Chapitre 2

L'analyseur statique ASTRÉE

2.1. Introduction

Dans le domaine des logiciels critiques embarqués, une erreur logicielle peut avoir des conséquences désastreuses, non seulement en termes économiques, mais également au niveau humain. Par exemple, l'échec du vol inaugural de la fusée Ariane 5 a entraîné une perte financière de l'ordre de 500 M$. Pourtant, cet accident était dû à une simple erreur logicielle : un dépassement de capacité lors d'une conversion de nombres flottants 64 bits vers des entiers 16 bits a provoqué une exception matérielle, qui n'était pas rattrapée car ce cas de figure avait été considéré impossible à la conception [LIO 96], cette erreur logicielle entraînant la perte des calculateurs de vols et donc du lanceur. Cet accident ne constitue pas un cas isolé : citons également le cas de l'échec du missile Patriot à Dharan en 1992 [SKE 92], le crash du prototype 39-1 du Saab Gripen JAS 39A le 2 février 1989 et du premier avion de production le 8 août 1993, en raison d'instabilités numériques dans les commandes de vol [NEU 89], etc.

Du point de vue du concepteur de tels systèmes, il importe donc de s'assurer de la sûreté des logiciels avant des les exécuter dans le système final. Les méthodes classiques, qui consistent en général à effectuer de longues campagnes de tests (tests unitaires au niveau de fragment de programmes, puis tests d'intégration au niveau du système complet), sont utiles mais pas suffisantes. En effet, la complexité de ces logiciels critiques empêche de tester toutes les exécutions, et le test est donc nécessairement *incorrect* lorsqu'il s'agit de prouver de manière sûre et certaine que le logiciel

Chapitre rédigé par Julien Bertrane, Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, Xavier Rival .

considéré satisfait bien des propriétés de sûreté importantes, telle que l'absence d'erreurs à l'exécution. Même dans le cas de tests couvrant totalement l'ensemble des chemins d'exécutions, il est possible qu'une donnée qui n'a pas été considérée lors des tests provoque une erreur sur un chemin pourtant exploré (couverture incomplète des données). Par ailleurs, une campagne de tests requiert des ressources de calcul importantes, ce qui rend la maintenance du logiciel extrêmement coûteuse, puisque toute modification nécessiterait de rejouer les tests afin de vérifier l'absence de régressions vis-à-vis du jeu de test utilisé lors du développement initial.

Les techniques de vérification ou de preuve assistées qui permettent de définir une représentation formelle d'un programme, de sa spécification et de ses invariants, puis de vérifier la correction en supposant qu'un utilisateur fournit les étapes critiques de la preuve de correction (sous la forme d'invariants intermédiaires ou de tactiques de preuve à appliquer) ne sont pas non plus idéales. En effet, dans le cadre d'un développement industriel, les ressources humaines sont également souvent limitées, alors que ces techniques exigent généralement la réalisation d'un travail fastidieux pour guider la preuve, à répéter en cas d'évolution du logiciel.

À l'inverse, l'analyse statique par interprétation abstraite permet de construire des outils produisant des preuves de correction de manière totalement automatique, et exigeant peu de travail de la part de l'utilisateur. Le principe de l'analyse statique par interprétation abstraite est de calculer une *sur-approximation* des comportements du programme considéré afin de tenter de prouver sa correction. Ainsi, si les calculs montrent que les valeurs d'une variable `x` sont toutes comprises dans l'intervalle $[10, 20]$, alors nous savons que l'opération qui consiste à diviser par `x` est sûre (i.e., il ne peut y avoir de division par zéro). Cela signifie qu'une telle analyse est *correcte*, c'est-à-dire qu'elle garantit effectivement que le programme vérifie une propriété de correction. Dans l'autre sens, si le calcul de l'analyse conclue que les valeurs de `x` sont comprises dans l'intervalle $[-5, 8]$, nous ne pouvons pas en déduire pour autant qu'une division par `x` entraînera une erreur pour au moins une exécution du programme, car l'analyse ne garantit pas que la valeur 0 est effectivement atteinte. L'analyse est donc *incomplète*, ce qui signifie qu'elle peut émettre des *fausses alarmes* ; autrement dit, elle peut échouer à établir la correction d'un programme qui est pourtant sûr.

Dans ce chapitre, nous présentons l'analyseur statique ASTRÉE [BLA 02, BLA 03, COU 05], qui vise plus particulièrement les applications embarquées de type synchrone développées en C. Il est possible de considérer dans un analyseur statique des familles plus générales de programmes C. Néanmoins, la conception d'un analyseur statique rapide et efficace (i.e., capable d'analyser des programmes de taille industrielle) mais également précis (i.e., émettant peu de fausses alarmes) est particulièrement complexe. Pourtant, rapidité et précision sont indispensables dans le cadre d'une application industrielle, où l'analyse statique doit s'intégrer de manière naturelle au processus de développement, sans ralentir les programmeurs, ni du fait du temps de l'analyse, ni par nécessité de vérifier manuellement de nombreuses alarmes.

Pour cette raison, ASTRÉE a été conçu comme un analyseur *spécifique* à certaines familles de programmes, pour lesquelles une preuve de sûreté par analyse statique est particulièrement importante, à savoir des applications critiques embarquées. Ainsi, la conception de l'analyseur exploite les caractéristiques de ces logiciels, ce qui a rendu possibles des analyses efficaces et précises, débouchant en particulier sur la preuve complètement automatique d'absence d'erreurs à l'exécution dans des applications aéronautiques telles que des commandes de vol électriques, dont la taille est de l'ordre de quelques centaines de milliers de lignes, en un temps de calcul de l'ordre de quelques dizaines d'heures sur des ordinateurs standards. Par la suite, il a été possible d'étendre le domaine d'application de l'analyseur ([BOU 09, BER 10]), tout en préservant ses performances. À la date d'écriture de ce chapitre, ASTRÉE est industrialisé par l'entreprise AbsInt [Abs 10].

Dans ce chapitre, nous suivons le plan suivant : tout d'abord, nous présentons la problématique de la preuve automatique d'absence d'erreurs à l'exécution dans le cas des logiciels critiques embarqués (section 2.2), puis nous décrivons les principes de l'analyse statique par interprétation abstraite (section 2.3) sur lesquels se fonde la conception d'ASTRÉE ; ensuite, nous abordons plus en détails la structure de l'analyseur ASTRÉE (section 2.4), puis nous décrivons ses principaux domaines d'application (section 2.5). Ce chapitre est en partie fondé sur l'article [BER 10] des même auteurs. Une bibliographie complète sur ASTRÉE comprendrait de plus les articles théoriques [BLA 02, BLA 03, FER 04, MIN 04a, FER 05b, FER 05a, COU 05, MAU 05, MON 05, MIN 06b, COU 06, MIN 06a, COU 07a, RIV 07], ainsi que [DEL 07, BOU 09, KÄS 10] pour ses applications. La théorie de l'interprétation abstraction est développée, entre autres, dans [COU 77b, COU 78a, COU 79, COU 81, COU 92].

2.2. Preuve d'absence d'erreurs à l'exécution par analyse statique

Dans cette section, nous définissons plus précisément le domaine d'application d'ASTRÉE et nous décrivons les principaux problèmes résolus par cet outil.

2.2.1. *Programmes analysés*

ASTRÉE a été initialement conçu pour l'analyse de programmes présentant des caractéristiques spécifiques. En particulier, les programmes visés sont typiquement des programmes embarqués, de type synchrone, écrits en C, et qui présentent les traits suivants :

– ces programmes n'allouent pas de mémoire dynamiquement (`malloc`), ce qui implique que la structure de la mémoire du programme est fixée statiquement ;

– ces programmes ne contiennent pas de fonctions récursives ni de sauts non locaux (tels que `longjmp`) ;

– ces applications effectuent généralement des calculs numériques (sur des nombres entiers et des nombres à virgule flottante) et booléens complexes, et incluent souvent des opérations telles que des fonctions d'interpolations, des filtres linéaires, des tests booléens avec branchements différés, etc. ;

– ces programmes sont de taille industrielle ; il peuvent comprendre un million de lignes de code et des dizaines de milliers de variables globales.

Comme nous le verrons dans les sections suivantes, certaines de ces caractéristiques simplifient la conception de l'analyse (comme l'absence de récursion et d'allocation dynamique), tandis que d'autres exigent un traitement particulier, comme la présence de calculs numériques et booléens complexes.

2.2.2. *But de l'analyse : vers une preuve de sûreté*

L'analyse statique de tels programmes par ASTRÉE a pour but de prouver leur sûreté, c'est-à-dire qu'ils ne causeront pas d'erreur à l'exécution ni de comportement indéfini par la sémantique du langage et qu'ils ne commettront pas d'action erronée du point de vue de la sémantique du langage C [ISO 07] ou de l'utilisateur. Plus précisément, ASTRÉE vise à établir les propriétés suivantes :

– l'absence d'erreurs à l'exécution liées à la manipulation de la mémoire, tels que des accès de tableaux en dehors des bornes, ou le déréférencement de pointeurs nuls ou invalides ;

– l'absence d'erreurs arithmétiques, mais aussi de comportements non définis dans les calculs arithmétiques, incluant non seulement les divisions par zéro entières, mais aussi les dépassements arithmétiques qui seraient en pratique gérés par l'arithmétique modulaire (le comportement exact du programme dépend alors de l'architecture cible et du compilateur utilisé) ou bien qui généreraient des infinis ou NaN (*Not a Number*) en virgule flottante [IEE 85] ;

– l'absence de comportements indéfinis selon la norme du langage C, comme dans le cas où le résultat d'une expression dépend de l'ordre d'évaluation ;

– la correction d'assertions (expressions booléennes C arbitraires) définies par l'utilisateur ; ces assertions offrent à l'utilisateur un mécanisme très souple pour définir ses propres conditions de sûreté à faire vérifier par ASTRÉE, comme le fait que des valeurs de sortie du programme sont bien dans l'intervalle souhaité, ou la validité d'invariants de programmes complexes.

2.2.3. *Exemples*

Par essence, l'analyse par ASTRÉE est *correcte* (ASTRÉE découvre tout comportement erroné possible), mais *incomplète* (il peut arriver que des alarmes levées par

ASTRÉE ne correspondent à aucun cas d'erreur réel). Considérons tout d'abord le programme ci-dessous :

```
int x, y; /* x est supposé dans l'intervalle [−100,100] */
if ( x > 0 )
    y = 400 / (x - 5);
```

La spécification (informelle) de ce programme précise que x peut prendre toute valeur dans l'intervalle $[-100, 100]$; en particulier, si x prend la valeur 5, l'évaluation de l'expression dans l'affectation provoquera à coup sûr une erreur (division par zéro). Une analyse de ce programme par ASTRÉE indiquera une *alarme*, à savoir un message signifiant que le programme ne peut pas être prouvé correct ; il incombe alors à l'utilisateur de vérifier à partir de ce message si une erreur est possible, et si il y a lieu de corriger le programme. Le message donné par l'analyseur aide significativement à cela en précisant, entres autres, la nature, la position et le contexte d'appel de l'alarme. ASTRÉE peut, de plus, à la demande de l'utilisateur, donner les informations inférées pour toute variable en tout point de programme.

Par ailleurs, l'approche suivie par ASTRÉE est incomplète (cela est nécessairement le cas, puisque le problème consistant à déterminer l'absence d'erreurs à l'exécution dans un programme est *indécidable* et qu'ASTRÉE est à la fois correct et automatique). Nous ne pouvons donc pas nous attendre à ce qu'ASTRÉE établisse de manière totalement automatique la correction de tout programme correct selon les propriétés de correction mentionnées en section 2.2.2. Considérons, par exemple, le programme ci-dessous :

```
int x,y; /* dans l'intervalle [0,10000] */
if ( x * x == 7 * y * y - 1 )
    y = 8 / 0;
```

Dans ce programme, l'affectation à l'intérieur de la conditionnelle provoquera une erreur à coup sûr si elle est exécutée. La seule solution pour prouver que ce programme est correct est de montrer que la condition s'évaluera toujours en faux, ce qui est le cas car la racine carrée de 7 n'est pas un nombre rationnel (notons qu'il n'y a pas de dépassement de capacité arithmétique étant donné l'intervalle choisi pour x et y). Or, le fait que l'équation $x^2 = 7y^2 - 1$ n'a pas de solution est un théorème mathématique non trivial. Il n'est pas raisonnable d'espérer qu'un analyseur non spécifiquement conçu pour résoudre les équations diophantiennes (apparaissant rarement dans les programmes embarqués synchrones ciblés par ASTRÉE) puisse prouver ce théorème (sauf à tester toutes les paires (x, y) de $[0, 10000]^2$, ce qui est simple mais trop coûteux). L'analyseur ASTRÉE ne gère donc pas ce type d'exemple, ainsi que beaucoup d'autres présentant des problèmes similaires.

Néanmoins, cela n'est pas une limitation réellement gênante. En effet, ASTRÉE n'est pas conçu pour apporter une réponse précise pour n'importe quelle application

écrite en C, mais pour des familles d'applications particulières présentant un grand intérêt pratique, i.e., des applications critiques embarquées ; par conséquent l'analyse est optimisée pour de tels programmes. Elle demeure incomplète, et il y aura donc toujours des cas pour lesquels la réponse fournie par l'analyseur n'est pas suffisamment précise. Cependant, ASTRÉE est conçu de manière à pouvoir être étendu avec des domaines abstraits plus expressifs, de manière à être à même d'améliorer l'analyse si nécessaire. Ainsi, il est possible de rendre l'analyseur très précis en pratique sur certaines familles de programmes intéressants, tout en restant incomplet en général.

2.3. Analyse statique par interprétation abstraite

Dans cette partie, nous présentons les fondements de l'analyse statique par interprétation abstraite, telle qu'elle est mise en oeuvre dans ASTRÉE.

