot 2^{0.01}$, on peut dire de la même façon que $\left(\frac{1}{\eta}+1\right)\cdot\varepsilon_{V_{2\text{NP}}} \leq \frac{2^{-k_{\text{NP}}+5}\cdot 2^{0.01}}{\eta}$. Ceci signifie que, pour toutes les valeurs de η, k_{NP} et k_b considérées, on obtient

$$\left(1 + \frac{1}{\eta}\cdot\varepsilon_{V_{1\text{NP}}} + \left(\frac{1}{\eta}+1\right)\cdot\varepsilon_{V_{2\text{NP}}}\right) - 1 \leq \frac{2^{-2k_{\text{NP}}+10.01}}{\eta} \leq 2^{-k_b-8}. \tag{5.43}$$

L'erreur relative sur le calcul de la mantisse g vient de l'arrondi du résultat intermédiaire plus précis à la précision k_b de sortie de g, de sorte que $g = \left(d\cdot 2^{\sigma_d}\cdot 2^{k_b-k_{\text{NP}}}\right)\cdot\left(1+\varepsilon_g\right)$, avec $2^{k_{\text{NP}}-1} \leq d\cdot 2^{\sigma_d} \leq 2^{k_{\text{NP}}}-1$. Ainsi la borne sur l'erreur ε_g est définie telle que

$$\left|\,\varepsilon_g\,\right| \leq \left(2^{k_{\text{NP}}-1}\cdot 2^{k_b-k_{\text{NP}}}\right)^{-1} = 2^{-k_b+1}. \tag{5.44}$$

Au final, on peut borner l'erreur relative globale de l'algorithme near path, dont la borne est formellement définie dans l'équation (5.42), telle que, pour toutes les valeurs de η, k_{NP} et k_b considérées, on a

$$\left|\,\varepsilon_{\phi_{\text{NP}}}\,\right| \leq \left(1+2^{-k_b-8}\right)\cdot\left(1+2^{-k_b+1}\right) - 1 \leq 2^{-k_b+1.01}. \tag{5.45}$$

5.3.4 Test d'arrondi et phase de récupération

Une fois les opérations de multiplication et d'addition ou soustraction effectuées grâce aux algorithmes rapides mais inexactes, on peut tenter de convertir le résultat dans la base de sortie à la précision k_n. On effectue ensuite un test d'arrondi pour vérifier si le résultat obtenu se trouve bien du bon côté de la frontière d'arrondi f. Si ce n'est pas le cas, on effectue une phase de récupération pour corriger l'arrondi du résultat.

5.3.4.1 Conversion dans la base de sortie

Le résultat des opérations précédentes est un nombre à virgule flottante en format binaire à la précision k_b bits tel que

$$\begin{aligned} &2^C \cdot g, \qquad 2^{k_b-1} \;\leq\; \left|\, g \,\right| \;\leq\; 2^{k_b}-1, \\ &\min\left(C_{\mathrm{FP}}^{\min}, C_{\mathrm{NP}}^{\min}\right) \;\leq\; C \;\leq\; \max\left(C_{\mathrm{FP}}^{\max}, C_{\mathrm{NP}}^{\max}\right). \end{aligned} \tag{5.46}$$

Si le format de sortie souhaité est un format binaire, on peut passer directement à l'étape suivante du test d'arrondi.

Sinon, le format de sortie souhaité est décimal de précision k_{10}, nous devons convertir la variable binaire avant d'effectuer le test d'arrondi telle que, soient $H, q \in \mathbb{Z}$ on a

$$\begin{aligned} &10^H \cdot 2^{-k_b+k_2'-1} \cdot q, \qquad 10^{k_{10}-1} \cdot 2^{k_b-k_2'+1} \;\leq\; q \;\leq\; \left(10^{k_{10}}-1\right) \cdot 2^{k_b-k_2'+1} \;<\; 2^{k_b}-1, \\ &\left\lfloor \log_{10}(2)\left(C^{\min}+k_b-1\right)\right\rfloor - k_{10}+1 \;\leq\; H \;\leq\; \left\lfloor \log_{10}(2)\left(C^{\max}+k_b-1\right)\right\rfloor - k_{10}+2. \end{aligned} \tag{5.47}$$

La mantisse est bornée par le produit d'une quantité décimale et une autre binaire. Cette représentation permet de distinguer la partie décimale à la précision k_{10} qui est la précision du format de sortie du FMA, et les bits de précision supplémentaire $k_b - k_2' + 1$ pour effectuer l'arrondi à la précision finale, où k_2' est la précision nécessaire pour représenter les points milieu des formats binaires et décimaux, et k_b est la précision binaire légèrement plus grande choisie pour effectuer les calculs d'addition et soustraction. Cela permet de fournir en entrée du test d'arrondi soit une variable binaire de précision k_b ou une variable décimale de précision similaire, tout en facilitant les calculs suivants.

On calcule d'abord l'exposant $H = \left\lfloor \log_{10}\left(2^C \cdot g\right)\right\rfloor - k_{10} + 1$. On peut effectuer ce calcul de sorte que, soit $\mu(C,g) \in \mathbb{N}$ un entier borné, on a

$$\begin{aligned} H &= \left\lfloor (C+k_b-1)\cdot \log_{10}(2)\right\rfloor - k_{10} + 1 + \mu(C,g), \\ \mu(C,g) &= \left\lfloor C\cdot\log_{10}(2) + \log_{10}(g) - \left\lfloor (C+k_b-1)\cdot\log_{10}(2)\right\rfloor\right\rfloor. \end{aligned} \tag{5.48}$$

Ceci nous permet d'utiliser la même méthode de calcul pour estimer $\left\lfloor (C+k_b-1)\cdot\log_{10}(2)\right\rfloor$ que dans la section 5.3.1. Dans ce cas, on peut calculer cette quantité exactement telle que $\left\lfloor A \cdot \left\lfloor \log_{10}(2)\cdot 2^{\xi}\right\rfloor \cdot 2^{-\xi}\right\rfloor$, avec A borné et une valeur minimum $\xi = \xi_{\min}$ pour laquelle on n'a pas de dépassement de capacité. Dans le cas des formats de 128 bits, on estime, grâce à la même méthode que celle utilisée dans la section 5.3.1, qu'une variable entière signée d'au moins 49 bits est nécessaire pour calcuer exactement $A \cdot \left\lfloor \log_2(5)\cdot 2^{\xi}\right\rfloor$. Les calculs sont donc effectués sur des entiers signés de 64 bits, et le résultat exact est renvoyé sur un entier signé de 32 bits.

On souhaite ensuite déterminer $\mu(C,g)$. Commençons d'abord par le borner, en remarquant que $\left\lfloor (C+k_b-1)\cdot\log_{10}(2)\right\rfloor = C\cdot\log_{10}(2) + (k_b-1)\cdot\log_{10}(2) + \delta$ avec δ borné

tel que $-1 < \delta \leq 0$. Ainsi, on peut écrire que $\mu(C,g) = \lfloor \log_{10}(g) - (\log_{10}(2^{k_b-1}) + \delta) \rfloor$. D'après la définition des bornes de g, on sait que $\log_{10}(2^{k_b-1}) \leq \log_{10}(g)$, c'est-à-dire que $\log_{10}(2^{k_b-1}) \leq \log_{10}(g)+1$, et on peut donc déterminer la borne inférieure telle que $0 \leq \lfloor \log_{10}(g)+1-\log_{10}(2^{k_b-1}) \rfloor$.

Pour la borne supérieure, on peut remarquer, de façon similaire en observant les bornes de g que $\log_{10}(g)-\log_{10}(2^{k_b-1}) \leq \log_{10}(2^{k_b}-1)-\log_{10}(2^{k_b-1}) = \log_{10}\left(2-\frac{1}{2^{k_b-1}}\right)$. Comme $1 < 2-\frac{1}{2^{k_b-1}} < 2$, alors $0 \leq \log_{10}\left(2-\frac{1}{2^{k_b-1}}\right) \leq 1$. Ainsi, la borne supérieure est $\left\lfloor \log_{10}\left(2-\frac{1}{2^{k_b-1}}\right)+1 \right\rfloor \leq 1$.

Au final, $0 \leq \mu(C,g) \leq 1$. Ceci implique que $\mu(C,g) \in \{0,1\}$. Pour déterminer $\mu(C,g)$, on peut observer qu'il s'agit en réalité d'une fonction croissante en g, c'est-à-dire qu'étant donné C, il existe un $g^*(C) \in \mathbb{Z}$ tel que $2^{k_b-1} \leq g^*(C) < 2^{k_b}$, pour lequel on a

$$\mu(C,g) = \begin{cases} 0 & \text{si } g \leq g^*(C), \\ 1 & \text{si } g > g^*(C). \end{cases} \tag{5.49}$$

Un tel $g^*(C)$ est pré-calculé comme un tableau d'entiers non signés de k_b bits, en satisfaisant l'équation $g^*(C) = \left\lceil 10^{\lfloor (C+k_b-1)\cdot\log_{10}(2)\rfloor - C\cdot\log_{10}(2)+1} \right\rceil$. On peut donc calculer l'exposant H avec des estimations correctes de la partie entière du logarithme en base 10 et un test avec une variable pré-calculée présente dans un tableau.

Une fois l'exposant H déterminé, on souhaite calculer la mantisse $q' = 2^{-k_b+k'_2-1}\cdot q$. L'idée est d'approcher la mantisse q' telle que

$$10^H \cdot 2^{k_b-k'_2+1}\cdot q' = 2^C \cdot g \cdot (1+\varepsilon_{q'}). \tag{5.50}$$

Posons la table pré-calculée $t(H)$ telle que $t(H) = \left\lfloor 10^{-H}\cdot 2^{\lfloor H\cdot\log_2(10)\rfloor+2k_b-1} \right\rfloor$. Pour borner les valeurs de ce tableau, on peut remarquer que $10^{-H}\cdot 2^{\lfloor H\cdot\log_2(10)\rfloor+2k_b-1} = 10^{-H}\cdot 2^{H\cdot\log_2(10)+\delta+2k_b-1}$ avec $-1 < \delta \leq 0$, soit $2^{-1} < 2^{\delta} \leq 1$. Ainsi $t(H) = \lfloor 2^{\delta+2k_b-1} \rfloor$, ce qui implique que $2^{2k_b-2} \leq t(H) \leq 2^{2k_b-1}$. Les valeurs de cette table sont donc représentées dans un tableau par plusieurs entiers non signés de 64 bits de sorte à former des variables de $2k_b-1$ bits. On peut donc directement estimer l'erreur sur le calcul de cette table, telle que $t(H) = 10^{-H}\cdot 2^{\lfloor H\cdot\log_2(10)\rfloor+2k_b-1}\cdot(1+\varepsilon_{t(H)})$, avec

$$\left|\varepsilon_{t(H)}\right| \leq 2^{-2k_b+2}. \tag{5.51}$$

La seconde étape pour calculer la mantisse q' consiste à étudier le produit $t(H)\cdot g$. On remarque que $t(H)\cdot g = 10^{-H}2^C g\cdot(1+\varepsilon_{t(H)})\cdot 2^{\lfloor H\cdot\log_2(10)\rfloor+2k_b-C}$. Étudions l'exposant du

dernier membre de cette équation, que l'on note $\alpha(H,C) \in \mathbb{Z}$. On a donc que

$$\begin{aligned}
\alpha(H,C) &= \lfloor H \cdot \log_2(10) \rfloor + 2k_b - C, \\
&= \lfloor \left(\lfloor \log_{10}\left(2^C \cdot g\right) \rfloor - k_{10} + 1 \right) \cdot \log_2(10) \rfloor + 2k_b - C, \\
&= \left(\log_{10}\left(2^C \cdot g\right) + \delta_1 - k_{10} + 1 \right) \cdot \log_2(10) + \delta_2 + 2k_b - C, \\
&= C + log_2\left(g\right) + \delta_1 \cdot \log_2(10) + (-k_{10}+1) \cdot \log_2(10) + \delta_2 + 2k_b - C, \\
\alpha(H,C) &= log_2\left(g\right) + \delta_1 \cdot \log_2(10) + (-k_{10}+1) \cdot \log_2(10) + \delta_2 + 2k_b,
\end{aligned} \tag{5.52}$$

avec $-1 < \delta_1 \leq 0$ et $-1 < \delta_2 \leq 0$. Ainsi, en prenant en compte les bornes de g rappelées dans l'équation (5.46), on peut borner $\alpha(H,C)$ tel que

$$\begin{aligned}
3k_b - 3 - \log_2(10) + (-k_{10}+1) \cdot \log_2(10) &< \alpha(H,C) < 3k_b - 1 + (-k_{10}+1) \cdot \log_2(10), \\
\alpha_{\min} = \lceil 3k_b - 3 - k_{10} \cdot \log_2(10) \rceil &\leq \alpha(H,C) \leq \lfloor 3k_b - 1 + (-k_{10}+1) \cdot \log_2(10) \rfloor.
\end{aligned} \tag{5.53}$$

Pour évaluer $\alpha(H,C)$, on effectue la même opération que dans la section 5.3.1 pour déterminer le nombre de bits nécessaires au calcul de $\lfloor H \cdot \log_2(10) \rfloor$. On peut ensuite stocker cette valeur sur un entier signé de 32 bits.

On définit $\alpha'(H,C) \in \mathbb{Z}$ tel que $\alpha'(H,C) = \alpha(H,C) - \alpha_{\min}$. On se sert de cette quantité pour le calcul de la mantisse q' puis on développe ce résultat selon les définitions de $t(H) \cdot g$ et $\alpha'(H,C)$ de sorte que

$$\begin{aligned}
q' &= \left\lfloor 2^{-k_b+2} \left\lfloor 2^{-\alpha'(H,C)} \left\lfloor 2^{-k_b} \cdot t(H) \cdot g \right\rfloor_1 \right\rfloor_2 \right\rfloor_3, \\
q' &= \left(10^{-H} 2^C g \cdot \left(1 + \varepsilon_{t(H)}\right)\right) \left(2^{-k_b+2} 2^{-\alpha'(H,C)} 2^{-k_b} 2^{\alpha(H,C)}\right) (1+\varepsilon_1)(1+\varepsilon_2)(1+\varepsilon_3),
\end{aligned} \tag{5.54}$$

avec ε_i les erreurs associées aux parties entières numérotées dans l'équation du dessus.

On peut simplifier l'exposant $-k_b + 2 - \alpha'(H,C) - k_b + \alpha(H,C)$ en remplaçant $\alpha'(H,C)$ par $\alpha_{\min}$ de sorte que $-k_b + 2 - \alpha'(H,C) - k_b + \alpha(H,C) = -2k_b + 2 + \alpha_{\min}$. On sait que $\alpha_{\min} = \lceil 3k_b - 3 - k_{10} \cdot \log_2(10) \rceil = 3k_b - 3 + \lceil -k_{10} \cdot \log_2(10) \rceil$. Une définition de k'_2 est donnée dans la section 5.2.1, on peut montrer que $k'_2 - 2 = \lfloor \log_2\left(10^{k_{10}}\right) \rfloor$, ainsi $\lceil -k_{10} \cdot \log_2(10) \rceil = -k'_2 + 1$. Ceci implique que l'exposant se simplifie de nouveau, tel qu'au final, on a $-k_b + 2 - \alpha'(H,C) - k_b + \alpha(H,C) = -2k_b + 2 + 3k_b - 3 - k'_2 + 2 = k_b - k'_2 + 1$. Ainsi

$$\begin{aligned}
q' &= \left(10^{-H} 2^C g\right) \left(2^{k_b - k'_2 + 1}\right) \left(1 + \varepsilon_{t(H)}\right) (1+\varepsilon_1)(1+\varepsilon_2)(1+\varepsilon_3), \\
q' &= q \left(2^{k_b - k'_2 + 1}\right) \left(1 + \varepsilon_{t(H)}\right) (1+\varepsilon_1)(1+\varepsilon_2)(1+\varepsilon_3),
\end{aligned} \tag{5.55}$$

et on retrouve bien les termes de l'équation (5.50), avec l'erreur relative sur le calcul de la mantisse q' telle que $\left(1 + \varepsilon_{t(H)}\right) = \left(1 + \varepsilon_{t(H)}\right)(1+\varepsilon_1)(1+\varepsilon_2)(1+\varepsilon_3) - 1$. On cherche à borner ces erreurs.

On rappelle d'abord les bornes de $t(H)$, qui sont $2^{2k_b-2} \leq t(H) \leq 2^{2k_b-1}$ et étant donné les bornes sur g, on peut borner $t(H)\cdot g$ tel que $2^{3k_b-3} \leq t(H)\cdot g < 2^{3k_b-1}$. Ainsi, les bornes de $2^{-k_b}\cdot t(H)\cdot g$ sont $2^{2k_b-3} \leq 2^{-k_b}\cdot t(H)\cdot g < 2^{2k_b-1}$, et on peut borner l'erreur ε_1 telle que $\left\lfloor 2^{-k_b}\cdot t(H)\cdot g\right\rfloor_1 = 2^{-k_b}\cdot t(H)\cdot g\cdot(1+\varepsilon_1)$ avec

$$|\,\varepsilon_1\,| \leq 2^{-2k_b+3}. \tag{5.56}$$

Pour borner l'erreur ε_2 de la seconde partie entière inférieure, il faut borner l'exposant $\alpha'(H,C)$. On sait que $\alpha'(H,C) = \alpha(H,C) - \alpha_{\min}$, avec les bornes de $\alpha(H,C)$ détaillées dans l'équation (5.53). Ainsi, on détermine facilement que $0 \leq \alpha'(H,C)$. La borne supérieure est égale à

$$\begin{aligned} \alpha'(H,C) &\leq \left\lfloor 3k_b - 1 + (-k_{10}+1)\cdot\log_2(10)\right\rfloor - \left\lceil 3k_b - 3 - k_{10}\cdot\log_2(10)\right\rceil \\ \alpha'(H,C) &\leq 2 + \left\lfloor -k_{10}\cdot\log_2(10)\right\rfloor + \left\lfloor\log_2(10)\right\rfloor + 1 - \left\lceil -k_{10}\cdot\log_2(10)\right\rceil \end{aligned} \tag{5.57}$$

car d'après la définition de la partie entière inférieure, $\lfloor x+y\rfloor \leq \lfloor x\rfloor + \lfloor y\rfloor + 1$. De plus, sachant que $\lfloor x\rfloor - \lceil x\rceil + 1 = 0$ pour $x \notin \mathbb{Z}$, alors $\left\lfloor -k_{10}\cdot\log_2(10)\right\rfloor + 1 - \left\lceil -k_{10}\cdot\log_2(10)\right\rceil = 0$ pour $k_{10}\cdot\log_2(10) \notin \mathbb{Z}$. Comme $\left\lfloor\log_2(10)\right\rfloor = 3$ on peut borner $\alpha'(H,C)$ tel que $0 \leq \alpha'(H,C) \leq 5$. Ainsi, la seconde partie entière inférieure est bornée telle que $2^{2k_b-8} \leq \left\lfloor 2^{-\alpha'(H,C)}\left\lfloor 2^{-k_b}\cdot t(H)\cdot g\right\rfloor_1\right\rfloor_2 \leq 2^{2k_b-3}$. Finalement, l'erreur ε_2 telle que $2^{-\alpha'(H,C)}\left\lfloor 2^{-k_b}\cdot t(H)\cdot g\right\rfloor_1 \cdot (1+\varepsilon_2)$ est bornée comme

$$|\,\varepsilon_2\,| \leq 2^{-2k_b+8}. \tag{5.58}$$

La dernière borne ε_3 du calcul de q' tel que défini dans l'équation (5.54) se détermine simplement, la dernière multiplication étant une multiplication par 2^{-k_b+2}. Ainsi, q' est borné tel que $2^{k_b-6} \leq q' \leq 2^{k_b-1}$. Ainsi, la borne de l'erreur ε_3 est définie telle que

$$|\,\varepsilon_3\,| \leq 2^{-k_b+6}. \tag{5.59}$$

L'erreur relative totale du calcul de $q' = \left(10^{-H}2^C g\right)\left(2^{k_b-k'_2+1}\right)\left(1+\varepsilon_{\rho_D}\right)$ combine ces quatre erreurs de sorte que $\varepsilon_{\rho_D} = \left(1+\varepsilon_{t(H)}\right)(1+\varepsilon_1)(1+\varepsilon_2)(1+\varepsilon_3) - 1$. D'après les bornes sur toutes ces erreurs, on peut borner ε_{ρ_D} dans un premier temps telle que $\left|\,\varepsilon_{\rho_D}\,\right| \leq 2^{-k_b+6.5}$. On cherche à affiner cette erreur, en remarquant que lorsqu'on remplace H par sa définition dans l'expression de q', on a

$$\begin{aligned} q' &= \left(10^{-(\lfloor\log_{10}(2^C g)\rfloor - k_{10}+1)}2^C g\right)\left(2^{k_b-k'_2+1}\right)\left(1+\varepsilon_{\rho_D}\right) \\ &= \left(10^{-(\log_{10}(2^C g)+\delta-k_{10}+1)}2^C g\right)\left(2^{k_b-k'_2+1}\right)\left(1+\varepsilon_{\rho_D}\right) \\ q' &= 10^{-\delta+k_{10}-1}2^{k_b-k'_2+1}\left(1+\varepsilon_{\rho_D}\right), \end{aligned} \tag{5.60}$$

avec $-1 < \delta \leq 0$. D'où $10^{k_{10}-1} \leq 10^{-\delta+k_{10}-1} < 10^{k_{10}}$. On trouve donc de nouvelles bornes pour q', telles que

$$\begin{aligned} 10^{k_{10}-1}2^{k_b-k'_2+1}\left(1-\varepsilon_{\rho_D}\right) &\leq q' < 10^{k_{10}}2^{k_b-k'_2+1}\left(1+\varepsilon_{\rho_D}\right) \\ \left\lceil 10^{k_{10}-1}2^{k_b-k'_2+1}\left(1-2^{-kb+6.5}\right)\right\rceil &\leq q' \leq \left\lfloor 10^{k_{10}}2^{k_b-k'_2+1}\left(1+2^{-kb+6.5}\right)\right\rfloor \end{aligned} \tag{5.61}$$

Au final, on a que, pour toutes les valeurs de k_b, k'_2 et k_{10} concernées, q' est borné tel que $2^{+k_b-4.18} \leq q' \leq 2^{-k_b}-1$. Ainsi, la borne sur l'erreur de la conversion vers le format décimal de sortie est

$$\left|\varepsilon_{\rho_D}\right| \leq 2^{-k_b+4.18}. \tag{5.62}$$

5.3.4.2 Test d'arrondi

Le but de ce test est de déterminer si la sortie de précision k_b bits de l'algorithme d'addition/soustraction peut être arrondie en un nombre à virgule flottante binaire, resp. décimal, de précision k_n quel que soit son signe ou son mode d'arrondi.