2.3.1. *Sémantique concrète*

À l'instar de [COU 78a], nous utilisons une *sémantique opérationnelle à petits pas* pour décrire les comportements des programmes. Un programme est défini par son ensemble d'états S, sa relation de transition $t \subseteq S \times S$ et l'ensemble $I \subseteq S$ de ses états initiaux. Étant donné un état initial $s_0 \in I$, un état successeur est un état $s_1 \in S$ tel que $\langle s_0, s_1 \rangle \in t$. Les états suivants dans l'exécution $s_1 s_2 \ldots s_n \ldots$ sont définis de même : $\langle s_i, s_{i+1} \rangle \in t$. Une exécution peut soit ne pas terminer soit s'arrêter sur un état qui n'a pas de successeur (dans le cas d'une erreur ou de la terminaison normale du programme). La relation de transition t est le plus souvent non déterministe ce qui signifie qu'un état $s \in S$ peut avoir plusieurs successeurs $s' \in S : \langle s, s' \rangle \in t$ (e.g., lors de la lecture d'une entrée). La *sémantique des traces maximales* est donc définie par $\mathcal{T}[\![t]\!]I \triangleq \{\langle s_0 \ldots s_n \rangle \mid n \geqslant 0 \wedge s_0 \in I \wedge \forall i \in [0, n) : \langle s_i, s_{i+1} \rangle \in t \wedge \forall s' \in S : \langle s_n, s' \rangle \notin t\} \cup \{\langle s_0 \ldots s_n \ldots \rangle \mid s_0 \in I \wedge \forall i \geq 0 : \langle s_i, s_{i+1} \rangle \in t\}$ et décrit l'ensemble des traces les plus longues partant d'un état initial, satisfaisant la relation de transition à chaque pas et qui soit se terminent par un état final, soit ne se terminent pas.

En pratique, la sémantique des traces maximales $\mathcal{T}[\![t]\!]I$ d'un programme décrit par un système de transitions $\langle S, t, I \rangle$ n'est ni calculable ni observable (par une machine ou un utilisateur). Toutefois, si nous considérons des propriétés de *sûreté* uniquement, alors nous pouvons nous limiter aux exécutions finies, de longueur non bornée. Cette observation est décrite formellement par *la sémantique des préfixes finis de traces* $\mathcal{P}[\![t]\!]I \triangleq \{\langle s_0 \ldots s_n \rangle \mid s_0 \in I \wedge \forall i \in [0, n) : \langle s_i, s_{i+1} \rangle \in t\}$. Cette sémantique est à la fois plus simple et suffisante pour déterminer toute propriété de sûreté, c'est-à-dire dont l'échec peut être constaté en observant l'exécution du programme. Par conséquent, nous la choisissons comme *sémantique collectrice* qui est, par définition, la plus forte propriété considérée, définissant ainsi l'expressivité des analyses statiques considérées. Dans l'idéal, calculer cette sémantique permettrait de déterminer toute

propriété de sûreté, mais cela n'est pas automatisable car la sémantique collectrice n'est pas calculable, sauf dans le cas particulier des systèmes finis de petite taille. Par conséquent, nous utiliserons des abstractions de cette sémantique afin de fournir des réponses correctes mais incomplètes.

Le choix de la sémantique collectrice dépend du problème considéré et du niveau d'observation des comportements du programme. Ainsi, si nous considérons des propriétés *d'invariance* uniquement, nous pouvons choisir comme sémantique collectrice l'ensemble des états *accessibles*, défini par $\mathcal{R}[\![t]\!]I \triangleq \{s' \mid \exists s \in I : \langle s, s'\rangle \in t^\star\}$ (où $t^\star$ désigne la *clôture réflexive et transitive* de t, définie par $t^\star = \bigcup_{n\geq 0} t^n$ où, pour tout $n \geqslant 0$, $t^n \triangleq \{\langle s, s'\rangle \mid \exists s_0 \ldots s_n : s_0 = s \wedge \forall i \in [0, n) : \langle s_i, s_{i+1}\rangle \in t \wedge s_n = s'\}$). Cette sémantique est plus abstraite que la sémantique des préfixes finis car il est possible de déduire l'ensemble des états accessibles à partir des préfixes de traces, mais pas l'inverse, l'ensemble des états accessibles oubliant l'ordre d'exécution. Ainsi, supposons que nous voulions déterminer si l'état s_1 apparaît toujours avant l'état s_2 dans toute exécution, avec pour seule information l'ensemble des états accessibles. Alors, si s_1 et s_2 sont tous les deux accessibles, il est impossible de déterminer si la propriété est vraie, ce qui constitue un exemple d'incomplétude de l'ensemble des états accessibles par rapport à la sémantique des préfixes finis.

Les sémantiques peuvent être exprimées comme des points fixes [COU 77b]. Par exemple, $t^\star$ est la plus petite solution pour $\subseteq$ des équations $X = 1_S \cup (X \circ t)$ et $X = 1_S \cup (t \circ X)$, où $1_S \triangleq \{\langle s, s\rangle \mid s \in S\}$ est l'identité sur S, et $\circ$ est la composition de relations $t \circ r \triangleq \{\langle s, s''\rangle \mid \exists s' : \langle s, s'\rangle \in t \wedge \langle s', s''\rangle \in r\}$. Nous écrivons donc $t^\star = \mathrm{lfp}^{\subseteq} F$ où $F(X) \triangleq 1_S \cup (X \circ t)$. Cela signifie que $t^\star$ est un point fixe de F (i.e., $t^\star = F(t^\star)$) et que parmi tous les points fixes de F, c'est le plus petit (si $X = F(X)$ alors $t^\star \subseteq X$). L'existence de tels points fixes est une conséquence du théorème de Tarski [TAR 55]. De plus, un plus petit point fixe peut être calculé de manière itérative. En commençant à partir de $\emptyset$, nous obtenons $X^0 \triangleq 1_S \cup (\emptyset \circ t) = 1_S = t^0$, $X^1 \triangleq 1_S \cup (X^0 \circ t) = 1_S \cup t$, $X^2 \triangleq 1_S \cup (X^1 \circ t) = 1_S \cup t \cup t^2$, etc. Par récurrence, nous voyons que $\forall n \geqslant 0 : X^n = \bigcup_{i=0}^{n} t^i$, et donc que $X^\star \triangleq \bigcup_{n\geq 0} X^n = \bigcup_{n\geq 0} \bigcup_{i=0}^{n} t^i = \bigcup_{n\geq 0} t^n$ qui est effectivement $t^\star$. Le calcul de $\mathcal{R}[\![t]\!]I$ et $\mathcal{P}[\![t]\!]I$ se fait de la même manière :

$$
\begin{aligned}
&\mathcal{R}[\![t]\!]I = \mathrm{lfp}^{\subseteq} F_R \\
&\qquad \text{où } F_R(X) \triangleq I \cup \{s' \mid \exists s \in X : \langle s, s'\rangle \in t\}, \\
&\mathcal{P}[\![t]\!]I = \mathrm{lfp}^{\subseteq} F_P \\
&\qquad \text{où } F_P(X) \triangleq \{\langle s_0\rangle \mid s_0 \in I\} \\
&\qquad\qquad\qquad \cup \{\langle s_0 \ldots s_n s_{n+1}\rangle \mid \langle s_0 \ldots s_n\rangle \in X \wedge \langle s_n, s_{n+1}\rangle \in t\} \ .
\end{aligned}
$$

2.3.2. *Abstraction*

Une abstraction [COU 77b] relie une sémantique concrète et une sémantique abstraite et assure qu'une propriété prouvée dans l'abstrait est aussi vraie dans le concret (c'est la *correction*), tandis qu'une propriété qui est vraie dans le concret n'est pas forcément prouvable dans l'abstrait (*incomplétude*). Par exemple, le lien entre la sémantique concrète des préfixes finis et l'ensemble des états accessibles s'exprime sous la forme d'une abstraction $\mathcal{R}[\![t]\!]I = \alpha_R(\mathcal{P}[\![t]\!]I)$ où $\alpha_R(X) \triangleq \{s \mid \exists\langle s_0 \ldots s_n\rangle \in X : s = s_n\}$. Cette abstraction oublie l'ordre des exécutions et ne retient que l'ensemble des états rencontrés.

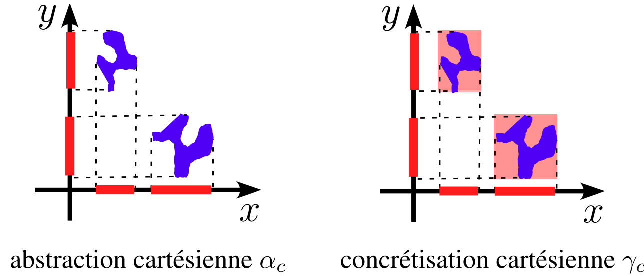
D'un autre côté, la *fonction de concrétisation* $\gamma_R(X) \triangleq \{\langle s_0 \ldots s_n\rangle \in X \mid \forall i \in [0, n] : s_i \in X\}$ reconstruit les préfixes finis de traces à partir de l'ensemble des états accessibles, mais considère qu'ils peuvent apparaître dans n'importe quel ordre, l'ordre d'exécution ayant été oublié. Par conséquent, $\mathcal{P}[\![t]\!]I \subseteq \gamma_R(\mathcal{R}[\![t]\!]I)$, c'est pourquoi nous dirons que l'abstrait "sur-approxime" le concret : la concrétisation de l'abstrait présente plus de comportement possibles que le concret. La *correction* est ainsi garantie : si une propriété est vraie dans l'abstrait alors elle est aussi vraie dans le concret (un comportement qui n'apparaît pas dans l'abstrait n'apparaît pas dans le concret). À l'inverse, l'*incomplétude* découle du fait que, si un comportement apparaît dans l'abstrait, nous ne pouvons pas être sûr qu'il existe aussi dans le concret.

Le couple $\langle\alpha_R, \gamma_R\rangle$ définit une *correspondance de Galois* $\langle\alpha, \gamma\rangle$ [COU 79] car

$$\forall T, R : \alpha(T) \subseteq^\sharp R \iff T \subseteq \gamma(R)$$

où l'inclusion abstraite $\subseteq^\sharp$ est un ordre partiel tel que $x \subseteq^\sharp y$ implique que $\gamma(x) \subseteq \gamma(y)$ (i.e., $\subseteq^\sharp$ est la version abstraite de l'implication logique). Les correspondances de Galois ont de nombreuses propriétés. En particulier, toute propriété concrète T a une meilleure abstraction $\alpha(T)$, c'est-à-dire que non seulement $\alpha(T)$ sur-approxime T (i.e., $T \subseteq \gamma(\alpha(T))$) mais, de plus, toute autre sur-approximation R de T dans l'abstrait (c'est-à-dire, telle que $T \subseteq \gamma(R)$) est moins précise que $\alpha(T)$ (i.e., $\alpha(T) \subseteq^\sharp R$).

L'abstraction cartésienne, où un ensemble de couples est abstrait par le couple de ses projections, constitue un autre exemple de correspondance de Galois :



abstraction cartésienne α_c concrétisation cartésienne γ_c

Étant donné un ensemble de couples X, son abstraction cartésienne $\alpha_c(X)$ est $\langle\{x \mid \exists y : \langle x, y\rangle \in X\}, \{y \mid \exists x : \langle x, y\rangle \in X\}\rangle$. La concrétisation est définie par $\gamma_c(\langle X, Y\rangle) \triangleq \{\langle x, y\rangle \mid x \in X \wedge y \in Y\}$. L'ordre abstrait $\subseteq_c$ est l'inclusion respective de chaque composante.

Par contre, certaines abstractions ne peuvent être décrites par une correspondance de Galois. Dans un tel cas, nous utiliserons la fonction de concrétisation seule [COU 92]. Le contre-exemple classique est celui d'un disque qui n'a pas de meilleure approximation dans le domaine des polyèdres convexes [COU 78b], comme l'a prouvé Euclide [Euc 00].

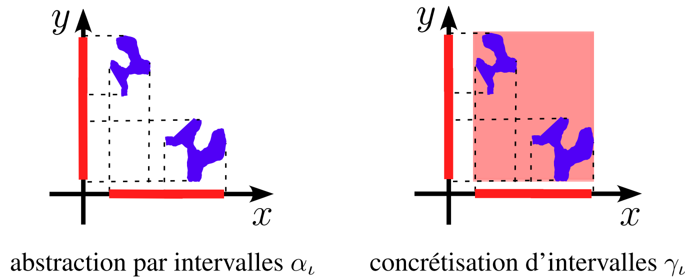
Nous avons vu que l'ensemble des états accessibles est plus abstrait que la sémantique des préfixes de traces. D'autres paires d'abstractions ne sont pas comparables, comme la parité et le signe des nombres entiers. La relation "être plus abstrait que" définit en réalité un treillis sur les propriétés abstraites [COU 77b, COU 79] : il existe une meilleure abstraction plus abstraite que deux abstractions données, ainsi qu'une abstraction la plus grossière qui est plus précise que deux abstraction données. Le problème consistant à choisir le bon niveau d'abstraction pour traiter un problème d'analyse est donc une question pratique fondamentale. Considérons une sémantique $\mathcal{S}$ et une abstraction spécifiée par une concrétisation γ. Nous chercherons alors des sémantiques abstraites $\mathcal{S}^\sharp$ *correctes*, c'est-à-dire telles que $\mathcal{S} \subseteq \gamma(\mathcal{S}^\sharp)$. Intuitivement, cela signifie qu'aucun cas concret n'est oublié dans l'abstrait (i.e., il ne peut y avoir de faux négatifs). La sémantique abstraite peut aussi se révéler *complète*, si $\mathcal{S} \supseteq \gamma(\mathcal{S}^\sharp)$; il n'y alors dans l'abstrait aucun comportement qui n'apparaît pas dans le concret (i.e., il ne peut y avoir de faux positifs). Le cas idéal est celui d'une abstraction correcte et complète. Dans le domaine de la preuve manuelle de programmes, les abstractions utilisées sont correctes et complètes. À l'opposé, dans le domaine de l'analyse statique, nous utilisons des abstractions correctes mais incomplètes, comme l'abstraction cartésienne vue plus haut, le domaine des intervalles (section 2.3.3) ou les systèmes de types [COU 97]. Ceci permet d'obtenir des abstractions calculables. En contrepartie, cela peut conduire à rejeter des programmes dont toutes les exécutions sont pourtant correctes.

2.3.3. *Sémantique abstraite*

Nous considérons à présent le calcul d'une sémantique abstraite. Comme la sémantique concrète $\mathcal{S}$ s'exprime sous la forme du plus petit point fixe $\text{lfp}^{\subseteq} F$ de la fonction concrète F, il est naturel d'exprimer la sémantique abstraite aussi comme le plus petit point fixe $\mathcal{S}^\sharp = \text{lfp}^{\subseteq^\sharp} F^\sharp$ d'une fonction abstraite $F^\sharp$ (où $X \subseteq^\sharp Y$ implique $\gamma(X) \subseteq \gamma(Y)$).

La fonction abstraite $F^\sharp$ est *correcte* si $F \circ \gamma \subseteq \gamma \circ F^\sharp$. Dans le cas où une correspondance de Galois existe, la meilleure fonction abstraite existe et est définie par $F^\sharp \triangleq \alpha \circ F \circ \gamma$. Si il n'y a pas de correspondance de Galois, ou si la meilleure fonction abstraite est trop complexe (non calculable ou trop coûteuse), nous nous contentons d'une fonction abstraite correcte. Si $\mathcal{S} = \text{lfp}^{\subseteq} F$, $F \circ \gamma \subseteq \gamma \circ F^\sharp$ et, sous les bonnes hypothèses, la sémantique abstraite $\mathcal{S}^\sharp = \text{lfp}^{\subseteq^\sharp} F^\sharp$ est correcte et permet donc de prouver des propriétés sur les programmes : si $\mathcal{S}^\sharp \subseteq^\sharp P^\sharp$, alors $\mathcal{S} \subseteq \gamma(P^\sharp)$, ce qui signifie que, si la propriété abstraite $P^\sharp$ est vraie dans l'abstrait, alors elle est également vérifiée par la sémantique concrète.

Nous considérons à présent un programme ayant un ensemble $\mathtt{V}$ de variables numériques (entières, flottantes, etc.). Un état $s \in S$ associe donc une valeur numérique $s(\mathtt{x})$ à chaque variable $\mathtt{x} \in \mathtt{V}$ du programme. Un problème classique consiste à déterminer les intervalles de variation de chaque variable, pour toute exécution du programme, ce qui revient à appliquer l'abstraction des états accessibles, puis l'abstraction cartésienne et enfin l'abstraction des intervalles :



abstraction par intervalles α_ι concrétisation d'intervalles γ_ι

L'abstraction des intervalles est $\alpha_\iota(\mathcal{R}[\![t]\!]I)$ où $\alpha_\iota(X)(\mathtt{x}) = [\min_{s\in X} s(\mathtt{x}), \max_{s\in X} s(\mathtt{x})]$ qui vaut $\emptyset$ si X est vide et peut avoir des bornes infinies ($-\infty$ ou $+\infty$) en l'absence d'un minimum ou maximum. Cette abstraction est infinie en ce sens que, par exemple, $[0,0] \subset [0,1] \subset [0,2] \subset \cdots \subset [0,+\infty)$. En supposant l'ensemble des valeur numériques borné (e.g., $-\infty = \mathtt{minint}$ et $+\infty = \mathtt{maxint}$ pour les entiers), cela donnerait

un ensemble d'états fini, mais très grand, entraînant une explosion combinatoire du nombre de cas à considérer.

L'intérêt pratique consistant à considérer des abstractions infinies apparaît même dans des exemples très simples comme le programme $P \equiv$ `x=0; while (x<=`n`) { x=x+1; }`, où $n \geqslant 0$ désigne une constante numérique positive arbitraire. Pour tout entier n, un analyseur statique par interprétation abstraite pourra déterminer que, à l'entrée du corps de la boucle, `x` $\in [0, n]$. Or, en se limitant à des abstractions finies, il n'est possible d'utiliser qu'un nombre fini d'intervalles, et il n'est donc pas possible de traiter tous ces programmes. Une autre solution consisterait à considérer une suite infinie d'abstraction finies et de précision croissante, et d'exécuter des analyses finies jusqu'à l'obtention d'une réponse précise (par exemple [CLA 00, COU 07b]), mais cela serait trop coûteux, et la terminaison ne serait pas garantie pour tous les programmes.

Le développement d'un analyseur statique nécessite le choix de structures de données pour représenter les propriétés abstraites et l'implantation de fonctions d'analyse. Encoder les propriétés abstraites de manière uniforme (comme des termes dans un outil de démonstration automatique ou des BDDs [BRY 86]) simplifierait considérablement le développement et la réutilisation d'analyseurs, mais limiterait de fait l'efficacité des analyses en rendant impossible l'utilisation de structures de données compactes et d'algorithmes efficaces, bien adaptés à des classes spécifiques de propriétés. Par exemple, il est plus intéressant de représenter un intervalle implicitement par ses bornes, plutôt que par l'ensemble en extension de ses valeurs. Par conséquent, les analyseurs statiques par interprétation abstraite utilisent de nombreux *domaines abstraits*, qui sont des algèbres du point de vue du mathématicien, et des modules du point de vue des informaticiens. Un domaine abstrait définit une représentation (e.g., soit $\emptyset$ soit un couple $[\ell, h]$ de nombres finis ou infinis tels que $\ell \leq h$ pour le domaine des intervalles) et des *fonctions de transfert* élémentaires permettant d'approximer les fonctions concrètes F_P, F_R, etc. Dans le cas du domaine des intervalles, les fonctions abstraites usuelles incluent le test d'inclusion ($\emptyset \subseteq \emptyset \subseteq [a, b]$, $[a, b] \subseteq [c, d]$ si et seulement si $c \leqslant a$ et $b \leqslant d$), l'addition d'intervalles ($\emptyset + \emptyset = \emptyset, \emptyset + [a, b] = [a, b] + \emptyset = \emptyset$ et $[a, b] + [c, d] = [a + c, b + d]$), l'union ($\emptyset \cup \emptyset = \emptyset, \emptyset \cup [a, b] = [a, b] \cup \emptyset = [a, b]$ et $[a, b] \cup [c, d] = [\min(a, c), \max(b, d)]$. Des bibliothèques de domaines abstraits numériques telles qu'APRON [JEA 07, JEA 09] sont à la disposition des développeurs d'analyseurs statiques et implantent déjà toutes ces opérations.

2.3.4. *Opérateurs d'extrapolation*

Jusqu'ici, nous avons considéré le calcul d'approximations dans l'abstrait pour des opérations simples. Néanmoins, le calcul approché d'une sémantique concrète définie sous la forme d'un plus petit point fixe pose un autre problème : la terminaison de l'analyse.

Considérons le programme ci-dessous :

```
int x = 1;
while (1)
    x = x + 2;
```

L'intervalle de variation de x est le plus petit intervalle X solution de l'équation $X = F_\iota(X)$ où $F_\iota(X) \triangleq [1,1] \cup (X + [2,2])$. Résoudre cette équation de manière itérative revient à calculer les itérés $X^0 = \emptyset$, $X^1 = [1,1] \cup (X^0 + [2,2]) = [1,1]$, $X^2 = [1,1] \cup (X^1 + [2,2]) = [1,1] \cup [3,3] = [1,3]$, et ainsi de suite, ce qui, après une infinité d'itérés et un passage à la limite permet d'obtenir la solution $[1,+\infty)$. Bien évidemment, ce raisonnement ne peut être automatisé de la sorte, dans la mesure où un ordinateur ne peut ni effectuer en temps fini le calcul d'une infinité d'itérés, ni faire preuve d'intuition et passer à la limite. Une solution consiste à accélérer la convergence en utilisant un *opérateur d'élargissement* [COU 77b, COU 78a] (*widening*), noté ∇, capable de faire une induction abstraite. Pour cela, nous résolvons $X = X \nabla F_\iota(X)$ au lieu de $X = F_\iota(X)$. L'élargissement définit un élément abstrait $X^{n+1} \triangleq X^n \nabla F_\iota(X^n)$ par extrapolation à partir des deux itérés précédents X^n et $F_\iota(X^n)$. Cette extrapolation doit calculer une sur-approximation de tous les $F_\iota(X^n)$, $n \geq 0$, en un nombre fini d'itérations ($X^k = X^{k+1}$ pour un certain k).

Un exemple d'élargissement pour les intervalles est défini par $x \nabla \emptyset = x$, $\emptyset \nabla x = x$, $[a,b] \nabla [c,d] \triangleq [\ell,h]$, où $\ell = -\infty$ si $c < a$ et $\ell = a$ si $a \leqslant c$; $h = +\infty$ si $b < d$ et $h = b$ si $d \leqslant b$. L'intervalle élargi est effectivement plus grand que les arguments (correction). De plus, cette définition empêche les itérations croissantes infinies (comme [0,0], [0,1], [0,2], etc.) en remplaçant par l'infini les bornes instables. Dans le cas de l'équation $X = X \nabla F_\iota(X)$ où $F_\iota(X) \triangleq [1,1] \cup (X + [2,2])$, la suite d'itérés converge à présent en deux itérations vers la limite $[1,+\infty)$:

$$
\begin{array}{lcl}
X^0 & = & \emptyset \\
X^1 & = & X^0 \nabla ([1,1] \cup (X^0 + [2,2])) = \emptyset \nabla [1,1] = [1,1] \\
X^2 & = & X^1 \nabla ([1,1] \cup (X^1 + [2,2])) = [1,1] \nabla [1,3] = [1,+\infty) \\
X^3 & = & X^2 \nabla ([1,1] \cup (X^2 + [2,2])) = [1,+\infty) \nabla [1,+\infty) = [1,+\infty) \\
 & = & X^2
\end{array}
$$

Considérons à présent une variante du programme précédent :

```
int x = 1;
while (x <= 99)
    x = x + 2;
```

Dans ce cas, une approximation de l'intervalle de `x` à l'entrée du corps de la boucle est obtenue en itérant l'équation $X = X \triangledown F_\iota(X)$ où $F_\iota(X) \triangleq [1,1] \cup ((X + [2,2]) \cap (-\infty, 99])$. De même que précédemment, après deux itérations, nous obtenons l'intervalle $[1, +\infty)$, ce qui ne semble pas suffisamment précis, puisque, dans le concret, la valeur de `x` est effectivement bornée par 99. Après une suite croissante d'itérés débouchant sur une sur-approximation du plus petit point fixe concret, il est possible de continuer avec le calcul d'une suite d'itérés décroissants, sans élargissement, ce qui permet de regagner en précision.

Afin de garantir la terminaison des itérations décroissantes, nous utilisons un opérateur d'extrapolation Δ appelé *rétrécissement* [COU 77b, COU 78a] (ou *narrowing* faisant une co-induction abstraite), en résolvant une équation de la forme $X = X \,\Delta\, F_\iota(X)$. Δ fournit alors une sur-approximation de l'intersection et garantit la terminaison de l'itération $X^{n+1} = X^n \,\Delta\, F_\iota(X^n)$. Un exemple d'opérateur de rétrécissement dans le domaine des intervalles est défini par $\emptyset \Delta x = \emptyset$, $x \Delta \emptyset = \emptyset$, $[a,b] \Delta [c,d] = [\ell, h]$ où $\ell = c$ si $a = -\infty$ et $\ell = a$ si $a \neq -\infty$ et, de manière similaire $h = d$ si $b = +\infty$ et $h = b$ sinon. Ainsi, les bornes infinies sont améliorées, mais pas les bornes finies, ce qui garantit la terminaison du processus : chaque borne est en effet améliorée au plus une fois. L'intervalle rétréci $[a,b] \,\Delta\, [c,d]$ est plus grand que $[c,d]$, ce qui assure la correction.

Dans le cas du programme ci-dessus, obtenons la suite d'itérations décroissantes suivante en itérant l'équation $Y = Y \,\Delta\, F_\iota(Y)$ à partir de l'approximation du point fixe concret obtenue plus haut par itérations croissantes :

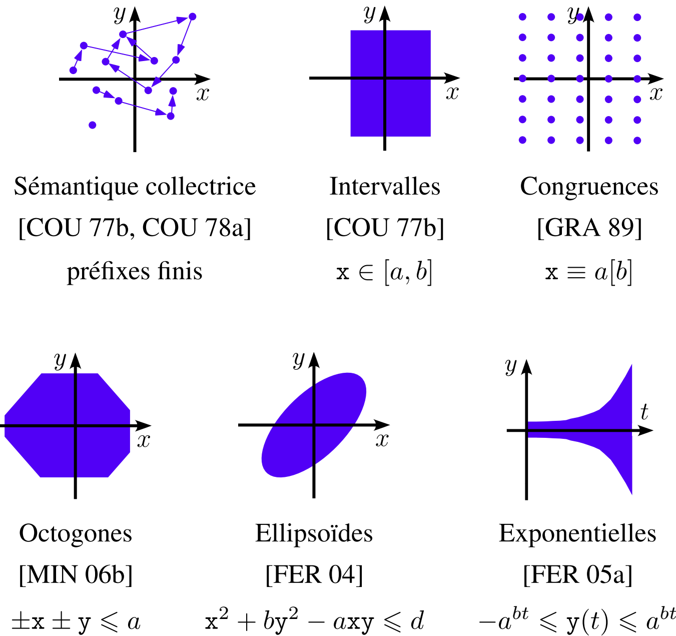
$$
\begin{array}{rcl}
Y^0 & = & [1, +\infty) \\
Y^1 & = & Y^0 \,\Delta\, ([1,1] \cup ((Y^0 + [2,2]) \cap (-\infty, 99])) \\
 & = & [1, +\infty) \,\Delta\, ([1,1] \cup ([3, +\infty) \cap (-\infty, 99])) \\
 & = & [1, +\infty) \,\Delta\, [1, 99] = [1, 99] \\
Y^2 & = & Y^1 \,\Delta\, ([1,1] \cup ((Y^1 + [2,2]) \cap (-\infty, 99])) \\
 & = & [1, 99] \,\Delta\, ([1,1] \cup ([3, 101] \cap (-\infty, 99])) \\
 & = & [1, 99] \,\Delta\, [1, 99] = [1, 99] \\
 & = & Y^1
\end{array}
$$

Le résultat est une sur-approximation du plus petit point fixe concret qui n'est pas forcément la meilleure possible, en général.