Dans le cas binaire, on a comme entrée de ce test $2^C \cdot g = \left(\psi \pm c\right)\left(1+\varepsilon_B^*\right)$, avec la mantisse g bornée tel que $2^{k_b-1} \leq g \leq 2^{k_b}-1$, et ε_B^* l'erreur relative globale combinant toutes les erreurs précédentes, telle que

$$\left|\varepsilon_B^*\right| \leq 2^{-k_b+4.63}. \tag{5.63}$$

Le test d'arrondi revient à calculer statiquement la "borne magique" β_B, telle que

$$\beta_B = \left\lfloor\left(2^{k_b}\cdot\frac{\varepsilon_B^*}{1+\varepsilon_B^*}\right)\right\rfloor, \tag{5.64}$$

et tester si $\left| g - \left\lfloor 2^{k'_2-k_b-1}\cdot g\right\rfloor 2^{k_b-k'_2+1}\right| \geq \beta_B$. Si cette condition n'est pas vérifiée, arrondir le résultat vers un nombre binaire à virgule flottante de précision k_2 bits n'est pas possible. Cependant, nous pouvons calculer f_2, la borne d'arrondi binaire la plus proche de $2^C \cdot g$, telle que

$$\begin{aligned} 2^{FT_2}\cdot 5^{FF_2}\cdot f_2 &= 2^C \cdot 2^{k_b-k'_2+1}\cdot\frac{1}{2}\cdot f_2, \qquad 2^{k'_2-1} \leq f_2 \leq 2^{k'_2}-1, \\ C^{\min}+k_b-k'_2 &\leq FT_2 \leq C^{\max}+k_b-k'_2, \qquad FF_2 = 0. \end{aligned} \tag{5.65}$$

Cette borne d'arrondi sera utilisée dans l'étape suivante, la phase de récupération.

Dans le cas où le format de sortie est décimal, on a en entrée du test d'arrondi $2^H \cdot 2^{k'_2-k_b-1}\cdot q = \left(\psi \pm c\right)\left(1+\varepsilon_D^*\right)$, avec $10^{k_{10}-1}\cdot 2^{k_b-k'_2+1} \leq |q| \leq (10^{k_{10}}-1)\cdot 2^{k_b-k'_2+1}$, et ε_D^* l'erreur relative globale combinant toutes les erreurs relatives précédentes, telle que

$$\left|\varepsilon_D^*\right| \leq 2^{-k_b+5.42}. \tag{5.66}$$

Le test d'arrondi revient à calculer la "borne magique" β_D, telle que

$$\beta_D \;=\; \left\lfloor \left(10^{k_{10}}-1\right)\cdot 2^{k_b-k_2'}\cdot\frac{\varepsilon_D^*}{1+\varepsilon_D^*}\right\rfloor, \tag{5.67}$$

et tester si $\left| q-\left\lfloor 2^{k_b-k_2'}\cdot q\right\rfloor 2^{k_b-k_2'}\right| \;\geq\; \beta_D$. Si cette condition est satisfaite, l'arrondi final vers un nombre décimal à virgule flottante de précision k_{10} chiffres significatifs est impossible. On peut cependant calculer f_{10} la borne d'arrondi binaire la plus proche de $10^H\cdot 2^{-k_b+k_2'-1}\cdot q$ telle que

$$\begin{aligned} &2^{FT_{10}}\cdot 5^{FF_{10}}\cdot f_{10} = 10^H\cdot 2^{-k_b+k_2'-1}\cdot 2^{k_b-k_2'+1}\cdot\frac{1}{2}\cdot f_{10}, \qquad 2\cdot 10^{k_{10}-1}\leq f_{10}\leq 2\cdot(10^{k_{10}}-1).\\ &H^{\min}-1 \;\leq\; FT_{10} \;\leq\; H^{\max}-1, \qquad H^{\min} \;\leq\; FF_{10} \;\leq\; H^{\max}. \end{aligned} \tag{5.68}$$

Cette borne d'arrondi sera utilisée dans l'étape suivante, la phase de récupération.

5.3.4.3 Phase de récupération

On cherche à arrondir le résultat de notre algorithme de FMA, donné en format binaire ou décimal approchant le résultat exact x tel que $2^C\cdot g \;=\; x\left(1+\varepsilon_B^*\right)$ avec $\left|\varepsilon_B^*\right| \;\leq\; 2^{-k_b+4.63}$ en binaire. On connaît les bornes de l'exposant C à la fin de tous les calculs, telle que décrite dans l'équation (5.46). Les problèmes d'arrondi que l'on souhaite aborder après ce test d'arrondi ne concernent qu'une petite partie des exposants représentables dans cet intervalle. En effet, à partir d'un certain exposant, on peut affirmer avec certitude que la valeur calculée ne sera pas représentable dans le format de sortie, à cause d'un dépassement de capacité.

Nous souhaitons donc réduire l'intervalle C sur lequel nous allons travailler pour fournir l'arrondi correct lors de cette phase de récupération. La conversion produit un overflow si $x \;\geq\; 2^{E^{\max}+1}$, c'est-à-dire $2^C\cdot g \;\geq\; 2^{E^{\max}+1}\left(1+\varepsilon_B^*\right)$, et enfin $C \;\geq\; \left\lfloor \log_2\left(\frac{2^{E^{\max}+1}\left(1+\varepsilon_B^*\right)}{2^{k_b-1}}\right)\right\rfloor$. De la même façon, un underflow se produit lorsque $x \;\leq\; 2^{E^{\min}-k_2-1}$, ce qui implique que $2^C\cdot g \;\leq\; 2^{E^{\min}-k_2-1}\left(1-\varepsilon_B^*\right)$, et enfin $C \;\leq\; \left\lceil \log_2\left(\frac{2^{E^{\min}-k_2-1}\left(1-\varepsilon_B^*\right)}{2^{k_b}-1}\right)\right\rceil$. Le nouvel intervalle de définition de C est donc

$$\left\lceil \log_2\left(\frac{2^{E^{\min}-k_2-1}\left(1-\varepsilon_B^*\right)}{2^{k_b}-1}\right)\right\rceil \;\leq\; C \;\leq\; \left\lfloor \log_2\left(\frac{2^{E^{\max}+1}\left(1+\varepsilon_B^*\right)}{2^{k_b-1}}\right)\right\rfloor \tag{5.69}$$

Respectivement, le résultat décimal calculé est $10^H\cdot 2^{-k_b+k_2'-1}\cdot q \;=\; x\left(1+\varepsilon_D^*\right)$ avec $\left|\varepsilon_D^*\right| \;\leq\; 2^{-k_b+5.42}$. L'intervalle de représentation de H est soumis aux mêmes contraintes d'overflow et d'underflow que l'arrondi dans un format à virgule flottante binaire. De façon

similaire, on peut borner H tel que

$$\left\lceil \log_{10}\left(\frac{10^{F^{\min}-k_{10}-1}\left(1-\varepsilon_D^*\right)}{10^{k_{10}}-1}\right)\right\rceil \;\leq\; H \;\leq\; \left\lfloor \log_{10}\left(\frac{10^{F^{\max}+1}\cdot\frac{1}{4}\cdot\left(1+\varepsilon_D^*\right)}{10^{k_{10}-1}}\right)\right\rfloor \tag{5.70}$$

Si le test d'arrondi indique que l'arrondi final est impossible avec les résultats à la précision k_b, alors ce résultat est trop proche d'une frontière d'arrondi f_n. Ceci implique qu'il est impossible de décider vers lequel des nombres à virgule flottante les plus proches ce résultat doit être arrondi. On possède cependant une information sur cette frontière d'arrondi f aux bornes harmonisées telles que $2^{k_2'-5} < f < 2^{k_2'}-1$, et les exposants FT et FF bornés tels que $\min\left(FT_2^{\min}, FT_{10}^{\min}\right) \leq FT \leq \max\left(FT_2^{\max}, FT_{10}^{\max}\right)$ et $\min\left(FF_2^{\min}, FF_{10}^{\min}\right) \leq FF \leq \max\left(FF_2^{\max}, FF_{10}^{\max}\right)$. Il est donc inutile de recalculer le FMA entièrement, on peut donc simplement calculer le signe de

$$\alpha_n = (-1)^{s_\psi}\cdot 2^L\cdot 5^M\cdot s + (-1)^{s_c}\cdot 2^{N_c}\cdot 5^{P_c}\cdot t_c - 2^{FT}\cdot 5^{FF}\cdot f. \tag{5.71}$$

Le signe de cette quantité peut être calculé comme le signe d'une quantité calculée sur $\mathbb{Z}$ sur un nombre de bits prédéfini. Il suffit de définir les deux variables suivantes

$$\underline{Z}_T = \min(L^{\min}, N_c^{\min}, FT^{\min}), \qquad \underline{Z}_F = \min(P_c^{\min}, M^{\min}, FF^{\min}). \tag{5.72}$$

On peut ensuite normaliser le résultat de α sur $\mathbb{Z}$ à l'aide de ces variables, tel que

$$\alpha_n' \;=\; (-1)^{s_\psi}\cdot 2^{L-\underline{Z}_T}5^{M-\underline{Z}_F}s + (-1)^{s_c}\cdot 2^{N_c-\underline{Z}_T}5^{P_c-\underline{Z}_F}t_c - 2^{FT-\underline{Z}_T}5^{FF-\underline{Z}_F}f_n \tag{5.73}$$

dont les exposants sont bornés tels que

$$\begin{aligned}
0 \;\leq\; L-\underline{Z}_T \;\leq\; L^{\max}-\underline{Z}_T, \qquad & 0 \;\leq\; M-\underline{Z}_F \;\leq\; M^{\max}-\underline{Z}_F,\\
0 \;\leq\; N_c-\underline{Z}_T \;\leq\; N_c^{\max}-\underline{Z}_T, \qquad & 0 \;\leq\; P_c-\underline{Z}_F \;\leq\; P_c^{\max}-\underline{Z}_F,\\
0 \;\leq\; FT-\underline{Z}_T \;\leq\; FT^{\max}-\underline{Z}_T, \qquad & 0 \;\leq\; FF-\underline{Z}_F \;\leq\; FF^{\max}-\underline{Z}_F.
\end{aligned} \tag{5.74}$$

Les bornes de ces exposants peuvent de nouveau être affinées. Ceci est important pour que l'étape de récupération consomme le moins de ressources possibles, en temps et en espace mémoire. On peut définir f dans les deux bases tel que

$$\begin{aligned}
f \;&=\; \left\lfloor 2^{-FT}5^{-FF}\cdot\left((-1)^{s_\psi}\cdot 2^L5^Ms + (-1)^{s_c}\cdot 2^{N_c}5^{P_c}t_c\right)\right\rceil,\\
f-\delta \;&=\; 2^{-FT}5^{-FF}\left((-1)^{s_\psi}\cdot 2^L5^Ms + (-1)^{s_c}\cdot 2^{N_c}5^{P_c}t_c\right),\\
f\cdot\left(1+\varepsilon_f\right) \;&=\; 2^{-FT}5^{-FF}\left((-1)^{s_\psi}\cdot 2^L5^Ms + (-1)^{s_c}\cdot 2^{N_c}5^{P_c}t_c\right),
\end{aligned} \tag{5.75}$$

avec $-\frac{1}{2} \leq \delta \leq \frac{1}{2}$ et $\varepsilon_f = -\frac{\delta}{f}$. On peut donc borner ε_f tel que $-\frac{1}{2}\cdot\frac{1}{2^{k_2'-5}} \leq -\frac{\delta}{f} \leq \frac{1}{2}\cdot\frac{1}{2^{k_2'}}$, soit $2^{-k_2'+4} \leq \varepsilon_f \leq 2^{-k_2'-1}$, ce qui implique que $\left|\,\varepsilon_f\,\right| \leq 2^{-k_2'+4}$.

On sait, d'après le lemme 5.29 que $\left| 1 - \frac{2^{N_c}5^{P_c}t_c}{2^L 5^M s} \right| \geq \eta$, ce qui peut être transformé en $2^{L-FT}5^{M-FF}s - 2^{N_c-FT}5^{P_c-FF}t_c \geq \eta \cdot 2^{L-FT}5^{M-FF}s$. Étant donné les bornes de f, ε_f et s, on en déduit les bornes de $2^{L-FT}5^{M-FF}$ telles que

$$\underline{z}_\psi = \frac{2^{k'_2-5}\left(1-2^{-k'_2+4}\right)}{2^{2\cdot k'_2}} \leq 2^{L-FT}5^{M-FF} \leq \frac{1}{\eta}\cdot\frac{2^{k'_2}\left(1+2^{-k'_2+4}\right)}{2^{2\cdot k'_2-1}} = \bar{z}_\psi. \tag{5.76}$$

Ainsi, on peut définir les nouvelles bornes de L et M, telles que

$$\begin{aligned} \left\lceil \log_5\left(\underline{z}_\psi \cdot 2^{FT^{\min}-L^{\min}}\right) + FF^{\min} \right\rceil &\leq M \leq \left\lfloor \log_5\left(\bar{z}_\psi \cdot 2^{FT^{\max}-L^{\max}}\right) + FF^{\max} \right\rfloor, \\ \left\lceil \log_2\left(\underline{z}_\psi \cdot 5^{FT^{\min}-M^{\min}}\right) + FF^{\min} \right\rceil &\leq L \leq \left\lfloor \log_2\left(\bar{z}_\psi \cdot 2^{FT^{\max}-M^{\max}}\right) + FF^{\max} \right\rfloor. \end{aligned} \tag{5.77}$$

Boucler sur ces équations permet d'obtenir des bornes plus fines pour L et M, qui se stabilisent à un certain moment. On peut ainsi utiliser ces bornes pour la suite de la phase de récupération.

De la même façon, on peut déduire du lemme 5.29 que $\left| 1 - \frac{2^L 5^M s}{2^{N_c}5^{P_c}t_c} \right| \geq \eta\frac{1}{\eta-1}$, et on se sert de cette équation pour retrouver les bornes de P et N. Étant donné les bornes de f, ε_f et t, on en déduit les bornes de $2^{N_c-FT}5^{P_c-FF}$ telles que

$$\underline{z}_c = \frac{2^{k'_2-5}\left(1-2^{-k'_2+4}\right)}{2^{k'_2}} \leq 2^{N_c-FT}5^{P_c-FF} \leq \frac{\eta-1}{\eta}\cdot\frac{2^{k'_2}\left(1+2^{-k'_2+4}\right)}{2^{k'_2-1}} = \bar{z}_c. \tag{5.78}$$

Ainsi, on peut déduire les nouvelles bornes de N_c et P_c de façon similaire, telles que

$$\begin{aligned} \left\lceil \log_5\left(\underline{z}_c \cdot 2^{FT^{\min}-N_c^{\min}}\right) + FF^{\min} \right\rceil &\leq P_c \leq \left\lfloor \log_5\left(\bar{z}_c \cdot 2^{FT^{\max}-N_c^{\max}}\right) + FF^{\max} \right\rfloor, \\ \left\lceil \log_2\left(\underline{z}_c \cdot 5^{FT^{\min}-P_c^{\min}}\right) + FF^{\min} \right\rceil &\leq L \leq \left\lfloor \log_2\left(\bar{z}_c \cdot 2^{FT^{\max}-P_c^{\max}}\right) + FF^{\max} \right\rfloor. \end{aligned} \tag{5.79}$$

Étant donné ces bornes, on sait que nous allons devoir calculer des puissances de 5 positives ou nulles, soit 5^D, avec $0 \leq D \leq \max\left(FF-\underline{Z}_F, M-\underline{Z}_F, P_c-\underline{Z}_F\right)$. De plus, nous savons que 5^D tient sur $\left\lceil \log_2\left(5^{D^{\max}}\right)\right\rceil$ bits, donc D_W mots de 64 bits. Ces valeurs sont décrites selon le format de sortie visé dans le tableau 5.4

On peut donc calculer exactement $\alpha' \in \mathbb{Z}$ dans un accumulateur alloué statiquement, dont on peut déterminer la taille. Ainsi on borne d'abord la taille de $2^L \cdot 5^M \cdot s$ telle que $i_1 = \left\lceil \log_2\left(2^{L^{\max}}5^{M^{\max}}s^{\max}\right)\right\rceil$. De même $i_2 = \left\lceil \log_2\left(2^{N_c^{\max}}5^{P_c^{\max}}t_c^{\max}\right)\right\rceil$ et enfin $i_3 = \left\lceil \log_2\left(2^{FT^{\max}}5^{FF^{\max}}f^{\max}\right)\right\rceil$. On a donc que la taille maximale de l'accumulateur nécessaire pour calculer exactement $\alpha' \in \mathbb{Z}$, telle que $AC^{\max} = \max(i_1, i_2, i_3) + 2$, et le nombre de mots machine de 64 bits nécessaires pour représenter le résultat est AC_W.

	binary64, decimal64	binary128, decimal128
$D^{\max}$ (bits)	1823	28614
D_W (mots machine)	29	448
$AC^{\max}$ (bits)	4220	62092
AC_W (mots machine)	66	971
bits de garde	4	52

Tableau 5.4 – Tailles binaires de l'accumulateur

Une fois $\alpha' \in \mathbb{Z}$ calculé, on peut alors déterminer le signe de α. Il y a trois cas : si $\alpha = 0$ alors le résultat est exactement la borne d'arrondi f, et sera arrondi correctement dans le format de sortie par notre algorithme d'arrondi final selon le mode d'arrondi courant. Si $\alpha < 0$, alors le résultat est en dessous de la borne d'arrondi f et sera arrondi vers le nombre à virgule flottante inférieur le plus proche dans le format de sortie. Par analogie, si $\alpha > 0$ alors le résultat sera arrondi vers le nombre à virgule flottante supérieur le plus proche dans le format de sortie.