Les opérateurs d'extrapolation (élargissements et rétrécissements) ne sont pas indispensables pour les abstractions finies (sans chaîne strictement croissante infinie), puisque les itérations croissantes et décroissantes convergent toujours ; néanmoins, il peut s'avérer judicieux d'utiliser de tels opérateurs afin d'accélérer la convergence, même si cela rend en général le calcul abstrait incomplet.

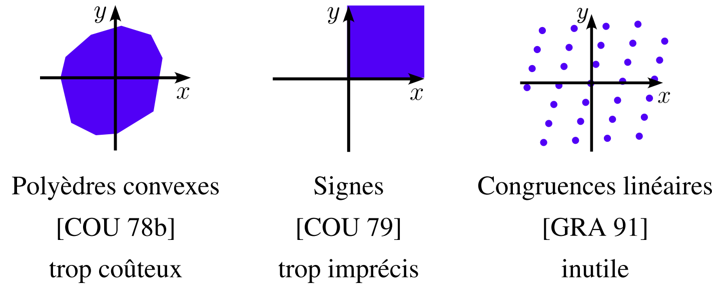
2.3.5. *Domaines abstraits fondamentaux*

Pour chaque outil d'analyse par interprétation abstraite, il faut choisir un domaine abstrait adapté au problème considéré. En pratique, il est généralement plus simple de construire un tel domaine en combinant des domaines existants, plutôt que de le définir d'un seul bloc. Il est ainsi possible de réutiliser des domaines plus simples. En particulier, il est utile de manipuler des conjonctions de propriétés exprimées par des domaines abstraits plus simples, au sein d'un domaine composé (nous reviendrons sur ce point dans la section 2.3.6). À titre d'exemple, la figure ci-dessous présente quelques uns des domaines abstraits de base utilisés dans ASTRÉE [COU 07a] (qui fournissent des abstractions différentes de la sémantique collectrice des préfixes finis) :



Ces domaines abstraits seront décrits dans la section 2.4.

À l'inverse, d'autres domaines pourtant classiques ne constituent pas des choix pertinents dans le contexte d'ASTRÉE, soit parce qu'ils ne permettent pas de passer à l'échelle, soit parce qu'ils expriment des propriétés qui ne sont pas utiles pour les analyses que nous souhaitons effectuer :



2.3.6. *Combinaison de domaines abstraits*

Comme nous l'avons vu plus haut, chaque domaine abstrait utilise une représentation spécifique et optimisée. Par conséquent, il n'est pas trivial de combiner plusieurs domaines pour en construire un nouveau, plus expressif.

2.3.6.1. *Produit réduit*

En particulier, il est possible de manipuler des conjonctions de propriétés abstraites exprimées au sein de domaines différents à l'aide d'un *produit* de domaines. Ainsi, si D_1 et D_2 sont deux domaines abstraits définissant des fonctions de concrétisation γ_1 et γ_2, alors nous pouvons définir le domaine produit $D = D_1 \times D_2$ avec la concrétisation γ définie par $\forall \langle p_1, p_2 \rangle \in D, \gamma(\langle p_1, p_2 \rangle) = \gamma(p_1) \cap \gamma(p_2)$. Dans ce nouveau domaine, $\langle p_1, p_2 \rangle$ représente exactement la conjonction des propriétés abstraites représentées par p_1 et p_2.

Toutefois, cette représentation n'est généralement pas optimale, dans la mesure où il existe souvent plusieurs paires d'éléments abstraits décrivant une même propriété concrète. Un exemple classique est obtenu en considérant le plus petit élément de chaque domaine : en supposant que $\bot_1 \in D_1$ et $\bot_2 \in D_2$ sont tels que $\gamma_1(\bot_1) = \gamma_2(\bot_2) = \emptyset$, alors $\forall p_1 \in D_1, \forall p_2 \in D_2, \gamma(\langle p_1, \bot_2 \rangle) = \gamma(\langle \bot_1, p_2 \rangle) = \emptyset$. Une telle redondance est source fréquente de pertes de précision ou d'efficacité au cours des analyses. Par exemple, pour le plus petit élément, le test du vide, qui renvoie “vrai” si une valeur abstraite représente $\emptyset$ et “faux” sinon est plus complexe. Plus généralement, si nous souhaitons effectuer une analyse par interprétation abstraite pour déterminer de telles conjonctions de propriétés, il n'est pas optimal de réaliser un simple produit d'analyses. Supposons par exemple que nous disposons d'une collection de domaines abstraits et que nous pouvons calculer $\text{lfp}^{\subseteq} F_1 \in D_1$, $\dots$, $\text{lfp}^{\subseteq} F_n \in D_n$ dans chacun de ces domaines $D_1, \dots, D_n$, de manière correcte par rapport à la sémantique collectrice $\mathcal{C}[\![t]\!]I$. La combinaison de ces analyses est correcte car $\mathcal{C}[\![t]\!]I \subseteq \gamma_1(\text{lfp}^{\subseteq} F_1) \cap \dots \cap \gamma_n(\text{lfp}^{\subseteq} F_n)$. Toutefois, cette combinaison d'analyses

n'est pas précise, puisqu'elle ne permet pas à chaque analyse d'utiliser les résultats des autres analyses afin d'améliorer son propre calcul. Considérons, par exemple, des analyses d'intervalle et de parité, qui découvrent que $\mathtt{x} \in [0, 100]$ et que x est *impair* durant leur première itération. Alors, en croisant ces résultats, l'intervalle découvert peut être amélioré en $\mathtt{x} \in [1, 99]$, puisque les valeurs 0 et 100 sont impossibles car paires. Ce gain de précision peut, à son tour, rendre la suite de l'analyse plus précise, e.g., en évitant un élargissement inutile.

Pour réaliser cette opération consistant à améliorer la précision des invariants manipulés au cours de l'analyse en utilisant les informations découvertes par un domaine abstrait pour raffiner celles découvertes par les autres domaines, nous utilisons un opérateur de réduction ρ. Le principe de cet opérateur est de tenter de se ramener à des représentations abstraites canoniques. Un tel opérateur doit en particulier être correct, c'est-à-dire ne pas oublier de comportement concret. L'opérateur de réduction pour le produit des analyses d'intervalles et de congruences pourra en particulier améliorer la précision de la manière suivante : $\rho(\langle [0, 100], impair \rangle) = \langle [1, 99], impair \rangle$. Dans le cas où les domaines abstraits de base définissent des correspondances de Galois, l'opération la plus précise est définie et s'appelle le *produit réduit* [COU 79] de domaines. En général, il n'est pas calculable ou bien trop coûteux, et nous devons nous contenter d'un opérateur de réduction approché, comme celui décrit en section 2.4.6.

2.3.6.2. *Partitionement*

Une autre technique particulièrement utile pour concevoir des domaines abstraits composés à partir de domaines abstraits plus simples consiste à appliquer un *partitionement* [COU 81]. L'intérêt de cette technique est d'éviter d'abstraire "en un seul bloc" un ensemble complexe d'états concrets et d'utiliser, au contraire, un ensemble fini d'ensembles d'états concrets plus simples, pour lesquels il est plus facile de trouver des abstractions correctes et précises. Considérons une sémantique collectrice définie comme l'ensemble $C = \wp(S)$ des partie d'un ensemble, ainsi qu'une *partition* finie $S_1, \ldots, S_n$ de S. Chaque partie $\wp(S_i)$ sera approximée à l'aide d'un domaine abstrait D_i (éventuellement le même pour tous les $\wp(S_i)$). Un élément abstrait du domaine partitionné est alors un n-upplet $\langle d_1, \ldots, d_n \rangle \in D_1 \times \cdots \times D_n$ dont la concrétisation est définie par $\gamma(\langle d_1, \ldots, d_n \rangle) \triangleq (\gamma_1(d_1) \cap S_1) \cup \cdots \cup (\gamma_n(d_n) \cap S_n)$. Par exemple, nous pouvons raffiner le domaine des intervalles en maintenant deux intervalles, l'un représentant les valeurs positives, l'autre les valeurs négatives. Ce nouveau domaine permet de représenter exactement certains ensembles disjoints, tel que l'ensemble des entiers non nuls $(-\infty, -1] \cup [1, +\infty)$, tandis que le domaine initial ne permet pas cela.

Cette technique peut être étendue de nombreuses manières. Il est par exemple possible de choisir une *famille recouvrante* $S_1, \ldots, S_n$ de S, plutôt qu'une partition. Dans ce cas, un même élément concret sera effectivement représenté par plusieurs éléments abstraits. Il est également possible de recouvrir S à l'aide d'une famille infinie d'ensembles $(S_i)_{i \in N}$, à condition que chaque élément de $\wp(S)$ puisse être recouvert par

une union finie d'éléments de $(S_i)_{i \in N}$ (cela offre plus de flexibilité à l'analyse dans le choix des représentations). Enfin, la famille $(S_i)_{i \in N}$ peut être définie elle-même par un domaine abstrait [BOU 92].

Nous verrons dans la suite de nombreux exemples de partitionements, utilisés pour l'abstraction d'états de contrôle (section 2.4.1), de traces d'exécution (section 2.4.2), et d'états mémoire (section 2.4.4).

2.3.7. *Application à l'analyse statique et à la vérification de programmes*

L'analyse statique vise à répondre à des questions relatives aux états accessibles afin, par exemple, de démontrer qu'aucun état d'erreur ne peut être atteint, pour aucune exécution. Afin de répondre à de telles questions, un analyseur statique calcule une sur-approximation de la sémantique collectrice. L'intérêt de cette approche est d'aboutir à des informations complexes sur les exécutions d'un programme, sans demander à l'utilisateur de fournir une spécification précise. Un analyseur statique par interprétation abstraite est construit en composant un ensemble de domaines abstraits, puis en exécutant un calcul abstrait dans le domaine obtenu. Ce calcul est guidé par la définition d'une fonction sémantique abstraite à partir de l'arbre de syntaxe abstrait du programme. La propriété abstraite ainsi calculée peut soit être directement donnée à l'utilisateur, soit être utilisée pour vérifier que le programme satisfait une spécification abstraite.

Une *spécification* est une propriété de la sémantique du programme, généralement définie par rapport à la sémantique collectrice. Par exemple, la *spécification d'accessibilité* est souvent définie par un ensemble d'états dangereux $D \in \wp(S)$, dont aucun ne doit être accessible (les états “sûrs” sont alors les éléments de $S \setminus D$). Une spécification est généralement énoncée à un certain niveau d'abstraction ; par exemple, une *spécification d'intervalle de variation* exprime que l'ensemble des valeurs qu'une variable x peut prendre au cours de toute exécution du programme doit être compris dans un intervalle $[\ell_{\mathtt{x}}, h_{\mathtt{x}}]$. Le processus de *vérification* consiste à prouver que la sémantique collectrice implique la spécification. Par exemple, la spécification d'accessibilité s'énonce $\mathcal{R}[\![t]\!]I \subseteq (S \setminus D)$, ce qui signifie qu'aucune exécution ne peut atteindre un état dangereux. La spécification étant définie à un certain niveau d'abstraction, il n'est pas indispensable de la vérifier dans le concret. Ainsi, la spécification d'intervalle de variation nécessite la vérification de la propriété : $\forall \mathtt{x} \in \mathtt{V} : \alpha_\iota(\mathcal{R}[\![t]\!]I)(\mathtt{x}) \subseteq [\ell_{\mathtt{x}}, h_{\mathtt{x}}]$. Pour cela, il est suffisant de vérifier, en utilisant la sémantique abstraite d'intervalles, que $\mathcal{I}[\![t]\!]I(\mathtt{x}) \subseteq [\ell_{\mathtt{x}}, h_{\mathtt{x}}]$, où $\mathcal{I}[\![t]\!]$ est une sur-approximation de la sémantique d'accessibilité $\mathcal{R}[\![t]\!]$ définie à base d'intervalles. Plus formellement, cela entraîne $\mathcal{I}[\![t]\!]I(\mathtt{x}) \supseteq \alpha_\iota(\mathcal{R}[\![t]\!]I)(\mathtt{x})$. Donc, si nous pouvons prouver $\forall \mathtt{x} \in \mathtt{V} : \mathcal{I}[\![t]\!]I(\mathtt{x}) \subseteq [\ell_{\mathtt{x}}, h_{\mathtt{x}}]$, il vient $\alpha_\iota(\mathcal{R}[\![t]\!]I)(\mathtt{x}) \subseteq [\ell_{\mathtt{x}}, h_{\mathtt{x}}]$, et donc $\forall \mathtt{x} \in \mathtt{V}, \forall s \in \mathcal{R}[\![t]\!]I : s(\mathtt{x}) \in [\ell_{\mathtt{x}}, h_{\mathtt{x}}]$.

De même qu'en mathématiques, pour prouver une propriété donnée, il est souvent nécessaire de prouver d'abord une propriété intermédiaire plus forte, prouver une spécification qui s'exprime à un certain niveau d'abstraction requiert fréquemment l'utilisation d'une abstraction plus précise de la sémantique collectrice. À titre d'exemple, considérons la règle des signes [BRA 28] : $\mathtt{pos} \times \mathtt{pos} = \mathtt{pos}$, $\mathtt{neg} \times \mathtt{neg} = \mathtt{pos}$, $\mathtt{pos} \times \mathtt{neg} = \mathtt{neg}$, etc., où $\gamma(\mathtt{pos}) \triangleq \{z \in \mathbb{Z} \mid z \geqslant 0\}$ et $\gamma(\mathtt{neg}) \triangleq \{z \in \mathbb{Z} \mid z \leqslant 0\}$). L'abstraction des signes est complète pour la multiplication en ce sens que le signe des arguments détermine avec certitude le signe du résultat, mais incomplète pour l'addition : par exemple, le signe de $\mathtt{pos} + \mathtt{neg}$ est inconnu. À l'opposé, lorsque les intervalles des arguments d'une addition sont connus, l'intervalle du résultat, et donc son signe, sont également connus sans incertitude. Ainsi, dans le but de vérifier la correction d'un programme vis-à-vis d'une spécification de signes, il peut se révéler indispensable d'avoir recours à un domaine abstrait d'intervalles.

2.4. Structure d'ASTRÉE

Dans cette section, nous décrivons la structure de l'analyseur ASTRÉE, ainsi que les principaux domaines abstraits utilisés par celui-ci, permettant ainsi de comprendre sa mise en oeuvre sur des programmes spécifiques.

2.4.1. *Structure générale de l'analyseur*

2.4.1.1. *Choix d'une sémantique concrète*

Comme nous l'avons vu dans la section 2.3, avant de concevoir une analyse par interprétation abstraite, il convient de choisir une sémantique collectrice concrète puis, ensuite, une abstraction. Dans le cas d'ASTRÉE, la sémantique concrète du langage analysé est définie par la norme du langage C ISO/IEC 9899 :1999/Cor 3 :2007 [ISO 07] sous la forme d'une sémantique opérationnelle à petits pas, présentée de manière informelle mais relativement détaillée. Cette sémantique ne définit pas complètement le comportement des programmes et laisse de nombreux choix à l'implantation (choix liés au compilateur ou à l'architecture cible), en particulier concernant la représentation en mémoire des types de données (entiers, tableaux, structures, unions) et l'effet des dépassements en arithmétique entière. Les programmes embarqués analysés par ASTRÉE n'étant généralement pas portables mais liés à une architecture particulière, la preuve d'absence d'erreur à l'exécution doit tenir compte de ces hypothèses sémantiques. Par conséquent, ASTRÉE permet à l'utilisateur de sélectionner une architecture cible souhaitée ; l'analyse sera alors correcte vis-à-vis de cette architecture, et seulement de celle-ci.

La définition précise des alarmes que l'analyseur est susceptible d'émettre lorsqu'il échoue à prouver la sûreté d'une opération donnée découle de cette sémantique et correspond aux cas d'erreurs donnés dans la section 2.2.2. Lorsqu'une alarme a été

émise, il est également nécessaire de définir le comportement de l'analyseur après le point où elle a été signalée. Lorsqu'il est possible de poursuivre l'analyse avec une sur-approximation de l'ensemble des états concrets (par exemple, après un dépassement dans un calcul entier, nous pouvons supposer, de manière conservative, que le calcul se poursuit avec n'importe quelle valeur entière pour résultat), l'analyse continue et peut ainsi signaler d'autres alarmes survenant dans la même exécution. Dans certains cas, comme par exemple lors d'une alarme relative à l'écriture en dehors de l'espace alloué à une variable, il n'est pas possible de définir une sur-approximation précise pour l'ensemble des comportements autorisés par la sémantique du langage (en pratique, toute partie de la mémoire est susceptible d'être modifiée, mais le programme peut aussi s'arrêter par une erreur de segmentation, soit immédiatement, soit longtemps après l'opération invalide) ; le choix qui a été fait lors de la conception d'ASTRÉE est de "couper" les exécutions produisant de tels comportements, permettant ainsi une analyse précise de la suite du code en ne considérant que les exécutions qui n'ont pas produit cette alarme.

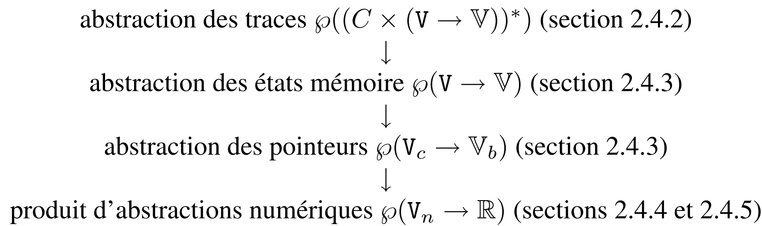
2.4.1.2. *Une hiérarchie de domaines abstraits paramétrables*

Une fois qu'une sémantique a été choisie, il est nécessaire de choisir une abstraction. Afin de rendre l'analyse suffisamment puissante pour analyser avec précision des familles d'applications assez générales, le domaine abstrait utilisé est en fait composé d'un ensemble de domaines qu'il est possible d'activer ou désactiver individuellement, et qui possèdent des paramètres servant à guider l'analyse ou les fonctions de transfert.

L'abstraction implantée dans ASTRÉE collecte des invariants pour chaque point de contrôle et chaque contexte d'appel (autrement dit, elle est à la fois "*flow sensitive*" et "*context sensitive*"). En effet, du point de vue de la sémantique collectrice, les états peuvent être décomposés en deux parties, l'une relative à l'*état du contrôle* et l'autre à l'*état des données* ; autrement dit, $S \triangleq C \times D$, où C précise le point de contrôle syntaxique situé juste avant la prochaine instruction à exécuter ainsi que le contexte d'appel et D désigne l'état de la mémoire du programme, qui comprend les variables locales et globales. Nous construisons une abstraction dépendante du contexte d'appel et du point de contrôle en procédant à un partitionement (section 2.3.6.2) où chaque élément de la partition correspond à un contexte d'appel et un point de contrôle. Ce partitionement rend l'analyse plus simple à implanter et plus précise puisque nous ne fusionnons pas d'invariants correspondant à des points de contrôle différents. Toutefois, cela limite l'analyse à des programmes qui n'ont qu'un nombre fini de points de contrôle et de contextes d'appels, excluant ainsi les programmes récursifs, ce qui n'est pas gênant dans le cadre des applications critiques embarquées.

ASTRÉE est construit de manière modulaire, ce qui permet de construire un domaine abstrait complexe à partir de plusieurs domaines plus simples. La sémantique collectrice considérée définit des traces d'exécutions qui sont abstraites par un foncteur transformant un domaine abstrait représentant des états mémoire (définis comme

des fonctions des variables vers les valeurs) en une abstraction pour des suites d'états. Ce domaine de partitionement de trace prend en compte les aspects spécifiques aux traces d'exécution, et délègue les opérations sur les états mémoire au domaine mémoire sous-jacent. Celui-ci est à son tour paramétré par une abstraction des pointeurs, qui est lui même paramétré par un domaine abstrait numérique, constitué d'un produit (section 2.3.6.1) de domaines numériques. On obtient ainsi la structure ci-dessous :



Cette structure modulaire a l'avantage de rendre la conception, l'implantation et la maintenance du code de l'analyseur plus simples.

2.4.1.3. *Itérateur abstrait*

Une abstraction ayant été définie, il ne reste plus qu'à implanter un itérateur abstrait, qui a pour but de diriger le calcul de la sémantique abstraite (section 2.3.3).

La sémantique du programme s'exprime sous la forme du plus petit point fixe $X = \mathrm{lfp}^{\subseteq} F$ de la fonction F, qui est définie par l'arbre syntaxique du programme à analyser. La sémantique abstraite s'exprime donc sous une forme similaire $X^\sharp = \mathrm{lfp}^{\subseteq^\sharp} F^\sharp$, où $F^\sharp$ consiste en une suite d'appels à des fonctions de transfert du domaine abstrait de l'analyseur. Une telle équation peut être résolue par un calcul itératif avec élargissement. Toutefois, en raison du partitionement d'états (section 2.4.1.2), cela ne donne pas un algorithme efficace : $X^\sharp$ a une composante par point de contrôle et contexte d'appel et ne peut donc être conservé complètement en mémoire à tout instant de l'analyse ; de plus, recalculer les invariants en chaque point de programme, à chaque pas d'analyse entraînerait beaucoup de calculs inutiles.

ASTRÉE ne suit pas une telle approche, mais procède à la manière d'un interpréteur, en suivant le flot d'exécution du programme analysé et calculant, à chaque étape, un nouvel invariant local représentant un ensemble de traces d'exécution partielles terminant à ce point (tandis qu'un interpréteur calculerait simplement un nouvel état mémoire). Après chaque pas de l'analyse, l'invariant correspondant au point précédent n'a plus besoin d'être conservé en mémoire. Lorsque plusieurs branches peuvent être prises, comme dans le cas d'une instruction conditionnelle `if then else`, chaque branche doit être explorée par l'analyseur, puis les invariants abstraits obtenus à la

sortie de chaque branche doivent être fusionnés à l'aide d'un opérateur d'union abstraite. Les calculs relatifs à chacune de ces branches sont indépendants et peuvent être effectués en parallèle [COU 77a, MON 05] si l'analyse est exécutée sur une architecture multi-processeurs ou sur une grappe de calcul.

Le cas des boucles nécessite toutefois de conserver en mémoire l'itéré précédent lors du calcul de l'invariant de boucle par un plus petit point fixe (seul l'itéré utilisé par l'élargissement ∇, en tête de boucle, a besoin d'être retenu). Lorsque le programme analysé contient des boucles imbriquées, nous calculons un invariant pour les boucles intérieures à chaque itération de la boucle extérieure. Il est possible d'améliorer la précision d'un invariant de boucle en déroulant celle-ci un certain nombre fini de fois.

Ainsi, cette stratégie d'itération permet de ne conserver des invariants locaux qu'en un nombre de points très faible pour la plupart des programmes, proportionnel au degré d'imbrication des structures de contrôle (boucles et instructions conditionnelles).

2.4.2. *Abstraction des exécutions*

La méthode de Floyd [FLO 67] est complète, ce qui implique que toute propriété d'invariance peut être prouvée en ne considérant que des invariants représentant des ensembles d'états. Néanmoins, cela ne signifie pas que cette méthode est optimale, ni qu'elle conduit aux invariants abstraits les plus simples. En fait, dans certains cas, il est plus efficace d'utiliser des invariants de traces d'exécution (suivant la méthode de Burstall [BUR 74]) plutôt que des invariants d'états.

En particulier, de nombreux domaines abstraits numériques ne permettent de représenter exactement que des ensembles *convexes* de valeurs concrètes (c'est le cas des intervalles, des octogones et des polyèdres convexes). Par conséquent, l'opération d'union abstraite induit une perte de précision qui peut se révéler importante. Par exemple, si la variable `x` peut prendre n'importe quelle valeur entière sauf 0, et s'il est nécessaire que l'analyse établisse ce fait pour prouver l'absence d'erreurs à l'exécution (si il faut montrer, par exemple, qu'une division par `x` ne causera pas d'erreur), alors il faut impérativement utiliser une abstraction non convexe. Pour cela, il est utile de distinguer les états pour lesquels $\mathtt{x} < 0$ de ceux pour lesquels $\mathtt{x} > 0$. Cela revient à effectuer un partitionement de l'ensemble des états du programme ou, de manière équivalente, à faire intervenir dans l'analyse des disjonctions de propriétés élémentaires. Notons que ce mécanisme de partitionement diffère du partitionement total des données vis à vis du contrôle, décrit en section 2.4.1.2.

La principale difficulté consiste à calculer automatiquement de "bonnes" partitions, c'est-à-dire des partitions pertinentes mais de taille raisonnable (afin d'éviter une explosion combinatoire). Dans de nombreux cas, ce choix peut être guidé par le

flot de contrôle du programme analysé. Par exemple, il est souvent pertinent de distinguer les exécutions qui ont traversé la branche “vrai” de celles qui ont traversé la branche “faux” après une instruction conditionnelle, ou encore de distinguer les exécutions à la sortie d'une boucle selon le nombre d'itérations de la boucle effectuées avant la sortie. Cette intuition se formalise en observant que cela revient à partitionner les états accessibles à chaque point de contrôle en utilisant des partitions dérivées de l'histoire des traces d'exécution. C'est la raison pour laquelle ASTRÉE sur-approxime non pas des états mais des traces d'exécution. Un élément du domaine abstrait de partitionement de traces [MAU 05, RIV 07] associe à chaque élément de la partition des exécutions une valeur abstraite du domaine sous-jacent.

Considérons une fonction d'interpolation calculant une fonction linéaire par morceaux, en localisant tout d'abord le domaine de la valeur d'entrée à l'aide d'une boucle, puis appliquant la formule correspondante. Cette fonction est décrite par le code suivant :

```
const double tx[5] = { -1.0, -1.0, 1.0, 2.0 };
const double tc[4] = { 0.0, -1.0, 8.0, 0.0 };
const double ty[4] = { -2.0, -2.0, -4.0, 4.0 };
double x, r;
int i = 0;
while ((i < 3) && (x >= tx[i+1]))
        i = i + 1;
r = (x - tx[i]) * tc[i] + ty[i];
```

Le code ci-dessus calcule une approximation (en nombres flottants) de la fonction mathématique f définie ci-dessous :

$$f : x \mapsto \begin{cases} -1 & \text{si } x < -1 \\ -3 - x & \text{si } x \in [-1, 1) \\ -12 + 8x & \text{si } x \in [1, 2) \\ 4 & \text{si } x \geq 2 \end{cases}$$

En supposant que la variable d'entrée est un nombre flottant quelconque, et que la fonction est analysée sans effectuer de partitionement, nous n'obtenons aucune relation entre `i` et le domaine de variation de `x` à la sortie de la boucle. Ce dernier est donc l'intervalle de tous les nombres flottants à la sortie, alors que la sortie de la fonction est effectivement bornée par l'intervalle $[-4, 4]$ (aux erreurs d'arrondi près). À l'inverse, un partitionement permet de relier à la sortie de la boucle chaque valeur de `i` à un intervalle précis ; par exemple, si nous effectuons deux itérations de la boucle, alors `i` vaut 2, et `x` est dans l'intervalle $[1, 2)$. La fusion des intervalles après un calcul par cas sur chaque partition donne bien l'intervalle attendu $[-4, 4]$ (aux erreurs d'arrondi près).