Calculer cette phrase de récupération est ainsi très coûteuse, mais par chance, cette partie de l'algorithme de FMA est peu utilisée. Elle est appelée uniquement lorsque les additions/soustractions far path et near path ne parviennent pas à calculer un résultat suffisamment précis pour l'arrondi final. Les opérations coûteuses sont les multiplications par une puissance de cinq, qui est représentée par un grand nombre de mots de 64 bits stockés dans un tableau, et une mantisse, qui est un nombre relativement petit par rapport à cette première opérande. Ainsi, fort heureusement, toutes les multiplications sont rectangulaires, c'est-à-dire qu'on multiplie un nombre avec un autre relativement petit en comparaison. Cela signifie qu'il est possible d'effectuer ces multiplications sans allocation de mémoire dynamique, ce qui sera toujours plus rapide que n'importe quel algorithme optimisé pour la multi-précision.

5.3.4.4 Arrondi final dans le format de sortie

L'arrondi final dans le format de sortie choisi, binary64, binary128, decimal64 ou decimal128, prend en entrée soit $(-1)^s \cdot 2^C \cdot g$, $2^{k_b-1} \leq g < 2^{k_b}$, ou $(-1)^s \cdot 10^H \cdot 2^{-k_b+k'_2-1} \cdot q$, $10^{k_{10}-1} \cdot 2^{k_b-k'_2+1} \leq g < 10^{k_{10}} \cdot 2^{k_b-k'_2+1}$, et les arrondi vers un nombre à virgule flottante respectivement binaire ou décimal, étant donné le mode d'arrondi IEEE 754 courant. L'arrondi binaire est effectué avec l'arrondi IEEE 754 binaire pour l'arithmétique à virgule flottante, et l'arrondi décimal avec l'arrondi décimal. Cet arrondi final permet également de lever les drapeaux `inexact`, `overflow` ou `underflow`, lorsque l'arrondi est respectivement

inexact, sujet à un underflow (petit ou inexact), ou sujet à un overflow.

La difficulté dans l'implémentation de cet arrondi est double : premièrement l'arrondi doit s'occuper de tous les cas spéciaux, tels que l'arrondi des nombres dénormalisés, l'arrondi des nombres normalisés, l'overflow, la dénormalisation décimale pour le plus petit exposant, etc. Deuxièmement, l'arrondi doit être effectué avec le mode d'arrondi approprié, dont l'accès est difficile. Il doit également lever les drapeaux. Nous résolvons ces problématiques en calculant exactement des nombres binaires, resp. décimaux, tels que donnés en entrée d'une opération à virgule flottante IEEE 754 sont arrondis à la même valeur que la donnée d'entrée.

Pour l'arrondi binaire de précision k_2, nous décomposons $(-1)^s \cdot 2^C \cdot g$ en une valeur binaire normalisée 2^A de précision k_2, une autre valeur binaire normalisée $2^B \cdot r$ de précision k_2 et une valeur normalisée ou dénormalisée $2^{C'} \cdot g'$ tels que, un FMA binaire calcule

$$2^A \times 2^B \cdot r + 2^{C'} \cdot g', \tag{5.80}$$

et l'arrondi de ce résultat est la même valeur binaire de précision k_2 que l'arrondi de $(-1)^s \cdot 2^C \cdot g$. Ceci signifie qu'on a

$$\begin{aligned} C' &= C + k_b - k_2' + 2, \qquad g' = \left\lfloor g \cdot 2^{-k_b + k_2' - 2} \right\rfloor \\ A + B &= C, \qquad r = g - 2^{k_b - k_2' + 2} \cdot g'. \end{aligned} \tag{5.81}$$

Expliquons d'abord l'arrondi vers le format decimal64, c'est-à-dire lorsqu'on a la précision $k_{10} = 16$. On représente $(-1)^s \cdot 10^H \cdot 2^{-k_b + k_2' - 1} \cdot q$ de façon exacte comme un nombre à virgule flottante IEEE 754 dans le format decimal128, et on effectue la conversion IEEE 754 du format decimal128 vers le format decimal64, qui fournit l'arrondi correct et lève les bons drapeaux. Cette manière d'effectuer l'arrondi final est sous-optimale en terme de performances mais a l'avantage d'être facile à implémenter.

On ne peut pas utiliser cette méthode dans le cas de l'arrondi vers le format decimal128, c'est-à-dire lorsqu'on a la précision $k_{10} = 34$, la conversion vers un nombre à virgule flottante supposé IEEE 754 decimal256 n'est pas possible. On doit donc effectuer l'arrondi manuellement, en trouvant le bon mode d'arrondi en effectuant les opérations en decimal128 suivantes

$$t = (-1)^s \cdot \left(10^{k_{10}-1} + 1\right), \qquad t' = (-1)^s \cdot \left(10^{k_{10}-1} + 2\right). \tag{5.82}$$

On ajoute δ à ces quantités, avec $\delta \in \left\{-\frac{3}{4}, -\frac{1}{2}, -\frac{1}{4}, 0, \frac{1}{4}, \frac{1}{2}, \frac{3}{4}\right\}$, en choisissant t, t' et δ selon la parité de la mantisse tronquée et la valeur du reste de cette troncature. Pour un arrondi vers un decimal64, nous pourrions effectuer une opération similaire.

Étant donné cet arrondi final des formats internes $2^C \cdot g$ et $10^H \cdot q$ vers indistinctement un binary64, binary128, decimal64 ou decimal128 FMAs, nous pouvons alors considérer

tous les cas des FMAs en bases mixtes hétérogènes, puis toutes les opérations arithmétiques de base.

5.4 Développements et résultat expérimentaux

Tous les développements et résultats expérimentaux ont été effectué pour le cas spécial d'opérations de FMA aux opérandes représentées en binary64 ou decimal64. Le générateur de code pour tous les FMA hétérogènes binary64, binary128, decimal64 et decimal128 se comporte essentiellement de la même manière.

5.4.1 Développement de deux références

Pour effectuer la validation de notre implémentation de FMA en bases mixtes, nous avons conçu et développé deux autres réalisations du FMA en bases mixtes : l'un, basé sur la bibliothèque GNU Multiple Precision Library (GMP) [49], respectant la même interface d'appel en C que notre implémentation de FMA optimisée, et l'autre, écrite en Sollya [58], avec une interface textuelle manuelle spéciale. Nous définissons les buts suivants lors de la conception de ces deux références : l'implémentation basée sur la bibliothèque GMP est conçue pour représenter au mieux un FMA en bases mixtes développé dans un laps de temps limité, en réutilisant des bibliothèques existantes. Elle doit être raisonnablement rapide et facile à concevoir. La référence développée à l'aide de Sollya quant à elle peut être lente mais doit renvoyer le résultat attendu lors du calcul d'un FMA en bases mixtes avec arrondi correct en toutes circonstances.

L'implémentation basée sur la bibliothèque GMP aborde les objectifs donnés ci-dessus en convertissant dans un premier temps toutes les entrées, indistinctement leur représentation en nombres à virgule flottante IEEE 754-2008 binaire ou décimal, en nombres rationnels GMP, c'est-à-dire fractions de (grands) entiers. On effectue ensuite les étapes de multiplication et d'addition du FMA en nombres rationnels GMP. GMP peut réaliser ces opérations sur des nombres rationnels de façon exacte, soit sans aucune erreur, en calculant de longs produits, sommes et pgcds comme exigé, en allouant la mémoire de façon dynamique. Le résultat rationnel, notons le $\frac{p}{q}$, est ensuite arrondi, en déterminant d'abord l'exposant E de la sortie pour un nombre à virgule flottante binaire, resp. décimal. Le nombre rationnel $\frac{r}{s} = 2^{-E} \cdot \frac{p}{q}$ (resp. $\frac{r}{s} = 10^{-E} \cdot \frac{p}{q}$) est "arrondi" vers une mantisse entière par une longue division des deux grands entiers r et s, en adaptant le nombre du reste de la division tel que requis par le mode d'arrondi. Comme les exposants des entrées/sorties peuvent être relativement grands, la bibliothèque GMP manipulera dans le pire cas des entiers d'environ quelques milliers de bits.

Dans la section 5.4.2 suivante, nous donnerons un aperçu du processus de test de notre implémentation de FMA en bases mixtes optimisé, lors duquel nous comparerons ses résultats à ceux de l'implémentation de référence en GMP, ainsi que leurs performances temporaires. Lors du développement de ces deux implémentations et en effectuant des tests intermédiaires de notre version optimisée et la version en GMP, nous avons trouvé des entrées pour lesquelles les résultats étaient différents. Nous avons donc conçu un troisième FMA en bases mixtes dans le but de trouver un moyen de tester et trouver les erreurs dans les résultats de ces deux implémentations. Cette troisième implémentation est écrite dans le langage numérique interprété Sollya [58]. Dans cet outil, les expressions numériques peuvent être écrite de façon exacte, et évaluées à n'importe quelle précision sans être sujettes à des arrondis fallacieux. Il est donc facile de donner les définitions des exposants et mantisses attendus en sortie du FMA en bases mixtes étant donné les exposants et mantisses des entrées. Les cas particuliers survenant lors des arrondis de nombres dénormalisés, d'overflow ou d'underflow peuvent être traités grâce aux calculs des minimim/maximum.

Nous avons utilisé notre implémentation de référence en Sollya pour générer les vecteurs de tests comprenant des entrées, le mode d'arrondi choisi et les résultats exacts attendus. Ces vecteurs de tests ont été utilisés pour la vérification de notre implémentation de FMA optimisé et la référence basée sur GMP. Après notre campagne de tests, les résultats de ces trois implémentations étaient cohérents. Quand bien même nous avons passé un certain temps à tester nos implémentations, nous considérons notre algorithme de FMA en bases mixtes correct uniquement car nous avons été capables d'en donner la preuve papier dans les sections précédentes.

5.4.2 Tests et performances temporelles

Nous avons testé notre implémentation du FMA en bases mixtes dont les formats des opérandes sont binary64 ou decimal64 dans ses 14 différentes versions sur un système basé sur un quad-core processeur Intel i7-7500U tournant au maximum à 2.7GHz, avec un système d'exploitation Debian/GNU Linux 4.9.0-5 en mode x86-64. Nous avons compilé notre code en utilisant *gcc* version 6.3.0, avec des optimisations de niveau 3 et en paramétrant le drapeau `-march=native`. Sur notre système, ces résultats utilisent des instructions AVX, en comprenant le FMA binary64 présent en matériel. Les types décimaux et instructions IEEE 754-2008 viennent de la bibliothèque *libgcc* de gcc. L'implémentation de référence GMP est basée sur GMP version 6.1.2. Sollya était au git commit 12bf2006c.

Nous avons testé chacune des 14 variantes sur fichier de vecteurs de tests approprié, contenant au moins 115000 points de test, ceux-ci couvrant tous les cas possibles de signes, zéros signés, infinis, NaNs, dénormalisés et nombres normalisés, ainsi que tous

les modes d'arrondi possibles en binaire et en décimal [2]. Nous avons trouvé que nos 14 implémentations de FMA en bases mixtes étaient justes vis-à-vis de leurs soties numériques pour tous les cas considérés.

Nous effectuons également le test des implémentations quant à la levée des drapeaux IEEE 754-2008 corrects, soit les drapeaux `inexact`, `overflow`, `underflow`, `divide-by-zero` et `invalid` [38]. Nous avons vérifié que tous les drapeaux sont levés correctement, c'est-à-dire que notre implémentation lève correctement les drapeaux IEEE 754-2008 appropriés lorsque l'opération était en effet inexacte (soit effectuant un arrondi), sujette à un overflow ou underflow, ou était invalide (telle que la multiplication de zéro et l'infini).

L'implémentation a également été vérifiée lorsqu'un drapeau doit être laissé inchangé s'il n'a pas rencontré la condition définie par la norme IEEE 754-2008 adaptée pour être levé. Un problème a été trouvé dans ce dernier cas de vérification des implémentations FMA en bases mixtes : pour certains résultats décimaux exactes, le drapeau `inexact` est levé de manière fallacieuse. Nous avons remonté cette erreur à l'implémentation de la conversion IEEE 754-2008 du format decimal128 vers decimal64 présente dans notre version de *libgcc*. Un rapport de bug a été envoyé à *libgcc*.

Concernant les tests de performance, nous avons exécuté nos implémentations sur chacune des entrées de nos vecteurs de test, en chronométrant le temps d'appel de la fonction sous test en termes de cycles machine utilisant l'instruction Intel `rdtsc` après l'individualisation avec `cpuid`. Nous avons déduit le temps d'appel mesuré pour une fonction vide avec le même prototype. Nous avons exécuté les mesures de chronométrage à plusieurs reprises et avec des caches préchauffées, afin d'atténuer les effets des processeurs superscalaires modernes. Nous nous sommes assurés que les vecteurs de test contenaient une représentation d'un sous-ensemble aléatoire de nombres à virgule flottante, c'est-à-dire que nous nous sommes assurés que les cas particuliers tels que l'addition et la multiplication par zéro ou NaN, etc. n'étaient pas surreprésentés.

Nous présentons nos mesures de performance temporelles sous la forme d'histogrammes associant un certain intervalle de nombre de cycles au nombre de fois où un nombre de cycles présent dans cet intervalle a été rencontré. Nous avons effectué le même test de performance sur notre implémentation optimisée du FMA en bases mixtes et sur celle basée sur GMP. Les résultats de chacune des 14 fonctions testées sont similaires, nous ne rapportons ainsi que les résultats des tests de performance d'une seule fonction FMA, appelée DBBB. Cette fonction prend tous ses argument sous dans le format IEEE 754-2008 binary64 et renvoie un decimal64.

L'histogramme à gauche de la figure 5.1 représente le nombre de cycles d'exécution de la fonction FMA DBBB optimisée. On voit facilement que notre implémentation calcule la

2. et toutes les combinaisons de modes d'arrondi, comme la norme IEEE 754-2008 spécifie des modes d'arrondis différents pour les sous-ensembles des opérations binaires et des opérations décimales [38]

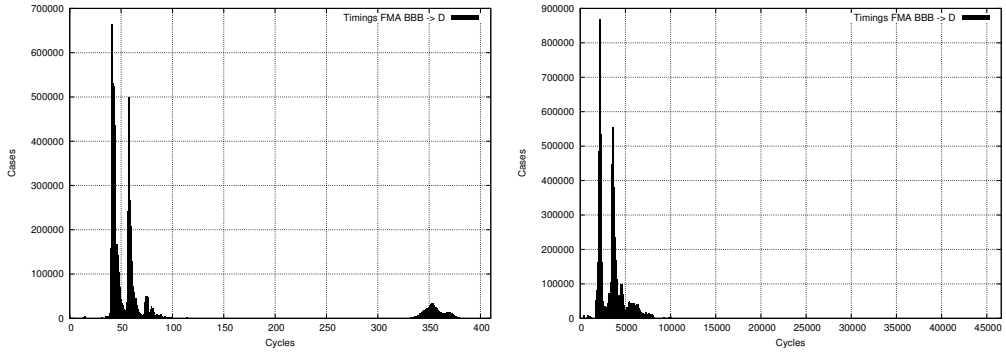


FIGURE 5.1 – Histograms des mesures de performances temporelles du FMA en bases mixtes optimisé (gauche) et de la référence GMP (droite)

plupart des cas en moins de 50 cycles. On remarque que le temps d'appel maximum de notre implémentation est d'environ 375 cycles, ce qui correspond aux cas faisant appel à la phrase de récupération pour effectuer l'arrondi final. Comme notre algorithme est borné de façon statique en terme de consommation mémoire et n'effectue pas d'allocation dynamique, ces bornes supérieures des performances temporelles sont strictes.

L'histogramme à droite de la figure 5.1 représente le nombre de cycles d'exécution de la fonction FMA DBBB implémentée avec les nombres rationnels de GMP. Pour la plupart des cas, l'implémentation GMP est environs 10 fois plus lente que l'algorithme présenté dans ce chapitre. Le temp d'appel maximum du FMA GMP atteint jusqu'à 46000 cycles dans quelques cas rares, visibles uniquement par la taille du graphe. Ces cas sont environs 100 fois plus lent que nos performances les plus mauvaises. Ceci est causé par le coût entraîné par le mécanisme d'allocation dynamique de mémoire de GMP.

5.5 Conclusion

Nous avons présenté un algorithme permettant de calculer une opération Fused-Multiply-and-Add correctement arrondie en bases mixtes produisant une sortie dans un format IEEE 754-2008 binary64, binary128, decimal64 ou decimal128. Cette opération prend en argument n'importe quelle combinaison de nombres IEEE 754-2008 binary64, binary128, decimal64 et decimal128. Notre algorithme permet de façon triviale d'effectuer l'addition et la multiplication en bases mixtes avec arrondi correct. De plus, notre algorithme est conçu pour permettre d'effectuer la division et la racine carrée en bases mixtes à partir de ce FMA.

Notre algorithme est basé sur un mélange de multiplication exacte des mantisses, de conversion entre les bases 2 et 10 à la précision finement ajustée, d'addition binaire et

d'une phase de récupération de l'arrondi exact qui est effectuée uniquement si l'arrondi de l'approximation ne permet pas de réaliser l'arrondi correct du résultat calculé. La phase de récupération est basée sur l'observation que tous les nombres à virgule flottante IEEE 754-2008 binaires et décimaux et leurs points milieux sont des entiers multiples d'une valeur fixe $2^Q \cdot 5^W$, où $Q, W \in \mathbb{Z}, Q, W \leq 0$ sont des constantes.

Nous avons effectué une analyse rigoureuse de notre algorithme de FMA, afin de généraliser notre approche aux formats à virgule flottante binary64, binary128, decimal64 et decimal128. Cette nouvelle approche nous a permis d'entreprendre l'implémentation d'un générateur de code, utilisant le calcul automatique des bornés de chaque variable et des valeurs clefs de l'algorithme.

Notre implémentation, dans le cas particulier d'opérandes aux formats binary64 et decimal64 peut traiter la plupart des cas en approximativement 50 cycles machine, ce qui est une performance temporelle raisonnable pour une opération à virgule flottante implémentée en logiciel. Quand la phase de récupération est utilisé, la latence reste inférieure à 400 cycles machine. Notre implémentation ne nécessite aucune allocation de mémoire dynamique, et peut donc être adapté pour l'intégration dans des environnements où la gestion de mémoire dynamique n'est pas disponible.

Notre implémentation a été minutieusement testée par rapport à deux autres implémentations de références que nous avons développées. Hormis un problème indépendant de notre volonté de drapeau IEEE 754-2008 `inexact` levé dans un contexte inapproprié, aucune erreur n'a été relevée lors de la campagne de tests. De plus, nous fournissons dans ce chapitre une preuve détaillée de chacune des étapes de l'algorithme.

Nous sommes confiants dans le fait qu'une fois le générateur de code implémenté et soigneusement testé, nous serons toujours plus rapides que notre implémentation de référence, car la phase coûteuse de récupération est tout de même plus optimisée pour des multiplications rectangulaires.

Quand bien même nous avons une forte conviction que notre implémentation l'une des premières étapes pour défricher le terrain pour l'arithmétique à virgule flottante IEEE 754-2008 généralisée en bases mixtes, un certain nombre de points doivent être traités par de futurs travaux : nous avons intégré toutes les valeurs binaires et décimales possibles en une combinaison de formats d'entrée/sortie en bases mixtes pour le FMA dans un seul algorithme. Cela a facilité le développement et la preuve de validité, mais induit une surestimation des intervalles de définition des variables internes, notamment des exposants. Des travaux futurs pourraient se charger d'effectuer une analyse plus fine au cas par cas.