En pratique, ce domaine abstrait est décrit par un foncteur qui transforme une abstraction pour des ensembles d'états mémoire (i.e., où une valeur abstraite sur-approxime des éléments de $\wp(\mathtt{V} \rightarrow \mathbb{V})$) en une abstraction pour des ensembles d'exécutions $\wp((C \times (\mathtt{V} \rightarrow \mathbb{V}))^*)$. Les opérations abstraites de ce foncteur peuvent être classées en trois catégories :

– la *création de partitions*, en découpant des partitions existantes, e.g., à l'entrée d'une instruction conditionnelle ou d'une boucle ;

– la *fusion de partitions* (lorsque certaines ne sont plus nécessaires) ;

– les *opérations du domaine sous-jacent*, qui sont appliquées directement sur chaque partition, e.g., pour l'analyse d'une affectation.

La création et la fusion de partitions sont généralement guidées par des stratégies, qui déterminent quand un partitionement peut améliorer la précision, comme par exemple lorsque plusieurs conditions successives testent une même variable. Toutefois, ce partitionement est *dynamique*, c'est-à-dire que la décision de créer une partition ou pas est faite lors de l'analyse et non avant. Ces stratégies assurent à la fois précision et efficacité.

2.4.3. *Abstractions des structures de données et des pointeurs*

À un point de contrôle donné, un état mémoire du programme $s \in D$ associe une valeur $s(\mathtt{x})$ à chaque variable $\mathtt{x} \in \mathtt{V}$. Notre but est d'abstraire les ensemble d'états $\wp(D)$.

Une première difficulté est que l'ensemble $\mathtt{V}$ des variables varie au cours de l'exécution. Pour la résoudre, il suffit de noter que l'ensemble des variables locales est entièrement défini par le point de programme courant et le contexte d'appel (i.e., la pile des appels de fonction depuis `main` jusqu'à la fonction courante). De plus, nous interdisons les allocations dynamiques de variables. Par partitionement vis à vis du point de programme et du contexte d'appel (analyse *flow sensitive* et *full context sensitive*), il n'est plus alors question que d'abstraire des ensembles d'états mémoires portant sur le même ensemble de variables.

Une deuxième difficulté provient du système de types particulièrement riche du langage C, contenant types de *base* (entiers et flottants machine, de taille variable), types *agrégés* (structures et tableaux, potentiellement imbriqués), types *unions* et types *pointeurs*. ASTRÉE comporte un domaine abstrait de structures de données dont le rôle est de traduire le problème de l'abstraction des variables de type complexe en un problème d'abstraction de variables de type de base, qui pourra alors être directement traité par un ou plusieurs domaines numériques (sections 2.4.4–2.4.5). Ignorant pour l'instant le problème des types unions, il est clair qu'une variable agrégée peut être décomposée statiquement en une collection plate $\mathtt{V}_c$ d'éléments de type de base,

que nous appellerons *cellules*. Par exemple la variable `struct { int a; char b } v[2]` sera traduite en quatre cellules : $\mathtt{V}_c = \{c_1, c_2, c_3, c_4\}$ correspondant respectivement à `v[0].a`, `v[0].b`, `v[1].a` et `v[1].b`. Ignorant également pour l'instant le problème des pointeurs, toute expression sur les variables C peut être traduite en une expression sur les cellules, acceptable par un domaine numérique. Cette transformation est gérée par le domaine de structures et doit se faire dynamiquement, en interaction avec les domaines numériques, puisque les indices de tableaux doivent être évalués pour connaître quelles cellules sont référencées par un accès de tableau donné. Il est également possible de replier les tableaux, c'est-à-dire, d'abstraire plusieurs cellules par une seule cellule abstraite. Ainsi, `v` peut être abstraite par $\mathtt{V}_c = \{c'_1, c'_2\}$, où c'_1 représente l'union des valeurs de `v[0].a` et `v[1].a`, de même que c'_2 pour `v[0].b` et `v[1].b`. Ceci permet une analyse plus efficace mais moins précise dans le cas de grands tableaux. ASTRÉE replie dynamiquement automatiquement les tableaux quand il détecte qu'une expression accède simultanément à un grand nombre d'éléments (par exemple dans la boucle `for (i=0; i<100; i++) tab[i] = 0` où, après élargissement, il est nécessaire de calculer l'effet de `tab[i] = 0` pour `i` $\in [0, 99]$).

Nous considérons maintenant le cas des types unions, qui permettent de stocker des champs de types différents dans un même bloc mémoire. Bien que cela ne soit pas supporté officiellement par la norme C, de nombreux programmes se permettent d'écrire une valeur dans un champ de l'union, puis de lire l'union en utilisant un champ différent. Ceci a pour effet de réinterpréter la représentation binaire de la valeur écrite comme la représentation binaire d'une valeur d'un autre type (méthode appelée *type punning*). Par exemple, le programme `union { int i; float f; } x; x.f = 2.5; x.i ^= 0x80000000` a pour effet d'inverser le signe du flottant `x.f` (en supposant une architecture 32 bits). Le même effet peut d'ailleurs être obtenu par un accès par pointeur après transtypage : `float f = 2.5; *((int*)&f) ^= 0x80000000`. Comme pour les types agrégés, le domaine de structure décomposera les types unions en un ensemble de cellules de type de base. Toutefois, celles-ci ne sont plus indépendantes : une écriture dans une cellule (par exemple `x.i`) peut modifier d'autres cellules (par exemple `x.f`). Le domaine de structure se charge de maintenir la cohérence entre les cellules en introduisant des affectations supplémentaires, ce qui permet aux domaines numériques de les traiter comme des variables indépendantes. ASTRÉE est capable d'exploiter, par réductions partielles entres cellules, quelques hypothèses sur la représentation binaire des types (par exemple, la position du bit de signe d'un flottant) tandis que, dans les autres cas, modifier une cellule supprimera simplement toute information sur les cellules occupant la même zone mémoire, ce qui est toujours sûr. Cette méthode est décrite en détails dans [MIN 06a].

Nous considérons maintenant le cas des types pointeurs. Ceux-ci sont modélisés dans la sémantique concrète comme des entiers symboliques, composés d'un nom de variable (la base) et d'un déplacement numérique positif (l'offset). Un ensemble de pointeurs est alors abstrait par abstraction séparée de ces deux composants : un ensemble de bases représenté en extension et une cellule entière pour représenter l'offset.

Un domaine de pointeurs se charge de maintenir l'information de base, et de traduire les opérations sur les pointeurs comme des opérations arithmétiques sur les cellules offset qui sont alors évaluées par les domaines numériques. Un avantage de cette méthode est de permettre l'inférence de relations entre pointeurs, ou entre pointeurs et entiers, si le domaine numérique le permet. Par exemple, dans `p=&t[10]; for (i=0; i<100; i++,p++)`, nous avons l'invariant de boucle `p=&t[i+10]`. Comme pour les accès de tableaux, les déréférencements sont résolus dynamiquement par le domaine de pointeurs afin de déterminer les cellules accédées.

2.4.4. *Abstractions numériques classiques*

Après l'abstraction des traces d'exécution et celle des structures de données, il nous faut encore abstraire les valeurs des cellules. Formellement, ces ensembles seront de la forme $\wp(\mathtt{V}_n \to \mathbb{R})$, où $\mathtt{V}_n$ est un ensemble fini de cellules contenant des valeurs numériques. Les programmes peuvent manipuler différents types de nombres : entiers, signés ou non, flottants, `char`, etc. Notre sémantique concrète les représente tous comme des réels (ce qui exclus les valeurs spéciales de flottants qui seront donc traitées à part). Comme annoncé plus haut, les ensembles de valeurs de cellules seront abstraits par une combinaison de plusieurs domaines abstraits. Dans cette section, nous décrivons des domaines abstraits généraux qui n'ont pas été développés spécifiquement pour les applications visées par ASTRÉE.

2.4.4.1. *Intervalles*

Le domaine abstrait des intervalles [COU 77b] ne garde que les bornes supérieures et inférieures de chaque cellule numérique. C'est un des domaines les plus simples, et pourtant cette information est primordiale pour prouver un grand nombre de propriétés de sûreté, comme l'absence de dépassements, la garantie que les accès de tableaux sont bien dans les bornes ou encore la validité des opérations de décalages de bits. La plupart des opérations abstraites sur les intervalles se fondent sur l'arithmétique d'intervalles[1]. La correction des opérations abstraites sur les flottants (en considérant tous les modes d'arrondi possibles) est assurée par un arrondi inférieur des bornes inférieures et un arrondi supérieur des bornes supérieures, et ce à la même précision que les opérations concrètes.

Les opérateurs d'élargissement, utilisés pour accélérer la convergence des boucles, sont très importants en interprétation abstraite. Dans ASTRÉE, l'élargissement rapide

---

1. Noter que l'arithmétique des intervalles fût conçue comme une méthode de calcul pour l'exécution des programmes dans les calculateurs pour approcher un nombre réel [MOO 66] alors qu'elle est utilisée ici pour l'analyse statique pour approcher des ensembles de nombres entiers ou réels, ce qui nécessite des opérations supplémentaires comme l'inclusion, l'union, l'élargissement, etc.

de la section 2.3.4 n'est pas assez précis. Il a été raffiné par l'introduction de paliers : les bornes instables sont d'abord extrapolées selon une suite finie de bornes de plus en plus grandes avant de passer à l'infini. Pour la plupart des calculs flottants stables (comme `while (1) { x = x *` α `+` $[0, \beta]$`; }`, qui est stable dans l'intervalle $\mathtt{x} \in [0, \beta/(1-\alpha)]$ quand $\alpha \in [0, 1)$), une simple rampe exponentielle suffit (`x` sera alors borné par le premier palier supérieur à $\beta/(1-\alpha)$). Certains de ces paliers peuvent aussi être inférés à partir des constantes apparaissant dans le programme, comme par exemple les constantes utilisées pour définir les tailles de tableaux. Certains domaines abstraits peuvent de plus inférer dynamiquement de nouveaux paliers durant l'analyse.

Un gros avantage du domaine des intervalles est son faible coût : $\mathcal{O}(|\mathtt{V}_n|)$ en mémoire et en temps par opération abstraite. De plus, le temps dans le cas le pire peut être réduit de manière significative en un temps logarithmique, en exploitant le fait que les opérations binaires (comme l'union) sont appliquées la plupart du temps à des arguments très similaires, qui ne diffèrent que par la valeur de quelques variables. ASTRÉE utilise une structure fonctionnelle avec partage pour tirer parti de cette propriété. Ceci permet au domaine des intervalles de gérer efficacement des dizaines de milliers de cellules.

2.4.4.2. *Abstraction des calculs flottants*

La plupart des logiciels de contrôle / commande sont conçus avec une sémantique utilisant des nombres réels (dans $\mathbb{R}$) parfaits, mais sont en fait implémentés en arithmétique flottante. Les arrondis introduits par cette arithmétique induisent une divergence entre les calculs utilisés durant la conception des algorithmes et leur implantation. Ces arrondis peuvent s'accumuler et causer des dépassements ou des divisions par zéro, même lorsque c'est impossible dans l'algorithme sur les réels. ASTRÉE analyse les calculs flottants par une approximation correcte de la sémantique IEEE 754–1985[IEE 85], ce qui permet une détection systématique de ce type de problèmes.

Les raisonnements sur les flottants sont généralement difficiles, car la plupart des propriétés mathématiques des réels ne sont pas vraies sur les flottants : par exemple l'associativité et la distributivité de l'addition et de la multiplication ne sont pas préservées. La plupart des domaines numériques abstraits dans ASTRÉE sont fondés sur des manipulations d'expressions (comme l'algèbre linéaire) qui ne seraient pas correctes en remplaçant les opérations réelles par des opération flottantes. ASTRÉE utilise donc un domaine abstrait spécifique [MIN 04a, MIN 04b], capable d'abstraire correctement les expressions : ainsi, une fonction $f : X \to Y$ est abstraite par une fonction non-déterministe $g : X \to \wp(Y)$ telle que $f(x) \in g(x)$ au moins pour tous les x dans un sous-ensemble atteignable R de X. La fonction g peut donc être utilisée à la place de f pour tous les arguments de R. En pratique, le domaine utilise des expressions linéaires à coefficients dans des intervalles, de la forme $g(\vec{\mathtt{V}}) = [\alpha, \beta] + \sum_i [\alpha_i, \beta_i] \times \mathtt{V}_i$, et R correspond à l'intervalle de variation des variables $\mathtt{V}_i$ tel qu'inféré par le domaine des intervalles. Dans g, contrairement à f, $+$ et $\times$ font référence aux opérations sur

les réels, ainsi g peut être directement utilisée par les domaines abstraits numériques raisonnant sur des réels.

Il est facile de manipuler les expressions linéaires à intervalles, car elles forment un espace affine, et les intervalles sont suffisamment expressifs pour abstraire tout ce qui n'est pas linéaire, comme les arrondis, par un effet non-déterministe. Considérons par exemple l'expression `z = x + 2.f * y`, évaluée dans les flottants simple précision. La linéarisation donnera : $[1.9999995, 2.0000005]\texttt{y} + [0.99999988, 1.0000001]\texttt{x} + [-1.1754944\times10^{-38}, 1.1754944\times10^{-38}]$. Si nous savons de plus que $\texttt{x}, \texttt{y} \in [-100, 100]$, alors nous pouvons alternativement linéariser en $2\texttt{y}+\texttt{x}+[-5.9604648\times10^{-5}, 5.9604648\times10^{-5}]$, ce qui est plus simple mais contient des constantes (correspondant à l'erreur d'arrondi) plus grandes, et est donc moins précis.

En plus des problèmes soulevés par les arrondis, les flottants peuvent aussi prendre des valeurs non numériques : $+\infty$, $-\infty$ et *NaN*. De plus, ils distinguent les valeurs $+0$ et -0. Ces valeurs particulières sont abstraites pour chaque cellule par un booléen indiquant leur présence possible, et $+0$ et -0 sont abstraits par un unique 0.

Comme nous l'avons vu plus haut, l'arithmétique d'intervalles est très facile à approcher en flottant. Cela permet une implantation efficace du domaine des expressions linéaires à intervalles. D'autres domaines plus précis sur les flottants existent [GOU 01], mais ils ne sont pas utilisés dans ASTRÉE car ils sont plus coûteux et la précision qu'ils apportent ne semble pas utile quand seule l'absence d'erreurs à l'exécution nous importe.

2.4.4.3. *Domaine des octogones*

Le domaine des intervalles ne permet pas de connaître les relations entre les valeurs des variables. Le domaine des octogones [MIN 04b, MIN 06b] permet de représenter certaines relations simples entre des couples de variables.

Formellement, si on appelle $Oct(\texttt{V}_n)$ le sous-ensemble des expressions linéaires $Oct(\texttt{V}_n) \triangleq \{\, \pm\texttt{x} \pm \texttt{y} \,|\, \texttt{x}, \texttt{y} \in \texttt{V}_n \,\}$, alors le domaine des octogones abstrait un ensemble de points concrets $X \in \wp(\texttt{V}_n \to \mathbb{R})$ en $\alpha_{Oct}(X) \triangleq \{\, \max_{x\in X} e(x) \,|\, e \in Oct(\texttt{V}_n \,\}$. Cela correspond à l'ensemble de contraintes le plus strict de la forme $\pm\texttt{x} \pm \texttt{y} \leq c$ satisfaites par X. Le nom d'*octogones* vient de la forme de ces ensembles en dimension 2.

Après une commande de la forme `if (x>y) x=y`, le domaine des octogone peut inférer $\texttt{x} \leq \texttt{y}$. Si, plus tard, l'intervalle de variations de `y` est réduit, par exemple à cause d'un test `if (y<=10)`, il peut utiliser cette information pour en déduire que $\texttt{x} \leq 10$. Bien que $\texttt{x} \leq 10$ soit une simple propriété d'intervalles, le domaine des intervalles est incapable de suivre les étapes de raisonnement nécessaires pour l'inférer.

Le domaine des octogones est aussi très utile dans les boucles. Prenons par exemple la boucle `for (i=0,x=0; i<1000; i++) { if (?) x++; if (?) x=0; }`. Dans ce cas, le domaine des octogones trouve la relation $\texttt{x} \leq \texttt{i}$ en tête de boucle, bien que `x` et `i` ne se trouvent jamais affectés l'un à l'autre ni testés ensemble. Comme $\texttt{i} = 1000$ à la sortie de la boucle, le domaine en déduit que $\texttt{x} \in [0, 1000]$.

Le domaine des octogones utilise une structure de données matricielle [LAR 97], avec un coût en mémoire en $\mathcal{O}(|\texttt{V}_n|^2)$, et l'algorithme de plus court chemins utilisé pour la clôture des contraintes est de complexité cubique. Comme le domaine des intervalles, le domaine des octogones peut être implanté efficacement en utilisant des flottants. De plus, il peut abstraire les entiers et l'arithmétique flottante (en utilisant la linéarisation de la section 2.4.4.2), et même inférer des relations entre cellules de type différent.

2.4.4.4. *Domaine des arbres de décision*

Le domaine des arbres de décision est spécialisé dans l'abstraction des booléens, et en particulier des relations parfois complexes entre des ensembles de variables booléennes et des variables numériques. Ce domaine est nécessaire dans ASTRÉE car une partie de l'information de flot de contrôle est en réalité maintenu au sein de variables booléennes.

Considérons par exemple le programme `b = (x >= 6); ... if (b) y = 10 / (x - 4);`. Ce programme n'effectue jamais de division par zéro car les divisions n'ont lieu que quand `b` est vrai, et donc quand `x` est plus grand que 6. La preuve de cette absence d'erreur ne peut être faite que si la relation entre `b` et `x` est connue au moment du test. Les relations linéaires ne sont pas assez expressives ici, il nous faut tenir compte du fait que `b` ne peut avoir que deux valeurs possibles, 0 ou 1. Le domaine des arbres de décision traite donc de façon différente certaines variables reconnues comme booléennes, et d'autres comme purement numériques.

Les valeurs abstraites de ce domaine sont des arbres de décision. Les feuilles de ces arbres sont des valeurs abstraites représentant des ensembles de valeurs numériques, et les nœuds internes sont étiquetés par des variables booléennes. Les deux sous-arbres partant d'un nœud interne étiqueté par la variable `b` correspondent à un ensemble de valeurs pour `b` faux ou vrai. La nature des valeurs abstraites apparaissant aux feuilles est un paramètre du domaine. Par défaut dans ASTRÉE, c'est le produit d'un petit nombre de domaines de base peu coûteux, comme les intervalles ou le domaine des égalités entre variables. La concrétisation d'un arbre est l'ensemble des valuations de cellules telles que, en suivant les décisions dans l'arbre correspondant aux valeurs des variables booléennes, nous aboutissons à une feuille et la valuation des cellules numérique est dans la concrétisation de cette feuille. Dans l'exemple ci-dessus, l'arbre de décision correspondra à “si `b` est vrai, alors `x` est supérieur à 6, et sinon `x` est inférieur à 6”.

Ces arbres ont des propriétés communes avec les diagrames de décision binaire de [BRY 86] : un partage des sous-arbres communs est possible, et les opérations binaires sont de l'ordre du produit de la taille des arbres.

Ce domaine abstrait définit un partitionement au sens de la section 2.3.6.2. Le partitionement en question s'opère sur les états et non sur les exécutions, ce qui signifie qu'il n'est pas affecté directement par le choix d'une branche lors d'un test, mais qu'une affectation peut modifier les partitions.

2.4.4.5. *Regroupement des variables*

Bien que le domaine des octogones, avec son coût cubique, soit très efficace comparé à d'autres domaines abstraits (comme celui des polyèdres [COU 78b]), il reste encore trop coûteux pour des programmes avec des dizaines de milliers de variables. Le domaine des arbres de décision est encore plus coûteux, et ne peut pas non plus s'appliquer sur toutes les variables des programmes de taille industrielle. ASTRÉE renonce donc à relier toutes les variables entre elles. Au contraire, l'analyseur regroupe les variables en petits paquets (non nécessairement disjoints) et ne maintient que des relations entres variables d'un même paquet. Cette solution fonctionne bien en pratique car, au niveau de la conception, on ne raisonne en général que sur des relations entre un nombre assez restreint de variables.

Pour les octogones, une simple analyse syntaxique regroupe les cellules apparaissant près les unes des autres dans le code (celles utilisées ensemble dans des expressions, les indices de boucles, etc.). Cette analyse extrêmement rapide est effectuée avant l'itération des domaines abstraits. Pour les arbres de décision, cela ne suffit pas car les affectations dans des booléens peuvent être très éloignées de leur utilisation. Un critère plus sémantique est utilisé : les dépendances sont calculées en fonction d'expressions sur lesquelles il a été identifié que les arbres de décision pouvaient ajouter de la précision.

Durant l'analyse avec les domaines abstraits, on associe un octogone ou un arbre de décision par paquet, mais aucune relation n'est directement gardée entre des variables qui ne seraient pas dans un même paquet. Le coût total devient donc linéaire en fonction de la taille du code (et cubique en fonction de la taille des paquets, qui ne dépend pas de la taille du code).

2.4.5. *Abstractions numériques spécifiques au domaine d'application*

2.4.5.1. *Filtrage numérique*

Un logiciel embarqué échange généralement des valeurs avec un environnement physique extérieur. Les filtres numériques sont des petits algorithmes qui permettent de lisser les flux de données ainsi reçus par les capteurs. Pour ce faire, les algorithmes

utilisent des équations différentielles, qui sont discrétisées et, bien souvent, linéarisées. Or, il est pratiquement impossible de borner la valeur des variables qui sont utilisées dans ces algorithmes de filtrage numérique sans utiliser des domaines abstraits numériques appropriés.

Aussi, dans l'analyseur ASTRÉE, nous avons implanté un domaine abstrait spécifique [FER 04] qui permet de traiter les algorithmes de filtrage linéaire. Ce domaine abstrait utilise à la fois des inégalités entre formes quadratiques (qui représentent le contour d'ellipses) et des développements formels. Un filtre linéaire simplifié consiste en une récurrence linéaire, dans laquelle la valeur d'une variable rémanente de sortie `s` est, à chaque itération de la boucle, définie comme une combinaison linéaire de la valeur courante d'une variable d'entrée `e` et des k dernière valeurs (où k est un nombre fixé) qui ont été prises par la variable `s` au cours des k dernières itérations de la boucle. Les cas les plus importants sont ceux des filtres du premier ordre, pour lesquels k vaut 1, et ceux du second d'ordre, pour lesquels k vaut 2. Aussi, les filtres simplifiés du premier ordre s'analysent de manière précise en utilisant le domaine des intervalles muni d'un élargissement par palier (voir la section 2.4.4.1), alors que les filtres simplifiés du second ordre s'analysent à l'aide d'inégalités qui bornent la valeur prise par des formes quadratiques. Les coefficients de la forme quadratique sont extraits directement du code à analyser, en l'occurrence, la forme quadratique adaptée à l'analyse d'une récurrence de la forme $s_{n+2} = \alpha_1 \cdot s_{n+1} + \alpha_2 \cdot s_n + e_{n+2}$ est la suivante : ${s_{n+2}}^2 - \alpha_1 \cdot s_{n+2} \cdot s_{n+1} - \alpha_2 \cdot {s_{n+1}}^2$.

On s'intéresse maintenant aux filtres d'ordre supérieur. Quel que soit son ordre, il est possible de transformer un filtre simplifié, en une combinaison linéaire de filtres simplifiés du premier et du second ordre. Pour cela, nous procédons de la manière suivante. En notant respectivement s_n et e_n la valeur des variables `s` et `e` au cours de la n-ième itération de la boucle, nous savons que $s_{n+k} = e_{n+k} + \sum_{1\leq j\leq k} \alpha_j \cdot s_{n+k-j}$. En factorisant le polynôme caractéristique de la récurrence $X^k - \sum_{1\leq j\leq k} \alpha_j \cdot X^{k-j}$, nous pouvons déduire un changement de variables qui permet d'exprimer la suite $(s_n)_{n\in\mathbb{N}}$ des valeurs de sortie comme une combinaison linéaire des valeurs de sortie de filtres numériques simplifiés du premier (pour les racines réelles) et du second ordre (pour les racines complexes) [FER 05b].

En pratique, les algorithmes de filtrage utilisent à chaque itération plusieurs valeurs consécutives du flux d'entrée. Appelons l le nombre de valeurs consécutives qui sont lues. Le filtre implante alors une récurrence de la forme $s_{n+k} = \sum_{0\leq j<l} \beta_j \cdot e_{n+k-j} + \sum_{1\leq j\leq k} \alpha_j \cdot s_{n+k-j}$ (au lieu de $s_{n+k} = e_{n+k} + \sum_{1\leq j\leq k} \alpha_j \cdot s_{n+k-j}$ comme c'était le cas pour les filtres simplifiés). Nous pourrions abstraire la contribution totale du flux d'entrée à chaque itération comme une valeur globale, mais ceci rendrait l'analyse très imprécise. En effet, les contributions d'une même valeur d'entrée à des itérations successives ont tendance à s'annuler entre elles (en particulier lorsque les coefficients β_j n'ont pas tous le même signe).

Pour obtenir une analyse plus précise, nous isolons la contribution des N dernières entrées, où N est un paramètre. Ainsi, nous exprimons, pour $n+k > N$, la valeur s_{n+k} comme la somme $s'_{n+k} + \sum_{0 \leq j < N} \delta_j^N \cdot e_{n+k-j}$. Dans cette somme, la combinaison linéaire $\sum_{0 \leq j < N} \delta_j^N \cdot e_{n+k-j}$ décrit la contribution exacte des N dernières entrées, alors que la nouvelle variable s'_{n+k} représente la sortie d'un filtre fictif, qui satisferait la récurrence suivante $s'_{n+k} = \sum_{0 \leq j < l} \beta_j \cdot \varepsilon_j^N \cdot e_{n+k-j} + \sum_{1 \leq j \leq k} \alpha_j \cdot s'_{n+k-j}$, les coefficients (δ_j^N) et (ε_j^N) étant calculés automatiquement. Si les calculs étaient effectués en arithmétique réelle, les coefficients (ε_j^N) tendraient vers 0 quand N tend vers l'infini. De ce fait, plus N serait grand, plus l'abstraction serait précise (nous pourrions même converger vers une analyse exacte du filtre). Cependant, en pratique, ces paramètres sont encadrés par un intervalle en arithmétique flottante contenant le résultat exact. Ainsi, l'analyse est correcte malgré les erreurs d'arrondi. Par contre, si la valeur du paramètre N est trop grande, la précision se détériore du fait des erreurs d'arrondi dès que le signe des coefficients n'est plus connu. En pratique, le meilleur choix pour le paramètre N est calculé automatiquement pour chaque filtre du programme. Il reste alors à encadrer la valeur de sortie s'_n du filtre fictif, ce qui ce fait en considérant la contribution globale des valeurs d'entrée comme une seule entrée globale. La perte de précision due à cette abstraction est négligeable, car elle est amortie par les coefficients (ε_j^N) dont la valeur est très petite.