TROISIÈME PARTIE

GÉNÉRATION AUTOMATIQUE DE TESTS NUMÉRIQUES

CHAPITRE 6

PROGRAMMATION PAR CONTRAINTES ET MUTATION TESTING

Il existe de nombreuses méthodes de génération de tests, comme détaillées dans le chapitre 3. Chacune de ces méthodes se distingue par le cadre applicatif dans lequel elle évolue, c'est-à-dire à partir de quelle sémantique elle se base, et quels objectifs de tests elle vise à remplir. Une des problématiques de la génération de tests pour la certification de logiciels embarqués pour l'aéronautique est de n'utiliser que la description du comportement fonctionnel du système. D'autre part, nous souhaitons nous concentrer sur les exigences numériques du système. Ces exigences peuvent être de différentes natures (combinatoires ou séquentielles, linéaires ou non, etc.), qui ne sont pas nécessairement compatibles avec un formalisme unique décrivant les fonctionnalités du logiciel. Ceci complexifie donc la détermination d'un algorithme de génération de tests dans le cas général.

La programmation par contrainte et le Mutation Testing sont deux approches qui ont déjà été considérées dans le contexte du test logiciel [9, 101]. Le but de ce chapitre est d'apporter une preuve de faisabilité d'une combinaison de ces deux méthodes pour la génération automatique de tests sur un exemple d'exigence simplifiée. Nous partons du cas particulier de la génération de tests numériques pour des exigences combinatoires (linéaires ou non). Dans un premier temps, nous nous intéresserons à la programmation par contraintes, dont nous détaillerons les caractéristiques pour la formalisation d'exigences fonctionnelles numériques. Puis nous présenterons en détail les spécificités du Mutation Testing, et l'application de cette méthode pour la génération de tests à partir d'objectifs de tests précis. Enfin nous appliquerons les principes du Mutation Testing sur une spécification formalisée par des contraintes, dans le but de construire un générateur automatique de tests adaptés pour le contexte de certification aéronautique.

Ces travaux en collaboration avec Martine Ceberio ont mené à la publication et la présentation d'un résumé lors de la conférence *18th International Symposium on Scientific Computing, Computer Arithmetics and Verified Numerics (SCAN)* [10] en 2018.

6.1 Spécification fonctionnelle sous forme de contraintes

La programmation par contrainte se distingue de toutes les méthodes de modélisations étudiées dans le chapitre 3 par sa capacité à représenter les exigences numériques par leur comportement fonctionnel, ou sans précision aucune sur le design, en permettant de garantir la précision du résultat. Dans cette section, nous verrons en détails les principes associés à la programmation par contrainte et son application pour la définition d'exigences fonctionnelles.

6.1.1 Introduction à la programmation par contraintes

La programmation par contraintes est un paradigme permettant d'exprimer les relations entre différentes entités sous la forme de *contraintes*, comme des exigences, obligations ou interdictions, devant être satisfaites. La programmation par contraintes est essentiellement caractérisée par deux niveaux [85] : la *modélisation* consiste en l'identification, l'examen, et la formalisation d'un problème réel par des experts, et la *résolution* consiste en la réduction du problème modélisé par des contraintes en un ensemble de solutions, à l'aide de *solveurs de contraintes*.

La programmation par contraintes est un outil puissant utilisé pour résoudre des problèmes de satisfaction de contraintes (CSP). Un CSP [109] est défini par un triplet $\mathscr{P} = (\mathscr{X}, \mathscr{D}, \mathscr{C})$ tel que

- $\mathscr{X} = \{x_1, \ldots, x_n\}$ est un ensemble fini de variables,
- $\mathscr{D} = \{D_1, \ldots, D_n\}$ est un ensemble de domaines de définition, tels qu'un domaine D_i définit toutes les valeurs possibles pour chaque variable x_i, pour $i = 1, \ldots, n$,
- $\mathscr{C} = \{c_1, ..., c_m\}$ est un ensemble fini de contraintes, restreignant les valeurs que les variables peuvent prendre simultanément.

Une *solution* du CSP est un ensemble de valeurs $\{d_1 \in D_1, \ldots, d_n \in D_n\}$ pour les variables $\{x_1, \ldots, x_n\}$, qui sont cohérentes par rapport aux contraintes, c'est-à-dire que chaque variable prend une valeur de son domaine de définition de sorte que toutes les contraintes sont satisfaites en même temps. Résoudre un CSP peut signifier rechercher plusieurs types de solutions dans la boîte initiale $D_1 \times \cdots \times D_n$. On peut vouloir déterminer si le système contient au moins une solution quelconque, ou bien trouver toutes les solutions du problème, ou encore déterminer une solution optimale étant donné un certain objectif défini selon plusieurs variables [110]. Une généralisation d'un CSP pour lequel on souhaite trouver une solution optimale selon un critère d'évaluation de coût est appelée un problème d'optimisation de contraintes (COP). Dans ce chapitre nous nous concentrerons sur la

résolution de CSP pour la preuve de concept de la méthode de génération automatique de tests. Nous nous intéresserons à l'optimisation des solutions dans un second temps.

Les domaines de définition des variables peuvent être de plusieurs natures. La nature des variables manipulées a un impact direct sur la représentation des contraintes et les algorithmes de résolution utilisés pour trouver des solutions. Les CSP contenant uniquement des variables booléennes correspondent à un problème de satisfiabilité, ou problème SAT [109]. Un grand nombre de problèmes est modélisé à l'aide de variables entières qui appartiennent à des domaines discrets de cardinalité finie. Les ensembles numériques continus peuvent quant à eux être représentés par des intervalles [111]. Nous nous intéressons à ce dernier type de domaines. Les contraintes sur variables aux domaines représentés par des intervalles continus utilisent l'arithmétique d'intervalles conforme à la définition donnée dans la section 2. La solution est ainsi représentée par une boîte formée de l'union d'intervalles. Dans les faits, les solveurs de contraintes permettant de résoudre ce genre de problèmes effectuent leurs calculs sur des nombres à virgule flottante, donc sur une discrétisassion des domaines continus. Néanmoins la programmation par contraintes sur les intervalles permet d'apporter des résultats garantis aux problèmes combinatoires non linéaires complexes [111, 6].

6.1.2 Résolution de contraintes sur des intervalles

Une fois le CSP défini et le domaine de définition des variables fixé, on peut faire appel à un solveur de contraintes pour chercher un ensemble de solutions. Les approches classiques de résolution de contraintes représentées par des domaines continus, comme les intervalles, consistent en des procédures par réduction et séparation (*branch and prune*) [94, 111]. L'ensemble de solutions défini à une certaine précision Δ est construit de façon progressive par l'exploration de différentes parties de l'espace de recherche.

Étant donné un CSP $\mathscr{P} = (\mathscr{X}, \mathscr{D}, \mathscr{C})$ et une précision absolue de la taille des boîtes Δ, l'étape de *réduction* consiste à évaluer la boîte $D_1 \times \cdots \times D_n$ définie par l'ensemble des domaines de $\mathscr{D}$ selon les contraintes $\mathscr{C}$. Les domaines D_i sont alors réduits aux intervalles de largeur minimum contenant des valeurs cohérentes avec les contraintes de $\mathscr{P}$. On peut catégoriser le résultat d'une réduction en trois classes [112] :

- aucune solution n'est présente dans la boîte donnée en entrée de l'étape de réduction, les valeurs associées sont supprimées des domaines D_i,
- toutes les valeurs de la boîte donnée en entrée de l'étape de réduction sont cohérentes avec les contraintes du problème $\mathscr{P}$, on parle de boîte interne, ou boîte `inner`,
- certaines valeurs de la boîte donnée en entrée de l'étape de réduction sont des

solutions de $\mathscr{P}$, tandis que d'autres ne le sont pas, on parle de boîte externe, ou boîte `outer`.

Dans cette dernière éventualité, on passe à l'étape de *séparation*, qui consiste à diviser les domaines de définitions en plusieurs intervalles plus petits, afin de définir de nouvelles boîtes. Ainsi, le problème initial $\mathscr{P}$ est divisé en un certain nombre de sous-problèmes disjoints non-vides $\mathscr{P}_i$ de sorte que l'union de ces sous-problèmes contient le problème initial, c'est-à-dire que les $\mathscr{P}_i$ forment une partition de $\mathscr{P}$.

L'algorithme s'arrête lorsqu'il ne reste plus de boîte à séparer. Ceci se produit lorsque toutes les boîtes de largeur supérieure à la précision absolue Δ de l'algorithme ont été évaluées. Pour certains problèmes, il peut être difficile de calculer toutes les boîtes contenant des solutions à la précision Δ [5]. Nous définissons donc un nombre maximum de boîtes $N_b \neq \infty$. Dans ce cas, l'algorithme s'arrête lorsqu'il a calculé N_b boîtes `inner` et `outer`. L'ensemble des boîtes internes et externes forme l'ensemble fiable des solutions du problème $\mathscr{P}$.

L'algorithme détaillé peut être trouvé dans [111]. On illustre cet algorithme sur un exemple dans la figure 6.1. Nous prenons une exigence à deux variables, une entrée x et une sortie y afin de montrer graphiquement l'évolution des boîtes.

"*L'entrée x est définie sur l'intervalle* $[-1.5, 1.5]$. *La sortie y est égale au cube de l'entrée x. L'erreur relative maximum de la sortie par rapport au résultat exact est* $\varepsilon = 3 \cdot 10^{-1}$."

Cette exigence peut être reformulée mathématiquement par l'équation suivante :

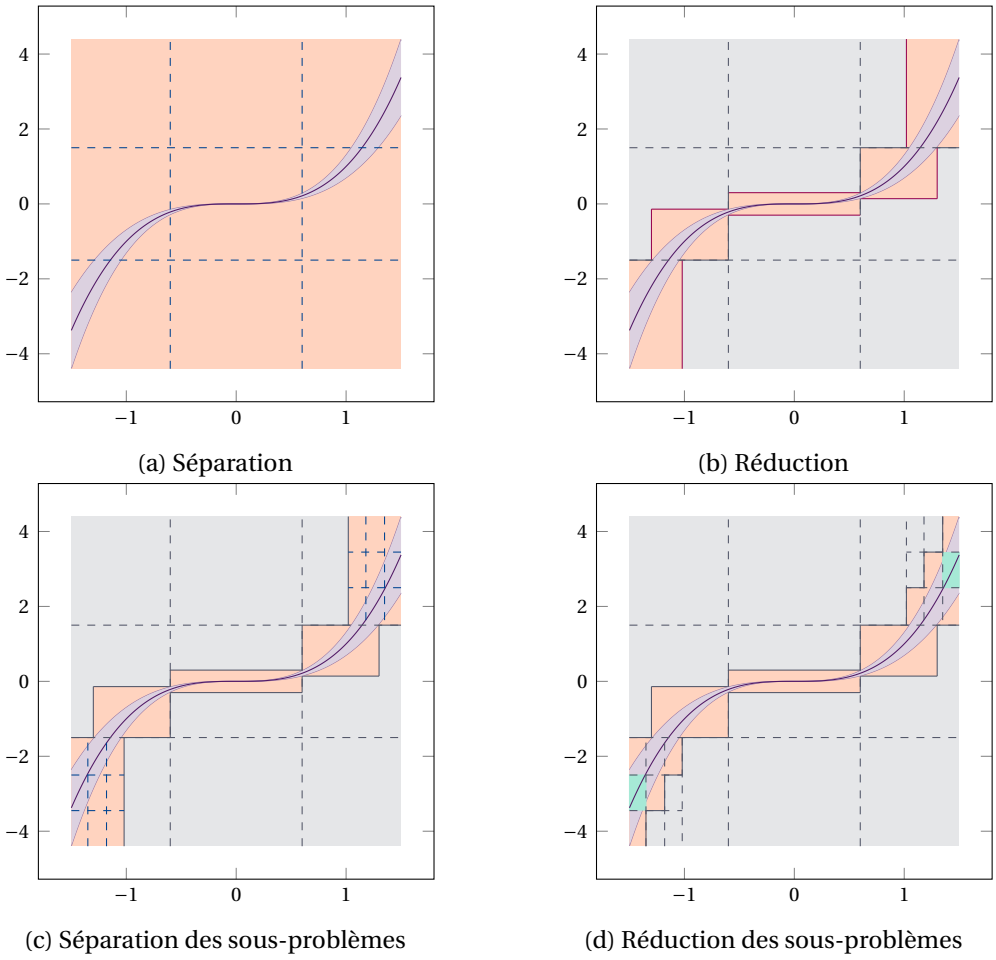
$$y = x^3 \cdot (1 \pm \varepsilon). \tag{6.1}$$

Le problème de satisfaction de contraintes sur des variables aux domaines de définition continus équivalents à cette exigence définit l'ensemble des valeurs de y qui sont des solutions de cette exigence. On définit donc le problème $\mathscr{P}$ à l'aide d'inéquations, telles que

$$\begin{aligned}
&\text{Constante :} && \varepsilon = 3 \cdot 10^{-1} \\
&\text{Variables :} && x \in [-1.5, 1.5], \qquad y \in [-\infty, \infty], \\
&\text{Contraintes :} && c_1 \; : \; y \leq \max\left(x^3 \cdot (1+\varepsilon), x^3 \cdot (1-\varepsilon)\right), \\
& && c_2 \; : \; y \geq \min\left(x^3 \cdot (1+\varepsilon), x^3 \cdot (1-\varepsilon)\right).
\end{aligned} \tag{6.2}$$

Une solution de $\mathscr{P}$ doit satisfaire les contraintes c_1 et c_2, c'est-à-dire la *conjonction* de c_1 et c_2, notée $c_1 \wedge c_2$.

La précision Δ et le nombre de boîtes N_b ne sont pas des informations relatives à la spécification fonctionnelle. Ce sont des outils faisant partie de la solution de formalisation des exigences choisies. Nous reviendrons sur la détermination de Δ pour l'algorithme de génération automatique de tests dans la section 6.3.2. Dans cet exemple, nous choisissons $\Delta = \varepsilon$.



(a) Séparation

(b) Réduction

(c) Séparation des sous-problèmes

(d) Réduction des sous-problèmes

FIGURE 6.1 – Algorithme de *branch and prune* pour la contrainte $y = x^3 \cdot (1 \pm 0.3)$

Si on résout $\mathscr{P}$ sur les réels, la première étape de réduction consiste à appliquer les contraintes c_1 et c_2 afin de transformer les domaines de x et y aux intervalles minimums contenant l'ensemble des solutions. L'intervalle de définition de y est ainsi transformé, de sorte que $y \in [-4.3875, 4.3875]$. La première boîte réduite, contenant toutes les solutions de $\mathscr{P}$ est donc $[-1.5, 1.5] \times [-4.3875, 4.3875]$. L'étape suivante est la séparation de cette boîte en plusieurs boîtes formant une partition de $\mathscr{P}$, comme illustré dans la courbe (a). Puis, dans la courbe (b), l'opération de réduction est appelée sur chaque sous-problème. Les boîtes qui ne contiennent pas de solution sont abandonnées (en gris). Les boîtes qui contenaient des solutions quant à elles de sorte que les boîtes sont de nouveau transformées en intervalles plus petits, pour former des bornes étroites (en orange). La courbe (c) illustre l'itération de l'opération de séparation sur deux sous-problèmes, et la réduction de ces boîtes est

effectuée dans la courbe (d). On observe notamment deux boîtes dont on peut affirmer avec certitude qu'elles appartiennent à l'ensemble des solutions de $\mathscr{P}$ (en vert).

Sur cette figure, on voit que l'algorithme s'arrête lorsque toutes les boîtes contenant composées de solutions du problème $\mathscr{P}$ ainsi que des valeurs qui ne sont pas cohérentes avec les contraintes (boîtes oranges, boîtes `outer`) ne sont plus divisibles. Ceci arrive lorsqu'il n'existe plus de boîte de largeur supérieure à Δ ou que le nombre de boîtes maximum N_b a été atteint.

Le but de la modélisation des exigences par un CSP est de calculer un oracle précis, qui permet d'assurer un résultat fiable. Avec cet oracle, nous souhaitons pouvoir distinguer un résultat appartenant à l'ensemble des solutions de l'exigence d'un résultat non conforme. Nous nous intéressons donc essentiellement à l'ensemble des boîtes `inner` des solutions de $\mathscr{P}$, tout en essayant de déterminer l'ensemble des boîtes `outer` le plus fin possible. La définition de la largeur acceptable pour l'ensemble des boîtes `outer` dépend de l'approche de génération de tests choisie. Calculer l'ensemble des solutions d'un CSP $\mathscr{P}$ quelconque à la précision Δ est cependant un problème difficile [5], surtout dans le cas d'exigences complètes ayant un grand nombre de variables. Nous favoriserons donc une solution de génération automatique de test n'exigeant pas une description complète de l'ensemble des solutions d'un CSP.

6.2 Mutation Testing pour la génération de tests

6.2.1 Motivations

Parmi toutes les méthodes de génération de tests et d'évaluation de la qualité d'une campagne de tests présentées dans le chapitre 3, nous distinguons deux candidats pour la génération automatique à partir de contraintes numériques. L'interprétation abstraite ainsi que les contraintes ont déjà été combinées dans de précédents travaux pour effectuer la vérification de logiciels numériques [94]. Dans cette approche, les contraintes modélisent le flot de contrôle des données par un graphe, qui représente l'espace d'exploration. Une analyse statique permet la vérification de propriétés du programme. Dans ce contexte, une approximation de l'ensemble des solutions du CSP modélisant le programme est donc nécessaire.

Le test par mutation, ou *Mutation Testing* quant à lui est une méthode permettant d'évaluer la qualité d'une campagne de tests en estimant si le critère d'adéquation des tests est atteint (définition 5). La création de fautes dans le programme permet de donner une

Opérateur	Description
`precision`	échange de la précision relative de calcul ε par une nouvelle valeur
`bounds`	échange des valeurs des bornes d'une variable par d'autres valeurs
`constant`	échange de la valeur d'une constante par une nouvelle valeur
`variable`	échange d'une variable par une autre variable
`unary`	échange d'un opérateur unaire par un autre $\left(\sqrt{}, \sin, \cos, \tan, \exp, \log\right)$
`binary`	échange d'un opérateur binaire par un autre $(+, -, \times, \div, \max, \min)$
`add`	ajout d'une instruction
`sup`	suppression d'une instruction

Tableau 6.1 – Ensemble des mutations considérées adapté de [3]

mesure de l'adéquation d'une campagne de tests à un programme [1] [83]. Dans l'étude [9], le test par mutation est utilisé sur des contraintes booléennes et entières, donc sur des domaines de définition finis afin de simuler des fautes dans le système, et ainsi générer des tests permettant de vérifier l'absence de ces fautes dans le logiciel sous test. La génération de tests à l'aide du test par mutation ne nécessite pas de modélisation complète de l'ensemble des solutions pour la génération de tests, comme nous le verrons dans les sections suivantes. Nous nous tournons vers cette approche, avec pour objectif d'effectuer une généralisation sur les variables aux domaines continus pour la génération automatique de tests fiables.

6.2.2 Principes de base

Le test par mutation est donc une méthode employée pour l'évaluation de la qualité d'une campagne de test, apportant des informations pertinentes et complémentaires avec l'évaluation par les métriques de couverture [113]. Cette méthode est initialement basée sur l'analyse de la structure d'un programme P. Soit une campagne de test T associée à la validation et vérification de P, la première étape du test par mutation consiste alors à modifier une partie contrôlée du programme P pour générer un programme P_M, qui est appelé *mutant* de P. La modification ayant permis la génération de P_M est appelée *mutation* ; l'ensemble des mutations Ω que nous considérons dans ces travaux sont décrits dans le tableau 6.1.

1. Le logiciel est le cadre applicatif le plus répandu pour le développement du test par mutation, nous parlerons ainsi de "programme" dans cette partie, avant de décrire en détails le cas d'une description fonctionnelle, notamment sur un CSP

La seconde étape consiste à exécuter la campagne de test T sur le mutant P_M, opération que nous notons (P_M, T), et à examiner le verdict. Deux résultats peuvent alors être observés :

- `killed` : (P_M, T) = `KO`, on dit que le mutant P_M est *tué* par la campagne de test T, c'est-à-dire que la campagne de test a réussi à identifier avec succès la mutation, ou la faute, effectuée sur P,
- `survived` : (P_M, T) = `OK`, on dit que le mutant P_M a *survécu* à la campagne de test T, c'est-à-dire qu'elle a échoué à identifier la mutation dans P. Cette campagne de test n'est donc pas robuste à cette mutation.

Après l'exécution de tous les mutants générés de cette façon, le testeur se retrouve avec deux informations [9]. Soient n_K et n_M tels que n_K est le nombre de mutants tués par la campagne de test T et n_M est le nombre total de mutants générés. La première information est l'*adéquation* d'une campagne de test T à un ensemble de mutations en fonction d'un ensemble de mutations donné $\mathscr{A}_{T\Omega}$. C'est la proportion du nombre de mutants qui ont été tués par la campagne de test T, définie par

$$\mathscr{A}_{T\Omega} = \frac{n_K}{n_M}. \tag{6.3}$$

La seconde information est le nombre de mutants ayant survécu à la campagne de test T. Ces mutants sont directement liés à la mutation qui a servi à leur génération, et donc à un opérateur de mutation appartenant à l'ensemble décrit dans le tableau 6.1. Le testeur a donc une information sur le type de modification ayant mené à l'échec de la vérification par le test, chaque mutation émulant une faute potentielle du logiciel.

Nous avons décrit dans les chapitres 1 et 3 les liens entre les critères d'évaluation d'une campagne de tests, les objectifs de test et les règles d'écriture des exigences fonctionnelles. Ces critères décrivent selon différents niveaux de détails le type de tests nécessaire pour former une campagne de tests adéquate. On peut regrouper ces critères en trois catégories, auxquelles nous faisons correspondre les opérateurs de mutation :

- précision : vérification de la précision algorithmique → `precision`,
- robustesse : vérification des valeurs aux limites et de la gestion des comportement anormaux → `bounds`, `add`, `sup`,
- comportement fonctionnel : vérification que le comportement fonctionnel implémenté n'a pas été altéré par des erreurs → `constant`, `variable`, `unary`, `binary`.

Ainsi, les opérateurs de mutations listés dans le tableau 6.1 permettent de vérifier l'adéquation de la campagne de test pour le programme sous test selon des critères précis, correspondant aux objectifs de test. Le test par mutation fournit ainsi un lien de *traçabilité*

entre les erreurs qui ne sont pas découvertes par la campagne de tests, soit les mutants qui ont survécu, les tests correspondants qui doivent être générés pour combler les lacunes de la campagne de test et les objectifs de test qu'ils doivent atteindre.

6.2.3 De la vérification de la qualité à la génération d'oracle de tests

Étant donné deux programmes P_1 et P_2, on dit qu'un cas de test t permet de *distinguer* les deux programmes, si les verdicts des deux campagnes de test sont différents [114]

$$(P_1, t) \neq (P_2, t). \tag{6.4}$$

Soient un programme P et un mutant P_M généré à partir de ce programme. On dit qu'un cas de test t est *pertinent* s'il permet de distinguer le programme de son mutant en tuant P_M [9], c'est-à-dire si

$$(P, t) = \texttt{OK}, \qquad (P_M, t) = \texttt{KO}. \tag{6.5}$$

Le but de la génération de tests à partir du test par mutation est de générer des tests pertinents, c'est-à-dire des tests permettant de tuer les mutants créés à partir de règles de mutations. Le critère d'adéquation des tests est atteint lorsque $\mathscr{A}_{T\Omega} = 1$, c'est-à-dire lorsque tous les mutants ont été tués par la campagne de test générée, ainsi tous les critères d'évaluation sont remplis. L'algorithme 6.1 décrit les étapes de la génération de tests à partir de P, un programme sous test et de $\Omega = \{\omega_1, \ldots, \omega_{n_\Omega}\}$, un ensemble fini de mutations définies préalablement. L'algorithme renvoie un ensemble de tests T sous la forme d'un ensemble de séquences de stimulations, et les comportements attendus associés pour vérifier le résultat des tests.

Algorithme 6.1 : Génération automatique de test à partir du test par mutations

Données : P – programme sous test
Ω – ensemble de mutations
Résultat : T – ensemble de tests

```
1 T ← ∅
2 pour ω dans Ω faire
3 |   M ← generateur_mutants(P,ω)
4 |   pour P_M dans M faire
5 |   |   si (P_M, T) = OK alors
6 |   |   |   T ← T ⋃ generateur_tests(P, P_M)
7 |   |   fin
8 |   fin
9 fin
```

Cet algorithme est constitué de deux boucles de taille finie. Il repose sur les résultats des deux procédures `generateur_mutants` et `generateur_tests`. La première génère un ensemble de mutants M à partir du programme P et d'une mutation ω. Prenons par exemple l'exécution de la procédure `generateur_mutants` sur le programme P par la mutation de la mutation ω = `variable` tel que

P :
```
double a=2000.0, c=-2.0, r;
_Decimal64 b=0.001D;
r = __builtin_fma(a,(double)b,c);
```

L'ensemble des mutants générés est donc $M = \{P_{M_1}, \ldots, P_{M_5}\}$, de sorte que l'instruction affectant à `r` la valeur du FMA en double (définition chapitre 5) est modifiée de la façon suivante dans chaque mutant :

P_{M_1} : `r = __builtin_fma(a,c,(double)b);`

P_{M_2} : `r = __builtin_fma(c,a,(double)b);`

P_{M_3} : `r = __builtin_fma(c,(double)b,a);`

P_{M_4} : `r = __builtin_fma((double)b,a,c);`

P_{M_5} : `r = __builtin_fma((double)b,c,a);`

Après la génération des mutants, l'algorithme parcourt l'ensemble des mutants M et la campagne de test est effectuée sur ces mutants afin de déterminer si elle est robuste aux erreurs simulées par ces mutations. Dans notre exemple, nous effectuons donc l'opération (P_{M_i}, T), $i = \{1, \ldots, 5\}$. Le premier mutant à être testé de cette façon dans cet exemple est P_{M_1}. Comme la campagne de test est vide au début de l'algorithme, on a que (P_{M_1}, T) = `OK`, c'est-à-dire que P_{M_1} a survécu à la campagne de test. Il faut donc générer un test pour enrichir la campagne de test T afin de tuer ce mutant. C'est le but de la procédure `generateur_tests`, qui utilise la résolution de contraintes afin de trouver les points *discriminants* du programme, c'est-à-dire les résultats qui diffèrent entre le programme et le mutant pour une même séquence de stimulation. Une procédure de génération de tests est introduite dans [9] à partir de CSP sur des domaines discrets et de test par mutation. Nous proposons une généralisation de cet algorithme à partir de CSP sur des domaines continus pour la génération de tests numériques fiables dans la section 6.3.

Un fois l'ensemble de test T enrichi d'un nouveau cas de test, l'algorithme continue la boucle sur les mutants et effectue la campagne de tests sur le second mutant P_{M_2}. On a alors que (P_{M_2}, T) = `KO`, car le test généré pour P_{M_1} permet de tuer le mutant P_{M_2}. La procédure de génération de tests n'est donc pas appelée à chaque tour, et plus l'ensemble de tests T sera complet, moins de mutants survivrons à l'étape d'évaluation de la campagne de tests selon un nouveau mutant.

L'algorithme se termine lorsque toutes les mutations et tous les mutants générés ont été explorés. La complexité de l'algorithme dépend donc du nombre de mutations n_ω qui est borné (dans le tableau 6.1 nous avons défini $n_\omega = 8$ mutations), et du nombre maximum de mutants générés par une mutation $n_{M^{\max}}$. Ce dernier dépend du nombre d'éléments ciblés par les mutations (variables, opérations, etc.) et peut être très grand relativement à n_ω. Il y a donc $n_\Omega \cdot n_{M^{\max}}$ tours de boucles. De plus, la campagne de tests T est effectuée à chaque tour pour évaluer sa robustesse au mutant sélectionné. Dans le pire cas, un test est généré pour tuer chaque mutant, la campagne de tests T contient ainsi au plus $n_\Omega \cdot n_{M^{\max}}$ éléments. La procédure de génération de tests est donc appelée autant de fois. Nous étudierons sa complexité dans la section 6.3, que nous notons $\tau = \mathcal{O}\left(\texttt{generateur\_tests}(\texttt{P}, \texttt{P}_\texttt{M})\right)$. Ainsi, on peut estimer la complexité de cet algorithme dans le pire cas comme

$$\mathcal{O}\left(\tau \cdot (n_\omega \cdot n_{M^{\max}})^2\right). \tag{6.6}$$

6.2.4 Optimisation de la génération de mutants

Le test par mutation peut ainsi être utilisé pour la génération automatique de tests, fournissant une campagne de test associée à un critère d'évaluation permettant d'attester de sa qualité. Cette approche systématique comporte cependant plusieurs problèmes d'efficacité. Pour que cette approche puisse être envisagée dans un contexte industriel, il faut qu'elle soit compétitive par rapport aux méthodes actuellement en place (génération de tests assistée, mais pas automatisée).

Un grand nombre de mutants générés équivaut à potentiellement un grand nombre de tests, augmentant ainsi le temps d'exécution d'une campagne de test [3]. De plus, la présence de mutants dont le comportement ne se distingue pas de celui du programme sous test impacte fortement la mesure de l'adéquation de la campagne de test générée T [114]. Les objectifs des différentes optimisations de cette approche sont d'affiner la génération des mutants, afin d'optimiser la génération de tests [115]. Créer le plus petit ensemble de tests nécessaire à la vérification du programme étant un problème NP-difficile [9], le but est d'essayer de réduire la taille de la campagne de tests au maximum.

Détection des mutants équivalents Un mutant P_M est dit *équivalent* au programme P lorsque, sans être identique à P, le mutant P_M se comporte systématiquement de la même façon que P pour une même séquence de stimulations [114]. Autrement dit, il n'existe pas de point discriminant permettant de distinguer le mutant P_M du programme P. La présence de mutants équivalents dans l'ensemble des mutants M de l'algorithme 6.1 implique que le verdict de la campagne de test est $(P_M, T) = \texttt{OK}$, et la procédure de génération de tests est donc exécutée. Cette procédure ne peut cependant pas trouver de point discriminant pour tuer ce mutant, et aucun test n'est généré.

La mesure de l'adéquation $\mathscr{A}_{T\Omega}$ est directement impactée par la présence de mutants équivalents. Il est impossible d'avoir l'adéquation maximum de la campagne de test pour un programme si des tests équivalents à ce programme ont été générés. Afin de rendre l'adéquation robuste à ce phénomène, on veut quantifier le nombre de mutants équivalents n_E, et ainsi modifier l'équation (6.7) telle que

$$\mathscr{A}_{T\Omega} = \frac{n_K}{n_M - n_E}. \tag{6.7}$$

La détection de mutants équivalents est un problème de décision analogue au problème d'équivalence de programmes, qui est connu pour être indécidable dans le cas général [116, 114]. Plusieurs études se sont penchées sur le problème de détection de mutants équivalents et aux méthodes permettant d'améliorer la génération de mutants [117, 114, 118]. Notre approche combinant le test par mutation à la programmation par contrainte sur des domaines continus doit donc être robuste à la présence de mutants équivalents pour la génération de tests maximisant la mesure de l'adéquation.

Sélection des mutations Une seconde méthode pour optimiser la génération de tests grâce au test par mutations est d'optimiser la génération des mutants, en sélectionnant des mutations générant moins de mutants redondants tout en conservant la même efficacité à simuler des erreurs. Plusieurs méthodes ont été étudiées pour optimiser la génération de mutants. La méthode d'échantillonnage des mutations [119] est une méthode heuristique, sélectionnant les mutations selon une fonction d'évaluation de l'efficacité des mutants. Cette dernière est déterminée grâce à une pondération des mutations, en fonction du nombre de mutants équivalents générés ainsi que la capacité des mutants à générer un ensemble minimal de tests pertinents. Une autre méthode permettant d'optimiser la génération de mutants est la définition de mutations de haut niveau [115].

Parallélisation Une autre approche d'optimisation est d'utiliser des méthodes de calcul hautes performances pour la génération de l'ensemble de mutants. L'exécution des méthodes de test par mutations sur des architectures parallèles permet de réduire le coût de la génération de mutants sans avoir favorisé un type de mutations particulier [120, 121].

6.3 Algorithme de génération automatique de tests

Nous proposons une nouvelle approche de génération automatique de tests utilisant le test par mutations sur un CSP continu. Cette approche se distingue des travaux effectués sur la génération de tests sur des domaines discrets [9] par la redéfinition des contraintes adaptées sur les domaines continus. Nous présentons dans un premier temps une approche générale, pour ensuite tenter d'optimiser la qualité des tests générés.

6.3.1 Test par mutation sur un CSP continu : une première approche

Reprenons de $\mathscr{P}$, le CSP décrit dans l'équation (6.2) pour illustrer la procédure de génération de tests sur des contraintes continues. Un mutant, suivant la mutation `constant` pourrait être de changer la constante 3 dans la puissance de X, de sorte que

$$y \;=\; x^2 \cdot (1 \pm \varepsilon)\,. \tag{6.8}$$

Le problème de satisfaction de contraintes $\mathscr{P}_M$ sur des variables aux domaines de définitions continus représentant ce mutant est donc le suivant :

$$\begin{array}{ll} \text{Constante :} & \varepsilon \;=\; 3 \cdot 10^{-1} \\ \text{Variables :} & x \in [-1.5, 1.5]\,, \qquad y \in [-\infty, \infty]\,, \\ \text{Contraintes :} & m_1 \;:\; y \;\leq\; \max\left(x^2 \cdot (1+\varepsilon)\,, x^2 \cdot (1-\varepsilon)\right), \\ & m_2 \;:\; y \;\geq\; \min\left(x^2 \cdot (1+\varepsilon)\,, x^2 \cdot (1-\varepsilon)\right). \end{array} \tag{6.9}$$

Le but de la génération de tests à partir de ce mutant est donc d'assurer que la fonctionnalité implémentée dans le programme sous test n'est pas la fonction carré. Pour ce faire, la procédure `generateur_tests`$(\mathscr{P}, \mathscr{P}_M)$ doit ainsi trouver un point discriminant $\left(x_t, y_t\right)$ permettant de distinguer $\mathscr{P}$ de $\mathscr{P}_M$. Nous cherchons donc un point solution de $\mathscr{P}$ qui n'est pas solution de $\mathscr{P}_M$, tel que $\left(x_t, y_t\right)$ satisfait les contraintes $\{c_1 \wedge c_2\}$ et ne satisfait pas les contraintes $\{m_1 \wedge m_2\}$, c'est-à-dire qui satisfait la contrainte opposée de m_1, notée $\neg m_1$ ou la contrainte inverse de m_2, notée $\neg m_2$. On peut donc définir les points distinguant $\mathscr{P}$ de $\mathscr{P}_M$ par un nouveau problème $\mathscr{P}_T$ formé de deux CSP $\mathscr{P}_{T_1}$ et $\mathscr{P}_{T_2}$. Une solution de $\mathscr{P}_T$ doit être conformes aux contraintes de $\mathscr{P}_{T_1}$ ou aux contraintes de $\mathscr{P}_{T_2}$, c'est-à-dire la *disjonction* de ces contraintes. Soit deux contraintes a_1 et a_2, la disjonction de ces contraintes est notée $a_1 \vee a_2$.

On définit donc $\mathscr{P}_{T_1}$ tel que

$$\begin{array}{l} c_1 \;:\; y \;\leq\; \max\left(x^3 \cdot (1+\varepsilon)\,, x^3 \cdot (1-\varepsilon)\right), \\ c_2 \;:\; y \;\geq\; \min\left(x^3 \cdot (1+\varepsilon)\,, x^3 \cdot (1-\varepsilon)\right), \\ \neg m_1 \;:\; y \;\geq\; \max\left(x^2 \cdot (1+\varepsilon)\,, x^2 \cdot (1-\varepsilon)\right), \end{array} \tag{6.10}$$

et $\mathscr{P}_{T_2}$ tel que

$$\begin{array}{l} c_1 \;:\; y \;\leq\; \max\left(x^3 \cdot (1+\varepsilon)\,, x^3 \cdot (1-\varepsilon)\right), \\ c_2 \;:\; y \;\geq\; \min\left(x^3 \cdot (1+\varepsilon)\,, x^3 \cdot (1-\varepsilon)\right), \\ \neg m_2 \;:\; y \;\leq\; \min\left(x^2 \cdot (1+\varepsilon)\,, x^2 \cdot (1-\varepsilon)\right). \end{array} \tag{6.11}$$

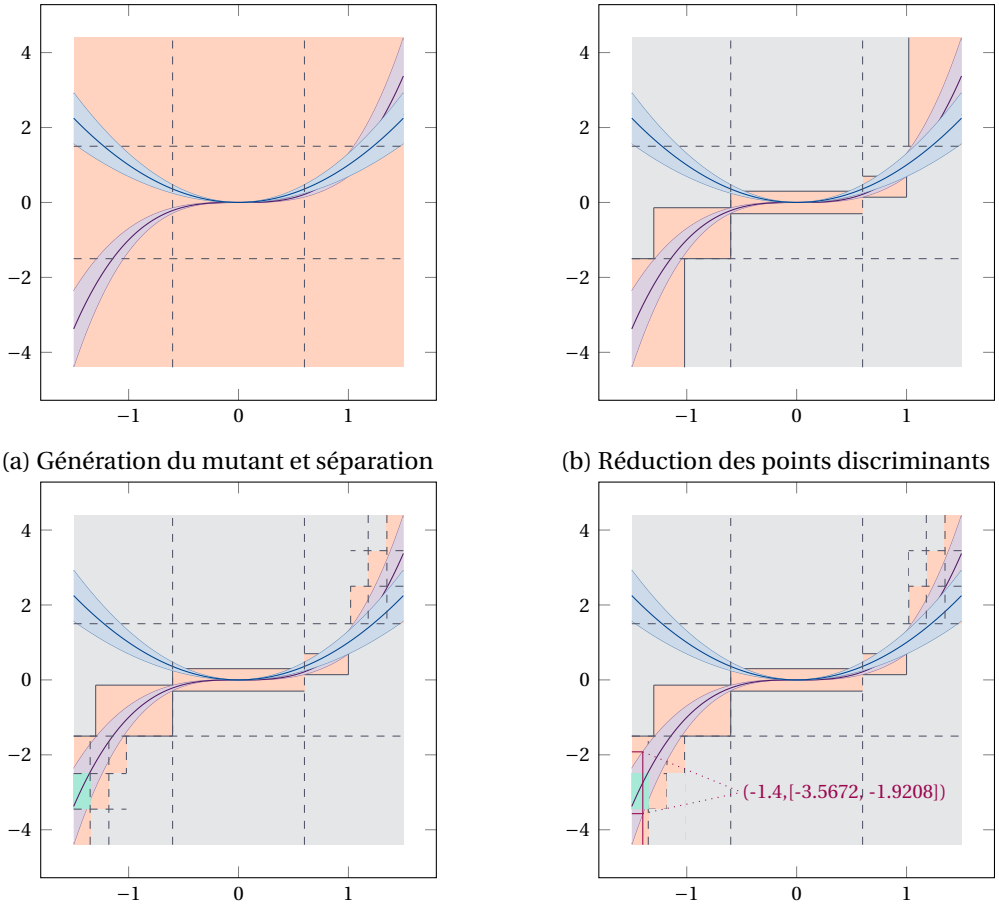
Il n'existe pas d'inégalités strictes pour les CSP sur des contraintes continues, car ceux-ci déterminent l'ensemble de solutions comme une sur-approximation fiable dans laquelle

aucune solution n'est perdue [6]. Les mutants opposés sont donc définis à l'aide d'inégalités larges.

Le terme général du CSP continu $\mathscr{P}_T$ définissant les points permettant de discriminer une exigence (composée de l'affectation d'une variable) de son mutant est donc composée des contraintes suivantes :

$$(c_1 \wedge c_2 \wedge (\neg m_1)) \vee (c_1 \wedge c_2 \wedge (\neg m_2)) \,. \tag{6.12}$$

L'algorithme de génération de tests à partir des CSP $\mathscr{P}_{T_1}$ et $\mathscr{P}_{T_2}$, qui décrivent les points de test pertinent pour le CSP $\mathscr{P}$ (fonction cube) et son mutant $\mathscr{P}_M$ (fonction racine carrée) est illustré dans la figure 6.2.



(a) Génération du mutant et séparation

(b) Réduction des points discriminants

(c) Détermination d'une boîte `inner`

(d) Sélection d'un cas de test

FIGURE 6.2 – Exemple de test par mutation sur un CSP $y = x^3 \cdot (1 \pm 0.3)$, avec le mutant $y = x^2 \cdot (1 \pm 0.3)$

Les trois premières étapes sont équivalentes aux étapes de l'algorithme *branch and prune* dont le principe est décrit dans la figure 6.1. Cependant, la génération de tests à partir de contraintes se distingue de cet algorithme. Celui-ci qui a pour but de trouver *toutes* les solutions conformes à un CSP à une précision Δ, la précision absolue bornant la largeur minimum que peuvent avoir les boîtes. Le but de l'algorithme de génération de tests est quant à lui de trouver *une* solution satisfaisant les contraintes décrites dans l'équation (6.12). Il s'arrête ainsi dès que la première boîte `inner` est trouvée au lieu d'énumérer toutes les solutions. La première boîte `inner` de la figure 6.2 est calculée dans l'étape (c) telle que $\mathbf{x}_t \times \mathbf{y}_t = [-1.35, -1.5] \times [-2.5, -3.15]$.

Ainsi, dans le pire des cas l'algorithme peut avoir à visiter toutes les boîtes `outer` dont la taille est supérieure à Δ avant de renvoyer une boîte `inner` satisfaisant ces contraintes, ou même ne pas trouver de solution conforme avec la précision Δ. Nous nous occupons des détails de l'algorithme dans la section 6.3.2, et nous verrons cependant dans les expérimentations présentées dans la section 6.4.3 qu'une boîte `inner` peut être déterminée en un temps raisonnable (≈ 0.5 secondes) dans le cas général.

Dans la boîte $\mathbf{x}_t \times \mathbf{y}_t$, l'intervalle $\mathbf{y}_t$ définit un ensemble de valeurs attendues discriminantes et conformes aux contraintes. L'intervalle $\mathbf{x}_t$ définit un ensemble de stimulations permettant d'atteindre ces valeurs de l'intervalle $\mathbf{y}_t$ de façon fiable, c'est-à-dire que pour chacune des valeurs de $\mathbf{x}_t$, une valeur associée existe dans l'intervalle $\mathbf{y}_t$. Déterminer un cas de test revient à choisir une séquence de stimulations, c'est-à-dire dans notre exemple, d'affecter une valeur à x_t. La stimulation choisie pour x_t dans figure 6.2 est ainsi $x_t = -1.4 \in \mathbf{x}_t$. Nous pouvons ainsi déterminer l'intervalle y_t déterminant l'ensemble des valeurs pour lesquelles le programme sous test sera conforme à l'exigence décrite par $\mathscr{P}$. Les bornes de y_t calculées avec une précision infinie sont donc exactement $y_t = [-3.5672, -1.9208]$. Cette boîte est une solution de $\mathscr{P}$. Elle permet de valider le comportement fonctionnel de cette exigence de façon fiable grâce à l'utilisation de l'arithmétique d'intervalles en discriminant le mutant $\mathscr{P}_M$ dont le comportement attendu pour une stimulation similaire est le point $y_{t_M} = [1.372, 2.548] \neq y_t$.

6.3.2 Algorithme

L'algorithme 6.2 décrit la procédure de génération de cas de test à partir d'un CSP décrivant les exigences numériques $\mathscr{P}$, et un mutant $\mathscr{P}_M$.

Le problème $\mathscr{P}$ décrit le comportement des variables $y_1, \ldots, y_w$ à partir des entrées $x_1, \ldots, x_v$. Le mutant $\mathscr{P}_M$ est généré à partir d'une des mutations `precision`, `constant`, `bounds`, `add`, `sup`, `variable`, `unary`, ou `binary`. On divise ces mutations en deux catégories pour la détermination du problème $\mathscr{P}_T$ permettant de définir les points discriminants pour la génération de cas de test.

Algorithme 6.2 : Procédure de génération de tests

Données : Exigence – $\mathscr{P} = \{\mathscr{X} = \{x_1, \ldots, x_v, y_1, \ldots, y_w\}, \mathscr{D}, \mathscr{C}\}$
Mutant – $\mathscr{P}_M = \{\mathscr{X}_M, \mathscr{D}_M, \mathscr{C}_M = \{m_1, m_2\}\}$
Précision – ε
Nombre de boîtes – N_b
Résultat : Cas de test – $t = \{t_{x_1}, \ldots, t_{x_v}, \mathbf{t}_{y_1}, \ldots, \mathbf{t}_{yw}\}$

1 $t \leftarrow \emptyset$
2 **pour** *m dans* $\mathscr{C}_M$ **faire**
3 $\quad \Delta \leftarrow \varepsilon$
4 $\quad \mathscr{P}_T \leftarrow$ `gen_csp_test`$(\mathscr{P}, \mathscr{P}_M, m)$ ▷ Voir cas 1 et 2
5 $\quad$ **tant que** $\Delta \geq \varepsilon^2$ **faire**
6 $\quad\quad r \leftarrow$ `first_inner_box`$(\mathscr{P}_T, \Delta, N_b)$
7 $\quad\quad$ **si** $r = \mathbf{x}_1^r \times \cdots \times \mathbf{x}_v^r \times \mathbf{y}_1^r \times \cdots \times \mathbf{y}_w^r$ ▷ Première boîte `inner`
8 $\quad\quad$ **alors**
9 $\quad\quad\quad t \leftarrow$ `pick_test`$(\mathscr{P}, r)$
10 $\quad\quad\quad$ Renvoyer t
11 $\quad\quad$ **fin**
12 $\quad\quad$ **si** $r =$ *"Pas de solution dans la boîte initiale"* ▷ Pas de solution à $\mathscr{P}_T$
13 $\quad\quad$ **alors**
14 $\quad\quad\quad$ Aller à la ligne 2
15 $\quad\quad$ **fin**
16 $\quad\quad \Delta \leftarrow \Delta \times 10^{-1}$ ▷ Pas de solution à la précision Δ
17 $\quad$ **fin**
18 **fin**
19 $t \leftarrow$ "Pas de test pour $\mathscr{P}_M$"
20 Renvoyer t

Cas 1 Dans le cas des mutations `add`, `sup`, `variable`, `unary`, et `binary`, le mutant $\mathscr{P}_M$ est défini par les mêmes variables que le problème de base. Les modifications effectuées par ces mutations ont un effet sur les contraintes m_1 et m_2, mais ne modifient pas $\mathscr{X}$ et $\mathscr{D}$. Ainsi, si on note $c_1, \ldots, c_m$ les contraintes $\mathscr{C}$ décrivant le problème $\mathscr{P}$, alors, la description du problème de génération de test $\mathscr{P}_T$ revient à inverser cette contrainte, comme dans les équations (6.10) et (6.11).

Cas 2 Dans le cas des mutations `precision`, `constant`, `bounds`, la description du problème $\mathscr{P}_T$ nécessite une étape supplémentaire. Les mutations `precision` et `constant` modifient toutes deux une constante (la précision ε étant définie comme une constante). La constante modifiée ε_M ou C_M est donc ajoutée à l'ensemble des constantes de $\mathscr{P}_T$, ainsi que toutes les constantes de $\mathscr{P}$. Pour la mutation `bounds`, les bornes d'une variable sont

modifiées. La mutation consiste donc à ajouter une variable x_M dans $\mathscr{X}_T$ et son domaine associé d_M dans $\mathscr{D}_T$, ainsi que toutes les variables du problème $\mathscr{P}$. Les mutants m_1 et m_2 sont donc les contraintes dont les constantes/variables ont été mutées. Ils sont ensuite inversés pour être ajoutés à $\mathscr{P}_T$ avec l'ensemble des contraintes de $\mathscr{P}$.

L'algorithme fait ensuite appel au solveur de contraintes, sous la forme de la procédure `first_inner_box`, pour trouver la première boîte `inner`, solution du problème $\mathscr{P}_T$, à la précision Δ. Cette précision est un paramètre clef de l'algorithme, elle détermine la taille des plus petites boîtes calculées par le solveur de contrainte avant que celui-ci ne s'arrête à chercher des solutions. Nous n'avons pas de terme général permettant de déterminer Δ en fonction de la complexité des contraintes et des variables utilisées. En l'absence de formalisme pour Δ, nous faisons évoluer Δ de $\Delta_{\max}$ à $\Delta_{\min}$. Ceci nous permet de trouver une solution rapidement dans le cas où une solution existe pour la précision $\Delta_{\max}$, et nous pouvons faire une recherche de solution plus précise dans le cas où aucune solution n'a été trouvé rapidement. Nous choisissons dans l'algorithme d'initialiser Δ à ε, même s'il est possible de trouver des cas pour lesquels $\Delta > \varepsilon$ permet de trouver des solutions du problème sous forme de boîtes `inner`. Nous estimons cependant qu'un compromis satisfaisant en termes de solutions trouvées et de temps de calcul est de faire évoluer Δ dans l'intervalle $[\varepsilon^2, \varepsilon]$, pour $\varepsilon \leq 1$. Pour assurer la terminaison de la procédure de résolution de contraintes en un temps raisonnable (nous nous donnons une borne de 2 minutes pour une exigences), il est nécessaire de donner au solveur de contraintes un nombre maximum de boîtes N_b. Ainsi, les cas d'arrêt du solveur sont les suivants :

- une boîte `inner` a été trouvée,
- aucune solution n'existe pour distinguer le problème $\mathscr{P}$ du mutant $\mathscr{P}_M$ par l'inversion de la contrainte m_i, $i = \{1,2\}$,
- le solveur n'a pas su trouver de boîte `inner` alors que des solutions existent, car la précision Δ est plus grande que la plus grande des boîtes `inner` existant, ou l'algorithme a visité la boîte numéro N_b avant de trouver une boîte `inner` satisfaisant ce problème.

Dans ce dernier cas, l'algorithme augmente la précision Δ dans l'intervalle que nous avons défini ou, si la précision minimale a été atteinte, l'algorithme arrête sa recherche de solution au problème $\mathscr{P}_T$ pour la contrainte m courante.

Problème des mutants équivalents Dans le cas où aucune solution n'existe pour distinguer les exigences du mutant, on a un mutant dont le comportement décrit un sous-ensemble des solutions. Ce n'est pas un mutant équivalent au système, mais il peut être considéré comme tel, et est ajouté au nombre des mutants équivalents pour la mesure

de l'adéquation. De cette façon, un certain nombre de mutants équivalents sont détectés automatiquement.

Les mutants équivalents sont problématiques dans le cas où l'ensemble de solutions décrit par le mutant est exactement le même que celui des exigences. Par exemple, prenons l'exigence $c \ : \ y = (a+b)(1 \pm \varepsilon)$ et le mutant $m \ : \ y = (b+a)(1 \pm \varepsilon)$, généré à partir de la mutation `variable`. Ainsi une partie du CSP définissant les points discriminants est le suivant :

$$\begin{aligned} c_1 \ : \ y \ &\leq \ \max((a+b)\cdot(1+\varepsilon),(a+b)\cdot(1-\varepsilon)), \\ c_2 \ : \ y \ &\geq \ \min((a+b)\cdot(1+\varepsilon),(a+b)\cdot(1-\varepsilon)), \\ \neg m_1 \ : \ y \ &\geq \ \max((b+a)\cdot(1+\varepsilon),(b+a)\cdot(1-\varepsilon)), \end{aligned} \tag{6.13}$$

On voit facilement que la solution est $y \ = \ \max((a+b)\cdot(1+\varepsilon),(a+b)\cdot(1-\varepsilon))$. Il est cependant impossible que cette contrainte produise une boîte `inner`, et parcourir toutes les boîtes contenant encore au moins une solution est la seule façon qu'à le solveur de contraintes de s'en assurer. La seule solution que nous avons actuellement pour nous occuper de ce cas de figure est d'arrêter la recherche d'une solution et de laisser un message d'erreur significatif au testeur qui pourra déterminer si le mutant est équivalent ou non aux exigences. Une des perspectives de nos travaux serait d'effectuer une analyse à plus haut niveau de la sémantique des mutants et des exigences afin de repérer automatiquement ce genre de mutant équivalent.

6.4 Prototypage d'une solution

Le but de ce développement est de montrer la faisabilité d'une telle approche pour la génération automatique de test selon des critères de performances et de fiabilité. Nous décrivons dans un premier temps l'architecture de la solution pour un solveur de contraintes quelconques, puis nous présentons le solveur de contraintes choisi : `RealPaver` [5, 111, 6]. Enfin, nous analysons cette solution par des tests de performance de cette approche sur un ensemble d'exemples représentatifs et nous présenterons les perspectives de ces travaux.

6.4.1 Architecture de la solution

La solution développée est un prototype, permettant d'évaluer la faisabilité de cette méthode de génération automatique de tests dans l'objectif de la déployer sur une large échelle. Les caractéristiques recherchées sont la vitesse d'exécution et la fiabilité des résultats. Lorsque les tests ne peuvent pas être générés en un temps raisonnable (nous prenons la limite arbitraire de 20 secondes par cas de test), un rapport de génération doit être créé détaillant précisément à partir de quel mutant le test n'a pas pu être généré afin de permettre au testeur d'effectuer l'analyse manuellement.

Nous utilisons un solveur de contraintes numériques afin afin de générer les tests. Le prototype de solution est développé à l'aide d'un langage fiable et offrant des fonctionnalités de manipulation de fichiers et d'analyse syntaxique de haut niveau. C'est pourquoi nous choisissons d'effectuer nos développements en Julia [2] [122, 123] version 1.0.2 (2018-11-08). Julia est un langage dynamique flexible approprié pour l'informatique scientifique et numérique, avec des performances comparables à celles des langages statiques traditionnels. Julia offre une facilité et une expressivité pour l'informatique numérique de haut niveau, tout en assurant une arithmétique fiable à l'aide des `BigFloat` (reposant sur MPFR **??**) et son paquet d'arithmétique d'intervalle.

L'architecture du prototype proposé développant la solution présentée dans les algorithmes 6.1 et 6.2 est décrite dans la figure 6.3. Cette architecture est composée en deux parties : le générateur de test écrit en Julia d'un côté fait appel à un solveur de contrainte, ici `RealPaver` (présenté dans la section 6.4.2). L'algorithme commence avec les exigences formalisées sous forme d'un problème de satisfaction de contraintes $\mathscr{P}$, traduit manuellement depuis des exigences textuelles. Cette traduction manuelle est représentée par la flèche en pointillés. Toutes les autres flèches illustrent les transformations automatiques des données.

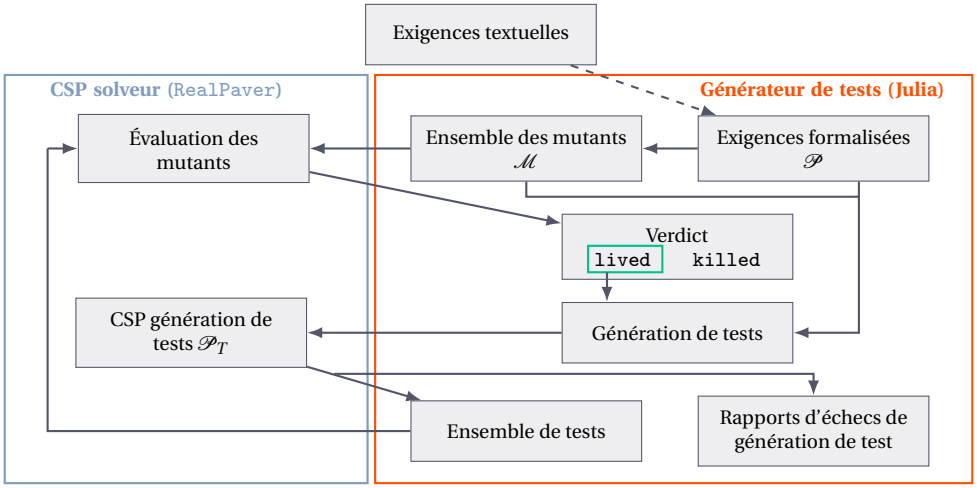


FIGURE 6.3 – Architecture de la solution implémentée avec le solveur `RealPaver`

Dans cette configuration, le générateur de test est interfacé avec un seul solveur de contraintes. Nous souhaitons toutefois mettre en avant la versatilité de cette solution : il est facile de changer l'interface pour permettre d'utiliser un autre solveur de contraintes en fonction du type d'exigences traitées. Cette modularité permet également de choisir le

2. `https://julialang.org/`

solveur de contraintes le plus performant en fonction des performances souhaitées (vitesse, fiabilité de la campagne de test en ayant la plus grande adéquation possible, complexité des exigences).

6.4.2 RealPaver : un solveur de contraintes numériques

Le solveur de contraintes pour effectuer la preuve de concept de ce prototype de générateur de tests est `Realpaver` [3] [5, 111]. `Realpaver` est un solveur de contraintes sur des variables discrètes ou continues représentées par des intervalles. Ce logiciel permet d'effectuer la modélisation et la résolution de systèmes non-linéaires, en calculant des approximations fiables de l'ensemble des solutions. Les systèmes sont donnés par des ensembles d'équations ou d'inégalités sous forme de contraintes sur des variables entières et réelles.

`Realpaver` permet de calculer l'ensemble des solutions d'un CSP, c'est-à-dire chaque boîte `inner` et `outer`. Dans notre prototype, nous cherchons uniquement la première boîte `inner` que peut renvoyer le logiciel. Nous modifions donc le fichier *search.c* des sources de `Realpaver` afin que ce dernier s'arrête après avoir renvoyé la première boîte `inner` trouvée. Ainsi, lors de la génération de tests à partir d'un ensemble de tests vide, le premier CSP en `Realpaver` à être généré dans cet algorithme est le CSP de génération de tests $\mathscr{P}_T$ comme illustré dans la figure 6.3. Si la première mutation à être considérée est la mutation `variable`, le $\mathscr{P}_T$ est alors de la forme suivante :

```
Branch precision = 10.0e-7,
number = 10000000;
Time = 20000;
Constant
eps = 10.0e-7, p = 3;
Variables
real x in [-300, 300], real y in [-oo, +oo];
Constraints
y <= max(( x ^ p )* ( 1 + eps ), ( x ^ p )* ( 1 - eps )),
y >= min( ( x ^ p )* ( 1 + eps ), ( x ^ p )* ( 1 - eps )),
x <= min(( x ^ p )* ( 1 + eps ), ( x ^ p )* ( 1 - eps ));
```

On retrouve dans ce fichier les informations de précision dans la variable `precision`, qui correspond à la précision du solveur Δ, ainsi que le nombre maximum de boîtes avec lesquelles le solveur va rechercher les solutions, représenté par la variable `n`. Le résultat renvoyé par `Realpaver` modifié après l'évaluation de ce CSP est

```
RealPaver v. 0.4 (c) LINA 2004
INITIAL BOX
```

3. http://pagesperso.lina.univ-nantes.fr/~granvilliers-l/realpaver/

```
  x in [-300 , +300]
  y in ]-oo , +oo[

INNER BOX 177149
  x in [299.9999994971087 , 300]
  y in [27000026.86421769 , 27000026.86421847]

  precision: 7.68e-07, elapsed time: 2,993 ms

END OF SOLVING
  Property:     reliable process (no solution is lost)
  Elapsed time: 2,993 ms
```

Les points fort de `Realpaver` sont la simplicité de représentation d'un CSP ainsi que la fiabilité des réponses. La résolution de problèmes difficiles peut cependant être lente, malgré la limite sur le nombre de boîtes allouées à la génération d'une solution. Lorsque le domaine de définition des variables est trop grand, ou qu'il y a beaucoup de variables, l'espace de recherche devient lui-même très grand, et déterminer une boîte `inner` jusqu'à la précision Δ peut devenir très lent. Dans ce cas, il faut reconsidérer le CSP, en augmentant la précision Δ ou en réduisant le domaine de définition des variables. Une perspective intéressante serait d'optimiser cette partie de l'algorithme afin de faciliter le travail du testeur en réduisant l'espace de recherche. Dans l'état, nous choisissons de borner le temps de résolution à 20 secondes à l'aide du paramètre `Time`.

6.4.3 Résultats expérimentaux

Le prototype de solution a été testé pour un sous-ensemble de 100 exigences, représentatives des CSP possibles, simples ou complexes. Ces CSP ont été générés pour une précision $\varepsilon \in [10^{-3}, 10^{-12}]$ et pour un nombre maximum de 7 variables et 10 opérations. Nous étudions les résultats pour quatre exigences représentatives d'une variété de fonctionnalités, dont des exigences étudiées précédemment dans ce chapitre et cette thèse, comme le cube ou la définition du débit d'air prélevé. Les résultats sont synthétisés dans le tableau 6.2 et la figure 6.4. Ces tests ont été réalisés pour un sous-ensemble des mutations décrites dans le tableau 6.1, soit les mutations `constant`, `unary`, `binary`, `variables` et `precision`. Les autres mutations sont en cours de développement.

La figure 6.4 illustre le nombre de mutants tués en fonction des tests générés pour les 4 exigences décrites dans le tableau 6.4, ainsi que la cinquième ligne décrivant la moyenne pour les 100 exigences générées, qui correspond à la cinquième barre de l'histogramme. On remarque tout d'abord dans ce tableau que l'étape de génération des tests est généralement la plus lente, quand bien même le temps d'exécution de `RealPaver` est limité par le paramètre `Time`. Par exemple dans la seconde exigence, on observe que le temps

Exigence	**Génération des mutants**	**Génération des tests**
$y = x^3 \cdot (1 \pm 10^{-7})$	0.65 s	0.6 s
$y = 5 \cdot \frac{x_1 - x_3 \div x_4}{t1} \cdot (1 \pm 10^{-7})$	0.11 s	80.2 s
$y = \sqrt{x_1 + x_2} \cdot \sin(x_2) \cdot (1 \pm 10^{-7})$	0.31 s	0.12 s
$y = \exp(\cos(x_1) + \sin(x_2)) \cdot (1 \pm 10^{-7})$	0.001 s	0.08 s
Moyenne sur 100 exigences	0.28 s	27.34 s

Tableau 6.2 – Performances de la solution en fonction de la complexité de l'exigence considérée

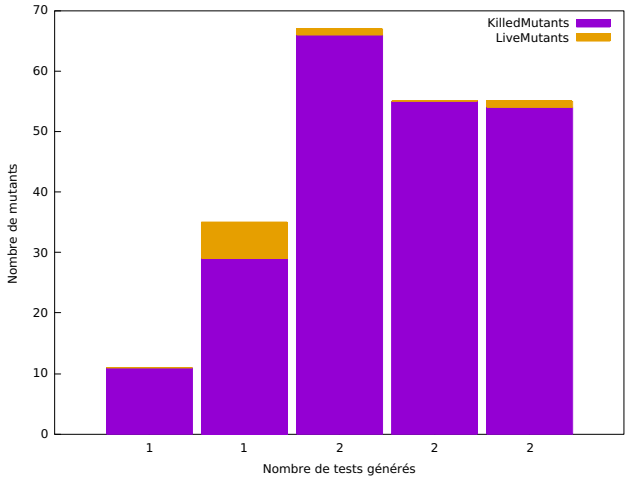


FIGURE 6.4 – Nombre de tests générés et répartition des mutants tués et équivalents en fonction de quatre exigences

d'exécution de la procédure de génération de tests bien plus élevé que les autres. Ces 80.2 secondes s'expliquent par le fait que la résolution de contrainte a été arrêtée au bout de 20 secondes pour 4 mutants. Aucun test n'a été généré pour ces mutants. Sur les 6 mutants ayant survécu à la campagne de tests générée (composée d'un test), 4 résultent de l'échec du générateur de test après ce temps maximum attribué pour la recherche. Dans le cas de la troisième exigence, un mutant n'a pas été tué, mais l'exécution du CSP a permis de détecter ce mutant équivalent de façon immédiate. Le temps d'exécution moyen de l'algorithme de génération de tests est de $\approx 30s$ pour l'ensemble d'exigences générées, et le nombre de tests moyen généré pour une exigence est de 2. Nous remarquons également un temps d'exécution anormalement élevé pour la génération des mutants de la première exigence qui est pourtant plus simple que les autres. Ce phénomène s'explique par les optimisations

effectuées par Julia en termes d'allocation de mémoire et du ramasse-miettes.

Ce qui ressort de ces tests est que cette méthode est globalement fiable pour les mutations implémentées. Elle permet de générer des tests pertinents en un temps raisonnable (moins d'une minute en moyenne), mais pourrait tout de même être optimisée selon plusieurs critères. En premier lieu, une optimisation du raffinement de la précision Δ et les bornes des variables pourrait mener à la génération de tests permettant de tuer les mutants ayant survécu à la campagne de test qui ne sont pas équivalents aux exigences. Ensuite toutes les valeurs des constantes et des bornes des variables sont traitées comme des chaînes de caractères, et les opérations arithmétiques fiables sont effectuées dans le solveur de contraintes. Cependant certaines mutations exigent de modifier ces valeurs, des conversions entre les formats décimaux et binaires (`BigFloat`) sont alors effectuées sans que les erreurs soient maîtrisées. Une optimisation permettant d'augmenter la fiabilité de cette solution serait d'intégrer la bibliothèque MPDI introduite dans le chapitre 4 afin d'effectuer tous les calculs uniquement en intervalles décimaux multi-précision.

6.5 Conclusion

Dans ce chapitre, nous avons introduit les notions de test par mutation et programmation par contraintes, afin de prototyper une solution de génération de tests à partir d'une spécification fonctionnelle. Les caractéristiques de la programmation par contraintes sur variables définies sur des domaines continus a été jugée comme une solution adéquate pour représenter les exigences numériques complexes présentes dans les spécifications de logiciels aéronautiques. Le test de mutation quant à lui est une méthode d'évaluation de la qualité d'une campagne de test versatile, pouvant être transformé en méthode de génération de tests selon des objectifs de tests définis comme des mutations. La combinaison de ces deux approches n'avait pas encore été étudiées pour la génération de tests numériques fiables. Nous proposons un algorithme optimisé pour la génération de tests numériques, pour lesquels une description de l'ensemble des solutions n'est pas envisageable. Cet algorithme tire parti des spécificités de la programmation par contraintes pour trouver un sous-ensemble de solutions fiables permettant de distinguer un mutant de l'exigence qui l'a engendré, et ainsi générer un test pertinent.

Nous proposons un prototype développé en Julia, utilisant le solveur de contraintes `RealPaver`. Après quelques modifications de ce dernier, nous pouvons tirer parti des solutions fiables qui sont alors calculées. Cette solution a été testée sur un ensemble représentatif d'exigences et pour un sous-ensemble de mutations. Certaines parties du développement ne sont pas encore optimisées, menant à des ralentissements du temps d'exécution, notamment lorsque le CSP de génération de test opère sur un espace de recherche trop grand. De plus certaines parties du logiciel effectuent des opérations arithmé-

tiques pouvant mener à des erreurs. Une perspective serait donc l'intégration des solutions d'arithmétique fiable implémentées dans les chapitres précédents de cette thèse.

CONCLUSION ET PERSPECTIVES

DANS cette thèse, nous avons traité les problématiques liées à l'automatisation de la génération des tests à partir d'un formalisme de spécification fonctionnelle, soumise aux contraintes de la certification aéronautique. La diversité des fonctionnalités décrites par les exigences ne favorisant pas l'étude d'une solution généraliste, nous nous sommes concentrés sur les exigences numériques, pour lesquelles l'effort de test est le plus important. Nous avons donc divisé nos contributions selon deux axes :

- l'étude du paysage de la génération automatique de tests et de la faisabilité d'une solution de génération automatique de tests compatibles avec les contraintes de la certification aéronautique,
- l'élaboration de bibliothèques arithmétiques proposant des opérations fiables dans les formats binaires et décimaux, afin de fournir des résultats précis pour la génération des tests numériques, qui utilise une combinaison de ces formats.

Nous allons dans un premier temps développer ces deux axes de contributions, puis présenter, dans un second temps, les perspectives résultant de ces travaux.

Contributions

Vers la génération automatique de tests numériques

Le sujet de la génération automatique de tests n'est abordé que depuis quelques années dans le monde du logiciel aéronautique. Le développement de ces produits est soumis à de fortes contraintes de certification. L'application de méthodes de génération automatique de tests numériques fiables sur un formalisme de spécification a peu été étudiée et n'a à notre connaissance jamais fait l'objet d'une solution outillée.

Dans un premier temps, nous avons donc proposé l'utilisation d'un formalisme permettant de représenter le comportement fonctionnel des exigences numériques : les

contraintes sur des domaines continus. Tout en étant une méthode de définition de l'ensemble des solutions précis, la programmation par contraintes est conforme aux critères de certification aéronautique déclarant que la spécification fonctionnelle ne doit pas décrire la solution logicielle.

À partir de cette formalisation, nous avons développé un algorithme de génération automatique de tests numériques fiables grâce à l'adaptation de la méthode de test par mutations sur des contraintes continues. Le test par mutations permet de simuler des erreurs potentielles à l'intérieur des contraintes en les modifiant selon un règle appelée mutation. Si la campagne de test, composée d'un nombre de tests positif ou nul, ne permet pas de détecter le mutant généré par cette mutation, un test est alors généré pour enrichir la campagne de tests. Dans le cas où le solveur de contraintes ne parvient pas à trouver une solution en un temps fini, l'algorithme renvoie un rapport d'erreur décrivant précisément quel mutant n'a pas été détecté, permettant au testeur d'enrichir la campagne de test manuellement. Nous avons développé un prototype de générateur automatique de tests en Julia [122] utilisant le solveur de contraintes `RealPaver` [5].

Arithmétique fiable pour la génération automatique de tests

L'arithmétique utilisée dans notre solution de génération automatique de test est un mélange d'arithmétique par intervalles à précision finie décimale (exigences et résultats du solveur de contraintes `RealPaver`) et d'arithmétique multi-précision binaire dans le générateur de tests (`BigFloat` en Julia). Nous proposons de fournir des solutions arithmétiques fiables en binaire et décimal afin d'apporter un certain niveau de confiance dans les résultats de la génération automatique de tests.

Bibliothèque MPDI. Notre première solution est de proposer une arithmétique d'intervalles décimale multi-précision avec arrondi correct : la bibliothèque MPDI. Cette bibliothèque fournit les opérations arithmétiques de base ainsi que l'exponentielle et le logarithme. Nous fournissons une preuve de l'algorithme de conversion binaire/décimal, central pour notre solution. La représentation décimale fournie est compatible avec la norme IEEE 754-2008 [38], et les opérations sur les intervalles sont conformes avec la norme IEEE 1788-2015 [42]. La solution implémentée repose sur les bibliothèques GMP [49] et MPFR [11], et a été minutieusement testée.

Arithmétique en bases mixtes. Dans notre seconde solution, nous nous intéressons aux formats IEEE 754-2008 standards, binaires et décimaux. Nous proposons les opérations arithmétique de base, prenant en entrée une combinaison des formats binaires et décimaux IEEE 754-2008, et renvoyant une sortie dans l'un de ces formats correctement arrondie. Cette arithmétique repose sur l'opération Fused-Multiply-and-Add en bases mixtes, dont

nous proposons une analyse détaillées. Notre implémentation a été testée et comparée à deux autres implémentations de référence par rapport auxquelles elle a été estimée valide et compétitive quant à ses performances temporelles.

Perspectives

Nous présentons succinctement ici les grands axes de nos travaux futurs.

À court terme

Formaliser les paramètres de précision de l'algorithme de génération de tests. Actuellement, la terminaison de l'algorithme de génération de tests repose sur des bornes sur le temps d'exécution maximum et sur la détermination d'un nombre fini de boîtes contenant les solutions, borné par la précision du solveur de contraintes Δ, déterminée à l'aide d'une estimation. Nous souhaitons fournir une description formelle de cette précision Δ, afin de cibler la précision du solveur la plus adéquate à partir de laquelle il existe des résultats sous forme de boîtes contenant les solutions recherchées.

Améliorer la fiabilité arithmétique pour la génération automatique de tests. À l'heure actuelle, les solutions d'arithmétique fiable développées dans cette thèse n'ont pas encore été intégrées au générateur de tests. Une première perspective immédiate de ces travaux est l'intégration de la bibliothèque MPDI dans le générateur de tests. Ceci revient à créer une interface entre MPDI et Julia, nous pourrons ainsi proposer un générateur de tests utilisant uniquement l'arithmétique décimale. Une fois l'arithmétique décimale fiable intégrée à la solution, nous pourrons proposer une analyse formelle de la propagation des erreurs dans le générateur de tests.

À moyen terme

Enrichir les fonctions de la bibliothèque MPDI. Une conséquence de l'intégration de la bibliothèque MPDI est la potentielle utilisation de mutations de haut niveau [115], c'est-à-dire une combinaison de plusieurs mutations. Afin d'y parvenir, il faudrait qu'à terme la bibliothèque MPDI propose les mêmes fonctions que le solveur de contraintes utilisés. Nous estimons pour l'instant que la démonstration et l'implémentation des fonctions *sinus* et *cosinus* est un objectif relativement facile à atteindre. Il suffit de connaître la valeur $r = 10^F \cdot n - \left\lfloor \frac{2 \cdot 10^F \cdot n}{\pi} \right\rfloor \cdot \frac{\pi}{2}$ et calculer en binaire $\sin(r)$ ou $\cos(r)$.

Améliorer la détermination du verdict. Actuellement, la détermination du verdict des tests s'effectue par la comparaison du résultat attendu calculé par le testeur et du résultat obtenu par le logiciel. L'intervalle représentant le comportement attendu est calculé en

milti-précision binaire à partir d'une chaîne de caractères représentant un nombre décimal, pour renvoyer un résultat à virgule flottante binaire comparable avec le résultat obtenu du logiciel, lui aussi binaire. Cette situation pourrait être simplifiée par l'utilisation de l'arithmétique en bases mixtes. Le résultat attendu serait alors représenté par un intervalle de nombres à virgule flottante décimaux, qui pourraient être comparés avec le résultat obtenu par le logiciel toujours représenté dans un format binaire. On pourrait également imaginer enrichir les fonctionnalités de l'outil d'exécution et de dépouillement des tests à l'aide d'opérations arithmétiques fiables en bases mixtes.

À long terme

Étude de la robustesse de la solution pour d'autres types d'exigences. Nous avons concentré notre étude sur un sous-ensemble d'exigences pour lesquelles l'effort de test était le plus important : les exigences numériques. Les stimulations et comportements attendus produits par ces exigences sont cependant rarement directement accessibles. Elles se trouvent généralement intégrées dans des systèmes d'exigences plus complexes, décrivant des comportements séquentiels ainsi que des expressions conditionnelles (si, alors, lorsque, etc.). Les perspectives à long terme sont de pouvoir intégrer nos travaux sur la génération automatique de tests numériques fiables dans un système d'exigences plus diversifiées. Il s'agirait alors d'étudier la robustesse de notre solution dans le cas d'exigences sujettes à la propagations d'erreurs numériques. Par exemple, lorsque les résultats sont réutilisés, dans le cas d'une boucle sur des valeurs numériques.

Étude de la solution avec d'autres solveurs de contraintes. La solution actuelle de génération automatique de tests repose sur le solveur de contraintes `RealPaver`. C'est un solveur simple et fournissant des résultats fiables qui suffisait pour notre prototype de générateur de tests. Dans le cas d'un système d'exigences plus complexes, nous serons menés à utiliser des fonctionnalités plus avancées. Il s'agirait alors d'étudier les limites de la solution actuelle, notamment l'efficacité de `RealPaver` dans cette situation par rapport à d'autres solveurs de contraintes, comme Ibex [124] par exemple. De plus, la divergence de résultat par l'utilisation de différents solveurs de contraintes dans notre générateur de tests pourrait fournir des indications intéressantes quant aux exigences numériquement instables.

BIBLIOGRAPHIE

[1] RTCA Special Committee 205 (SC-205) and EUROCAE Working Group 71 (WG-71), "DO–178C, ED–12C, Software Considerations in Airborne Systems and Equipment Certification," tech. rep., RTCA, Inc, 2011.

[2] RTCA Special Committee 205 (SC-205) and EUROCAE Working Group 71 (WG-71), "DO–333, Formal Methods Supplement to DO-178C and DO-278A," tech. rep., RTCA, Inc, 2011.

[3] Y. Jia and M. Harman, "An analysis and survey of the development of mutation testing," *IEEE Transactions on Software Engineering*, vol. 37, pp. 649–678, Sep. 2011.

[4] Y. Michot, "Rapport sur l'industrie aéronautique et spatiale française." La documentations Française, Premier ministre, 2004.

[5] L. Granvilliers, *RealPaver User's Manual Solving Nonlinear Constraints by Interval Computations*, 01 2004.

[6] O. Ponsini, C. Michel, and M. Rueher, "Refining abstract interpretation based value analysis with constraint programming techniques," in *Principles and Practice of Constraint Programming - 18th International Conference, CP 2012, Québec City, QC, Canada, October 8-12, 2012. Proceedings*, pp. 593–607, 2012.

[7] R. E. Moore, R. B. Kearfott, and M. J. Cloud, *Introduction to Interval Analysis*. Philadelphia, PA, USA : Society for Industrial and Applied Mathematics, 2009.

[8] P. Reales Mateo, M. Polo, J. Fernández-Alemán, A. Toval, and M. Piattini, "Mutation testing," *Software, IEEE*, vol. 31, pp. 30–35, 05 2014.

[9] R. A. DeMilli and A. J. Offutt, "Constraint-based automatic test data generation," *IEEE Transactions on Software Engineering*, vol. 17, pp. 900–910, Sep. 1991.

[10] M. Ceberio, A. F. G. Contreras, C. Jeangoudoux, and F. larribe, "Constraints over Intervals for Specification Based Automatic Software Test Generation," in *18th In-*

ternational Symposium on Scientific Computing, Computer Arithmetics and Verified Numerics, (Tokyo, Japan), Sept. 2018.

[11] L. Fousse, G. Hanrot, V. Lefèvre, P. Pélissier, and P. Zimmermann, "MPFR : A multiple-precision binary floating-point library with correct rounding," *ACM Trans. Math. Softw.*, vol. 33, June 2007.

[12] S. Graillat, C. Jeangoudoux, and C. Lauter, "MPDI : A Decimal Multiple-Precision Interval Arithmetic Library," *Reliable Computing Journal*, 2017.

[13] S. Graillat, C. Jeangoudoux, and C. Lauter, "A Decimal Multiple-Precision Interval Arithmetic Library," in *17th International Symposium on Scientific Computing, Computer Arithmetics and Verified Numerics*, (Uppsala, Sweden), Sept. 2016.

[14] N. Brisebarre, C. Lauter, M. Mezzarobba, and J.-M. Muller, "Comparison between binary and decimal floating-point numbers," *IEEE Transactions on Computers*, vol. 65, no. 7, pp. 2032–2044, 2016.

[15] C. Jeangoudoux and C. Lauter, "A Correctly Rounded Mixed-Radix Fused-Multiply-Add," in *2018 IEEE 25th Symposium on Computer Arithmetic (ARITH)*, (Amherst, MA, United States), IEEE, June 2018.

[16] R. L. Alonso and A. L. Hopkins, *The Apollo guidance computer*. M.I.T. Instrumentation Laboratory, 1963.

[17] EASA, "Aircraft certification." `https://www.easa.europa.eu/easa-and-you/aircraft-products/aircraft-certification`.

[18] Federal Aviation Administration, "Order 8110.49a : Software approval guidelines." U.S. Department of Transportation, 2018.

[19] M. Bartnes, "Safety vs. security ?," 05 2006.

[20] RTCA Special Committee 205 (SC-205) and EUROCAE Working Group 71 (WG-71), "DO–331, Model-Based Developement and Verification Supplement to DO–178C and DO–278A," tech. rep., RTCA, Inc, 2011.

[21] RTCA Special Committee 205 (SC-205) and EUROCAE Working Group 71 (WG-71), "DO–330, Software Tool Qualification Considerations," tech. rep., RTCA, Inc, 2011.

[22] G. Nicolescu and P. J. Mosterman, *Model-Based Design for Embedded Systems*. Boca Raton, FL, USA : CRC Press, Inc., 1st ed., 2009.

[23] J. H. Gallier, *Logic for Computer Science : Foundations of Automatic Theorem Proving*. Wiley, 1987.

[24] D. A. Peled, "Model checking basics," in *Engineering Dependable Software Systems*, pp. 335–362, IOS Press, 2013.

[25] P. Cousot and R. Cousot, "Abstract interpretation : Past, present and future," in *Proceedings of the Joint Meeting of the Twenty-Third EACSL Annual Conference on Computer Science Logic (CSL) and the Twenty-Ninth Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*, CSL-LICS '14, (New York, NY, USA), pp. 2 :1–2 :10, ACM, 2014.

[26] J. Cleland-Huang, O. C. Z. Gotel, J. Huffman Hayes, P. Mäder, and A. Zisman, "Software traceability : Trends and future directions," *Future of Software Engineering, FOSE 2014 - Proceedings*, pp. 55–69, 05 2014.

[27] G. J. Myers and C. Sandler, *The Art of Software Testing*. USA : John Wiley &#38 ; Sons, Inc., 2004.

[28] G. Berry, "The foundations of Esterel," in *Proof, Language, and Interaction : Essays in Honour of Robin Milner* (G. Plotkin, C. Stirling, and M. Tofte, eds.), (Cambridge, MA, USA), MIT Press, 2000.

[29] J.-L. Boulanger, F.-X. Fornari, J.-L. Camus, and B. Dion, *SCADE : Language and Applications*. Wiley-IEEE Press, 1st ed., 2015.

[30] T. Le Sergent, F.-X. Dormoy, and A. Le Guennec, "Benefits of Model Based System Engineering for Avionics Systems," in *8th European Congress on Embedded Real Time Software and Systems (ERTS 2016)*, (TOULOUSE, France), Jan. 2016.

[31] D. Sadoun, *Des spécifications en langage naturel aux spécifications formelles via une ontologie comme modèle pivot*. Theses, Université Paris Sud - Paris XI, June 2014.

[32] European Aviation Safety Agency, "Somca - safety implications in performing software model coverage analysis," 2011. `https://www.easa.europa.eu/document-library/research-projects/easa20103#group-easa-downloads`.

[33] J. J. Chilenski, "An investigation of three forms of the modified condition decision coverage (mcdc) criterion," tech. rep., Office of Aviation Research, 2001.

[34] R. Brent and P. Zimmermann, *Modern Computer Arithmetic*. New York, NY, USA : Cambridge University Press, 2010.

[35] D. Goldberg, "What every computer scientist should know about floating-point arithmetic," *ACM Comput. Surv.*, vol. 23, pp. 5–48, Mar. 1991.

[36] J.-M. Muller, N. Brunie, F. de Dinechin, C.-P. Jeannerod, *et al.*, *Handbook of Floating-Point Arithmetic, 2nd edition*. Birkhäuser Boston, 2018. ACM G.1.0 ; G.1.2 ; G.4 ; B.2.0 ; B.2.4 ; F.2.1., ISBN 978-3-319-76525-9.

[37] C. Severance, "An interview with the old man of floating-point. Reminiscences elicited from William Kahan." World-Wide Web document., Feb. 1998.

[38] *IEEE Standard for Floating-Point Arithmetic*, Aug 2008.

[39] J.-M. Muller, "On the definition of ulp(x)," Tech. Rep. RR-5504, INRIA, Feb. 2005.

[40] A. Neumaier, *Interval methods for systems of equations*. Cambridge University Press, Cambridge, UK, 1990.

[41] H. Dawood, *Theories of Interval Arithmetic : Mathematical Foundations and Applications*. LAP Lambert Academic Publishing, 2011.

[42] *IEEE Standard for Interval Arithmetic*, June 2015.

[43] G. Melquiond, *From interval arithmetic to program verification*. Theses, École Normale Supérieure de Lyon, Nov. 2006.

[44] V. Ménissier, *Arithmétique exacte : conception, algorithmique et performances d'une implémentation informatique en précision arbitraire*. PhD thesis, Université Paris 7, 1994.

[45] U. Kulisch and V. Snyder, "The Exact Dot Product As Basic Tool for Long Interval Arithmetic," *Computing*, vol. 91, pp. 307–313, Mar 2011.

[46] M. Joldes, O. Marty, J. M. Muller, and V. Popescu, "Arithmetic algorithms for extended precision using floating-point expansions," *IEEE Transactions on Computers*, vol. 65, pp. 1197–1210, April 2016.

[47] D. A. Pope and M. L. Stein, "Multiple precision arithmetic," *Commun. ACM*, vol. 3, pp. 652–654, Dec. 1960.

[48] B. P. Serpette, J. Vuillemin, and H. Jean-Claude, *BigNum : a portable and efficient package for arbitrary-precision arithmetic*. Digital, Paris Research Laboratory, 1989.

[49] T. Granlund and the GMP development team, *GNU MP : The GNU Multiple Precision Arithmetic Library*, 6.1.1 ed., 2016. `http://gmplib.org/`.

[50] N. Revol and F. Rouillier, "Motivations for an Arbitrary Precision Interval Arithmetic and the MPFI Library.," *Reliable Computing*, vol. 11, no. 4, pp. 275–290, 2005.

[51] S. M. Rump, "Intlab — interval laboratory," in *Developments in Reliable Computing* (T. Csendes, ed.), pp. 77–104, Dordrecht : Springer Netherlands, 1999.

[52] D. Y. Nadezhin and S. I. Zhilin, "Jinterval library : Principles, development, and perspectives," *Reliable Computing*, vol. 19, no. 3, pp. 229–247, 2014.

[53] A. Goldsztejn, "Modal intervals revisited, part 1 : A generalized interval natural extension," *Reliable Computing*, vol. 16, pp. 130–183, 2012.

[54] A. Goldsztejn, "Modal intervals revisited, part 2 : A generalized interval mean value extension," *Reliable Computing*, vol. 16, pp. 184–209, 2012.

[55] B. Josh, C. Mike, and D. D. Joseph, "Java™ Platform, Standard Edition 7 API Specification, Class BigDecimal." `http://docs.oracle.com/javase/7/docs/api/java/math/BigDecimal.html`.

[56] K. Stefan, "mpdecimal, a Package for Correctly-Rounded Arbitrary Precision Decimal Floating Point Arithmetic." `http://www.bytereef.org/mpdecimal/index.html`.

[57] C. Daramy, D. Defour, F. Dinechin, and J.-M. Muller, "Cr-libm : A correctly rounded elementary function library," *Proceedings of SPIE - The International Society for Optical Engineering*, vol. 5205, 12 2003.

[58] S. Chevillard, M. Joldeş, and C. Lauter, "Sollya : An Environment for the Development of Numerical Codes," in *Mathematical Software - ICMS 2010*, vol. 6327 of *LNCS*, pp. 28–31, Sept 2010.

[59] S. Chevillard, C. Lauter, and M. Joldes, *Users' manual for the Sollya tool, Release 7.0*. LIP, LIP6, LORIA, CNRS, APICS, INRIA.

[60] F. Jézéquel and J.-M. Chesneaux, "CADNA : a library for estimating round-off error propagation," *Computer Physics Communications*, vol. 178, no. 12, pp. 933–955, 2008.

[61] Federal Aviation Administration, William J. Hughes Technical Center, Aviation Research Division, "Advanced Verification Methods for Safety-Critical Airborne Electronic Hardware," tech. rep., U.S. Department of Transportation, Federal Aviation Administration, 2015.

[62] D. G. Luenberger and Y. Ye, *Linear and Nonlinear Programming*. Springer Publishing Company, Incorporated, 2015.

[63] T. Niermann and J. H. Patel, "Hitec : A test generation package for sequential circuits," in *Proceedings of the Conference on European Design Automation*, EURO-DAC '91, (Los Alamitos, CA, USA), pp. 214–218, IEEE Computer Society Press, 1991.

[64] R. M. Hierons, K. Bogdanov, J. P. Bowen, R. Cleaveland, *et al.*, "Using formal specifications to support testing," *ACM Comput. Surv.*, vol. 41, pp. 9 :1–9 :76, Feb. 2009.

[65] F. Wagner, R. Schmuki, T. Wagner, and P. Wolstenholme, *Modeling software with finite state machines : a practical approach*. Auerbach Publications, 2006.

[66] D. Harel, “Statecharts : A visual formalism for complex systems,” *Sci. Comput. Program.*, vol. 8, pp. 231–274, June 1987.

[67] J. M. Spivey, *The Z Notation : A Reference Manual*. Upper Saddle River, NJ, USA : Prentice-Hall, Inc., 1989.

[68] J.-R. Abrial, *The B-book : Assigning Programs to Meanings*. New York, NY, USA : Cambridge University Press, 1996.

[69] H. Ehrig, B. Mahr, I. Claßen, and F. Orejas, “Introduction to algebraic specification. part 1 : Formal methods for software development,” *Comput. J.*, vol. 35, pp. 460–467, 10 1992.

[70] E. Astesiano, M. Bidoit, H. Kirchner, B. Krieg-Brückner, P. D. Mosses, D. Sannella, and A. Tarlecki, “CASL : the Common Algebraic Specification Language,” *Theoretical Computer Science*, vol. 286, no. 2, pp. 153 – 196, 2002. Current trends in Algebraic Development Techniques.

[71] A. M. Moreira, C. Ringeissen, D. Déharbe, and G. Lima, “Manipulating algebraic specifications with term-based and graph-based representations,” *The Journal of Logic and Algebraic Programming*, vol. 59, no. 1, pp. 63 – 87, 2004. Annotated Terms (ATerms).

[72] J. W. Lloyd, “Practical Advantages of Declarative Programming,” in *Joint Conference on Declarative Programming*, 1994.

[73] M. Bramer, *Logic Programming with Prolog*. Springer Publishing Company, Incorporated, 2nd ed., 2014.

[74] S. Thompson, *The Haskell : The Craft of Functional Programming*. Boston, MA, USA : Addison-Wesley Longman Publishing Co., Inc., 2nd ed., 1999.

[75] Wolfram Research, Inc., “Wolfram programming lab, Version 11.3.” Champaign, IL, 2018.

[76] K. Apt, *Principles of Constraint Programming*. New York, NY, USA : Cambridge University Press, 2003.

[77] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud, “The synchronous dataflow programming language lustre,” *Proceedings of the IEEE*, vol. 79, pp. 1305–1320, sep 1991.

[78] J. C. H. Lee and J. H.-M. Lee, “Towards practical infinite stream constraint programming : Applications and implementation,” in *CP*, 2014.

[79] P. Mauro and M. Young, *Software testing and analysis : process, principles, and techniques*. Wiley, 2008.

[80] S. Anand, E. K. Burke, T. Y. Chen, J. Clark, *et al.*, "An orchestrated survey of methodologies for automated software test case generation," *J. Syst. Softw.*, vol. 86, pp. 1978–2001, Aug. 2013.

[81] R. Bagnara, M. Carlier, R. Gori, and A. Gotlieb, "Symbolic path-oriented test data generation for floating-point programs," in *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation*, pp. 1–10, March 2013.

[82] F. Hutter, H. H. Hoos, and K. Leyton-Brown, "Sequential model-based optimization for general algorithm configuration," in *Proceedings of the 5th International Conference on Learning and Intelligent Optimization*, LION'05, (Berlin, Heidelberg), pp. 507–523, Springer-Verlag, 2011.

[83] T. A. Budd and D. Angluin, "Two notions of correctness and their relation to testing," *Acta Inf.*, vol. 18, pp. 31–45, Mar. 1982.

[84] E. J. Weyuker, "The evaluation of program-based software test data adequacy criteria," *Commun. ACM*, vol. 31, pp. 668–675, June 1988.

[85] F. Rossi, P. van Beek, and T. Walsh, eds., *Handbook of Constraint Programming*, vol. 2 of *Foundations of Artificial Intelligence*. Elsevier, 2006.

[86] B. Marre and A. Arnould, "Test sequences generation from lustre descriptions : Gatel," in *Proceedings ASE 2000. Fifteenth IEEE International Conference on Automated Software Engineering*, pp. 229–237, Sep. 2000.

[87] H. Collavizza, C. Michel, O. Ponsini, and M. Rueher, "Generating test cases inside suspicious intervals for floating-point number programs," in *Proceedings of the 6th International Workshop on Constraints in Software Testing, Verification, and Analysis*, CSTVA 2014, (New York, NY, USA), pp. 7–11, ACM, 2014.

[88] A. Biere, A. Biere, M. Heule, H. van Maaren, and T. Walsh, *Handbook of Satisfiability : Volume 185 Frontiers in Artificial Intelligence and Applications*. Amsterdam, The Netherlands, The Netherlands : IOS Press, 2009.

[89] P. Chassaing, S. Conchon, V. Delecroix, B.), L. Paulevé, N. Ray, A. Richard, L.), D. Sokolov, E. Jeandel, *et al.*, *Informatique mathématique : une photographie en 2018*. CNRS éditions [alpha], CNRS éditions, 2018.

[90] B. Marre, F. Bobot, and Z. Chihani, "Real Behavior of Floating Point Numbers," in *The SMT Workshop*, (Heidelberg, Germany), SMT 2017, 15th International Workshop on Satisfiability Modulo Theories, July 2017.

[91] E. Jeandel and V. Laurent, *Informatique Mathématique, une photographie en 2018*. CNRS Editions, 2018.

[92] R. Hamlet, "Random testing," in *Encyclopedia of Software Engineering*, pp. 970–978, Wiley, 1994.

[93] P. Cousot and R. Cousot, "Abstract interpretation : Past, present and future," in *Proceedings of the Joint Meeting of the Twenty-Third EACSL Annual Conference on Computer Science Logic (CSL) and the Twenty-Ninth Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*, CSL-LICS '14, (New York, NY, USA), pp. 2 :1–2 :10, ACM, 2014.

[94] O. Ponsini, C. Michel, and M. Rueher, "Verifying floating-point programs with constraint programming and abstract interpretation techniques," *Automated Software Engg.*, vol. 23, pp. 191–217, June 2016.

[95] L. Chen, A. Miné, J. Wang, and P. Cousot, "An abstract domain to discover interval linear equalities," *Proc. of the 11th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI'10)*, 01 2010.

[96] P. Cousot and R. Cousot, "Modular static program analysis," in *Proceedings of the 11th International Conference on Compiler Construction*, CC '02, (London, UK, UK), pp. 159–178, Springer-Verlag, 2002.

[97] D. Delmas, E. Goubault, S. Putot, J. Souyris, *et al.*, "Towards an industrial use of fluctuat on safety-critical avionics software," in *Proceedings of the 14th International Workshop on Formal Methods for Industrial Critical Systems*, FMICS '09, (Berlin, Heidelberg), pp. 53–69, Springer-Verlag, 2009.

[98] P. Cuoq, F. Kirchner, N. Kosmatov, V. Prevosto, *et al.*, "Frama-c : A software analysis perspective," in *Proceedings of the 10th International Conference on Software Engineering and Formal Methods*, SEFM'12, (Berlin, Heidelberg), pp. 233–247, Springer-Verlag, 2012.

[99] C. Michel, M. Rueher, and Y. Lebbah, "Solving constraints over floating-point numbers," in *Proceedings of the 7th International Conference on Principles and Practice of Constraint Programming*, CP '01, (London, UK, UK), pp. 524–538, Springer-Verlag, 2001.

[100] P. McMinn, "Search-based software testing : Past, present and future," in *2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops*, pp. 153–163, March 2011.

[101] I. W. Soares and S. R. Vergilio, "Mutation analysis and constraint-based criteria : Results from an empirical evaluation in the context of software testing," *Journal of Electronic Testing : Theory and Applications*, vol. 20, pp. 439–445, August 2004.

[102] E. P. Enoiu, A. Čaušević, T. J. Ostrand, E. J. Weyuker, *et al.*, "Automated test generation using model checking : An industrial evaluation," *Int. J. Softw. Tools Technol. Transf.*, vol. 18, pp. 335–353, June 2016.

[103] J. Harrison, "Decimal Transcendentals via Binary," in *2009 19th IEEE Symposium on Computer Arithmetic*, pp. 187–194, June 2009.

[104] A. Ziv, "Fast evaluation of elementary mathematical functions with correctly rounded last bit," *ACM Trans. Math. Softw.*, vol. 17, pp. 410–423, Sept. 1991.

[105] P. H. Sterbenz, *Floating-Point Computation*. Prentice-Hall series in automatic computation, Prentice-Hall, 1974.

[106] M. Cornea *et al.*, "A Software Implementation of the IEEE 754R Decimal Floating-Point Arithmetic Using the Binary Encoding Format," in *18th IEEE Symposium on Computer Arithmetic*, pp. 29–37, June 2007.

[107] O. Kupriianova, *Towards a Modern Floating-Point Environment*. Theses, Université Pierre et Marie Curie - Paris VI, Dec. 2015.

[108] F. De Dinechin *et al.*, "On Ziv's Rounding Test," *ACM Transactions on Mathematical Software*, vol. 39, no. 4, p. 26, 2013.

[109] R. Amadini, *Portfolio Approaches in Constraint Programming*. PhD thesis, University of Bologna, Italy, 2015.

[110] R. Barták, "Constraint programming : In pursuit of the holy grail," in *In Proceedings of the Week of Doctoral Students (WDS99 -invited lecture*, pp. 555–564, 1999.

[111] L. Granvilliers and F. Benhamou, "Algorithm 852 : RealPaver : an interval solver using constraint satisfaction techniques," *ACM Transactions on Mathematical Software*, vol. 32, no. 1, pp. 138–156, 2006.

[112] G. Chabert and L. Jaulin, "Contractor programming," *Artificial Intelligence*, vol. 173, no. 11, pp. 1079 – 1100, 2009.

[113] J. Andrews, L. Briand, and Y. Labiche, "Is mutation an appropriate tool for testing experiments ?," Dec. 2005.

[114] S. Nica and F. Wotawa, "Using constraints for equivalent mutant detection," in *WS-FMDS*, 2012.

[115] Y. Jia and M. Harman, "Higher order mutation testing," *Inf. Softw. Technol.*, vol. 51, pp. 1379–1393, Oct. 2009.

[116] S. Gupta, A. Saxena, A. Mahajan, and S. Bansal, "Effective use of SMT solvers for program equivalence checking through invariant-sketching and query-decomposition," in *Theory and Applications of Satisfiability Testing - SAT 2018 - 21st International Conference, SAT 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 9-12, 2018, Proceedings*, pp. 365–382, 2018.

[117] K. Adamopoulos, M. Harman, and R. Hierons, "How to overcome the equivalent mutant problem and achieve tailored selective mutation using co-evolution," *Genetic and evolutionary computation*, vol. 3103, pp. 1338–1349, 06 2004.

[118] B. J. M. Grün, D. Schuler, and A. Zeller, "The impact of equivalent mutants," in *2009 International Conference on Software Testing, Verification, and Validation Workshops*, pp. 192–199, April 2009.

[119] E. S. Mresa and L. Bottaci, "Efficiency of mutation operators and selective mutation strategies : An empirical study.," *Softw. Test., Verif. Reliab.*, vol. 9, pp. 205–232, 12 1999.

[120] C. Byoungju and A. P. Mathur, "High-performance mutation testing," *Journal of Systems and Software*, vol. 20, no. 2, pp. 135 – 152, 1993.

[121] P. Reales Mateo and M. Polo, "Parallel mutation testing," *Software Testing, Verification and Reliability*, vol. 23, 06 2013.

[122] J. Bezanzon, S. Karpinski, V. Shah, and A. Edelman, "Julia : A fast dynamic language for technical computing," in *Lang.NEXT*, Apr. 2012.

[123] J. Bezanson, A. Edelman, S. Karpinski, and V. B. Shah, "Julia : A fresh approach to numerical computing," *SIAM Review*, vol. 59, pp. 65–98, 2017.

[124] J. Ninin, "Global Optimization based on Contractor Programming : an Overview of the IBEX library," in *MACIS*, (Berlin, Germany), Nov. 2015.