Notons, qu'il est impossible en pratique d'analyser des algorithmes de filtrage numérique par raffinement automatique (tel que [CLA 00]). Même dans le cas d'un filtre simplifié (où la récurrence est de la forme $s_{n+k} = e_{n+k} + \sum_{1 \leq j \leq k} \alpha_j \cdot s_{n+k-j}$), il est très difficile de borner la valeur du flux de sortie (s_n) sans utiliser de formes quadratiques. De plus, si il est imaginable d'encadrer une ellipse en utilisant des inégalités linéaires (en utilisant un grand nombre de tangentes), trouver les bons demi-plans est au moins aussi difficile que découvrir la bonne ellipse. En pratique, cela peut être fait avec le domaine des polyèdres, en retardant autant que possible l'utilisation de l'opérateur d'élargissement, mais cette méthode serait très coûteuse. La situation est encore plus délicate dans le cas, plus général, des filtres non simplifiés, car il faut tenir compte de l'effet annulateur des contributions successives d'une même valeur d'entrée.

2.4.5.2. *Lentes dérives exponentielles*

Les logiciels critiques embarqués font souvent des calculs qui seraient numériquement stables s'ils étaient effectués en arithmétique réelle, mais qui divergent lentement en arithmétique flottante du fait de l'accumulation des erreurs d'arrondi. C'est par exemple le cas lorsqu'une variable `x` est tout d'abord divisée par une constance `c`, puis multipliée par cette même constante plus tard dans le corps d'une boucle (ce style de constructions est assez courante dans les logiciels qui sont générés automatiquement à partir d'une spécification de plus haut niveau). Un autre exemple est celui d'une variable `x`, dont la valeur serait calculée comme étant la moyenne barycentrique de valeurs précédentes (pas nécessairement consécutives) de `x`.

Nous utilisons des suites arithmético-géométriques [FER 05a] pour borner de tels calculs. L'idée principale consiste à sur-approximer, pour chaque variable $\mathtt{x}$, l'action d'une itération d'une boucle par une transformation linéaire qui agirait sur la borne de la valeur absolue de la variable. Ce faisant, nous obtenons une borne sur la valeur absolue de la valeur de la variable $\mathtt{x}$ qui dépend du compteur de boucle $\mathtt{t}$ (ou du nombre de tics de l'horloge) de manière exponentielle. Plus précisément, nous obtenons une contrainte de la forme suivante :

$$|\mathtt{x}| \leq (1+a)^{\mathtt{t}} \cdot \left(m - \frac{b}{1-a}\right) + \frac{b}{1-a},$$

dans laquelle m est une borne sur la valeur absolue de la variable $\mathtt{x}$ au début de l'itération, $\mathtt{t}$ est le compteur de boucle, alors que a et b sont des très petits coefficients qui sont inférés par l'analyse. En fait, a est de l'ordre de grandeur des erreurs relatives en arithmétique flottante, alors que b est de l'ordre de grandeur du plus petit flottant dénormal (ces deux grandeurs dépendent de la taille des nombres flottants utilisés). Par exemple, si $\mathtt{t}$ varie entre 0 et $3\,600\,000$ (cet intervalle correspond à 10 heures d'exécution d'un programme qui effectue 100 itérations de boucle par seconde), et que nous utilisons des flottants 32 bits, a est généralement environ égal à 10^{-7}, ce qui donne une valeur pour $(1+a)^{\mathtt{t}}$ proche de 1.43, alors qu'avec des flottants 64 bits, a est alors proche de 10^{-13}, ce qui donne environ $1 + 10^{-7}$ pour la valeur de $(1+a)^{\mathtt{t}}$. Ainsi, en pratique, $(1+a)^{\mathtt{t}}$ est suffisamment petit pour prouver que ces divergences lentes n'induisent pas de débordements arithmétiques.

2.4.5.3. *Quaternions*

En trois dimensions, il est possible de représenter la position et l'orientation des objets ainsi que les rotations qui peuvent agir sur ces objets par des quadruplets de nombres réels, appelés quaternions. L'usage des quaternions est très répandu dans les logiciels de l'aérospatial.

Afin de représenter des rotations et la composition de rotations, les quaternions sont munis de primitives algébriques. Ainsi, il est possible d'ajouter deux quaternions $+$, de les soustraire $-$, de les multiplier par un nombre scalaire $\cdot$, de les multiplier deux à deux $\times$, ou encore de les conjuguer $\bar{\cdot}$. De plus, il est courant de convertir les quaternions en matrices de rotation, et inversement. Bien entendu, les calculs sur les quaternions peuvent être la cause de débordements arithmétiques.

La norme $||q||$ d'un quaternion $q = (u, i, j, k)$ est, par définition, égale à la valeur $\sqrt{u^2 + i^2 + j^2 + k^2}$. Souvent, les quaternions sont supposés être normalisés, ce qui signifie qu'ils sont de norme 1. Cependant, du fait des erreurs d'arrondi et des algorithmes approchés qui sont utilisés dans les programmes, leur norme peut diverger (lentement) au cours de l'exécution du programme. De ce fait, les quaternions sont

souvent renormalisés (ce qui revient à les multiplier par l'inverse de leur norme) explicitement en cours d'exécution, afin de prévenir les débordements arithmétiques. Heureusement, la norme se comporte bien par rapport aux autres primitives algébriques, comme indiqué par les propriétés suivantes :

$$\begin{array}{rl} |\,||q_1|| - ||q_2||\,| \leq ||q_1 + q_2|| \leq ||q_1|| + ||q_2|| & \text{(inégalité triangulaire)} \\ |\,||q_1|| - ||q_2||\,| \leq ||q_1 - q_2|| \leq ||q_1|| + ||q_2|| & \text{(inégalité triangulaire)} \\ ||\lambda \cdot q|| = |\lambda| \cdot ||q|| & \text{(homogénéité)} \\ ||q_1 \times q_2|| = ||q_1|| \cdot ||q_2|| & \\ ||\overline{q}|| = ||q||. & \end{array} \tag{2.1}$$

Comme il est difficile de prouver l'absence d'erreurs à l'exécution sans analyser spécifiquement les calculs sur les quaternions, nous avons conçu un domaine numérique spécifique. Ce domaine manipule des prédicats de la forme $Q(\mathtt{x}_1, \mathtt{x}_2, \mathtt{x}_3, \mathtt{x}_4, I)$, dans lesquels $\mathtt{x}_1, \mathtt{x}_2, \mathtt{x}_3, \mathtt{x}_4$ sont quatre variables et I est un intervalle. Un tel prédicat indique que la valeur de l'expression $\sqrt{\mathtt{x}_1^2 + \mathtt{x}_2^2 + \mathtt{x}_3^2 + \mathtt{x}_4^2}$ est dans l'intervalle I. Ainsi, ces prédicats décrivent les propriétés d'intérêt de nos domaines. Afin de propager ces propriétés, nous avons besoin de propriétés intermédiaires pour représenter le fait que la valeur d'une certaine variable est égale à une coordonnée donnée d'un quaternion en cours de calcul. En conséquence, notre domaine manipule des prédicats de la forme $P(\mathtt{x}, i, \phi, \varepsilon)$, dans lesquels $\mathtt{x}$ est une variable, i est un élément de $\{1, 2, 3, 4\}$, ϕ est une formule arithmétique sur les quaternions dans la grammaire suivante :

$$\begin{array}{rcl} \phi & \triangleq & [\mathtt{x}_1, \mathtt{x}_2, \mathtt{x}_3, \mathtt{x}_4] \\ & | & \lambda \cdot \phi \\ & | & \phi_1 \times \phi_2 \\ & | & \phi_1 + \phi_2 \\ & | & \overline{\phi}, \end{array}$$

et ε est un nombre réel positif ou nul.

Un tel prédicat $P(\mathtt{x}, i, \phi, \varepsilon)$ signifie que la i-ème coordonnée du quaternion représenté par l'expression mathématique (sans erreur d'arrondi) ϕ prend une valeur comprise entre $x - \varepsilon$ et $x + \varepsilon$, où x est la valeur de la variable $\mathtt{x}$. Ainsi, les erreurs d'arrondi accumulé lors des opérations sur les quaternions sont bornées par ε. Enfin, la signification des formules arithmétiques est la suivante : la formule $[\mathtt{x}_1, \mathtt{x}_2, \mathtt{x}_3, \mathtt{x}_4]$ décrit le quaternion dont les quatre coordonnées sont les valeurs des variables $\mathtt{x}_1$, $\mathtt{x}_2$, $\mathtt{x}_3$, et $\mathtt{x}_4$, alors que les autres constructions représentent les primitives algébriques correspondantes. Lorsque les quatre coordonnées d'un quaternion donné sont connues (c'est-à-dire que l'analyseur ASTRÉE a pu inférer quatre prédicats $P(\mathtt{x}_1, 1, \phi, \varepsilon_1)$, $P(\mathtt{x}_2, 2, \phi, \varepsilon_2)$, $P(\mathtt{x}_3, 3, \phi, \varepsilon_3)$, and $P(\mathtt{x}_4, 4, \phi, \varepsilon_4)$ dans lesquels ϕ est une formule, $\mathtt{x}_1$, $\mathtt{x}_2$, $\mathtt{x}_3$, et $\mathtt{x}_4$ sont quatre variables du programme, et ε_1, ε_2, ε_3, ε_4 quatre nombres réels positifs ou nuls), le quaternion est promu. Dans ce cas, l'analyseur ASTRÉE découvre un nouveau prédicat $Q(\mathtt{x}_1, \mathtt{x}_2, \mathtt{x}_3, \mathtt{x}_4, I)$ pour lequel l'intervalle I est obtenu

an appliquant les formules (2.1) reliant la norme et les primitives arithmétiques. Enfin, la première inégalité triangulaire est utilisée pour borner la contribution des erreurs d'arrondi. Il est possible, par soucis d'efficacité, de borner la taille des formules ϕ qui peuvent apparaître dans les prédicats.

L'utilisateur peut utiliser une directive spécifique pour déclarer que certains quadruplets $(\mathtt{x}_1, \mathtt{x}_2, \mathtt{x}_3, \mathtt{x}_4)$ forment des quaternions, de norme comprise dans un intervalle I. Cela permet de donner explicitement des hypothèses sur les entrées du programmes. Dans un tel cas, l'analyseur ASTRÉE suppose que le prédicat $Q(\mathtt{x}_1, \mathtt{x}_2, \mathtt{x}_3, \mathtt{x}_4, I)$ est valide, sans aucune vérification. De plus, lorsque les valeurs x_1, x_2, x_3, x_4 des quatre variables $\mathtt{x}_1$, $\mathtt{x}_2$, $\mathtt{x}_3$, et $\mathtt{x}_4$ sont toutes divisées par la valeur de l'expression $\sqrt{x_1^2 + x_2^2 + x_3^2 + x_4^2}$, alors l'analyseur ASTRÉE créé un nouveau prédicat pour le quaternion $[\mathtt{x}_1, \mathtt{x}_2, \mathtt{x}_3, \mathtt{x}_4]$ avec une norme proche de 1 (en prenant en compte les erreurs d'arrondi).

2.4.6. *Combinaison de domaines*

L'analyseur ASTRÉE utilise des douzaines de domaines abstraits susceptibles d'interagir entre eux [COU 06]. Il est ainsi possible de raffiner les propriétés abstraites qui s'expriment dans un domaine, pour tenir compte d'une propriété abstraite d'un autre domaine (voir la section 2.3.6). De plus, certains domaines peuvent utiliser des informations venant des autres domaines pour raffiner ou guider leurs fonctions de transfert. Des précautions sont nécessaires lorsque ces raffinements sont utilisés après une étape d'extrapolation (élargissement ou rétrécissement), afin de ne pas détruire leur propriété de terminaison et empêcher ainsi la construction en temps fini d'un invariant inductif.

Dans l'analyseur ASTRÉE, les domaines abstraits sont des modules indépendants, qui partagent une interface commune. Ils implantent des primitives de calcul comme les fonctions de transfert (affectation abstraite, test abstrait, union de flots de contrôle) et des opérateurs d'extrapolation (élargissement, rétrécissement). De plus, afin de permettre la collaboration entre les différents domaines, chaque domaine abstrait dispose de primitives pour exprimer ses propres propriétés abstraites sous un format partagé qui peut être compris par tous les (autres) domaines. Dit simplement, une étape de réduction doit être initiée lors d'un calcul dans un domaine abstrait. Il y a alors deux cas : soit la réduction est demandée par un domaine auquel il manque une certaine propriété (ce domaine demande alors si cette information est disponible dans les autres domaines), ou la réduction est demandée par le domaine qui calcule une propriété qu'il estime assez importante pour la communiquer aux autres domaines.

Donnons maintenant, un peu plus de détails sur ces différents types de raffinements. Comme écrit précédemment, les domaines abstraits peuvent demander une propriété qui leur manque. Pour cela, nous utilisons ce que nous appelons un *canal*

d'entrée. Ce canal d'entrée propage les contraintes qui ont déjà été découvertes jusqu'à maintenant, par les domaines abstraits. Cet ensemble de contraintes peut être mis à jour après chaque nouveau calcul dans un domaine abstrait. Pour des raisons d'efficacité, ce canal d'entrée est géré de manière paresseuse, ce qui permet de n'effectuer les calculs que s'ils sont effectivement nécessaires. De plus, l'utilisation d'un cache permet de ne faire qu'une seule fois les calculs, même si ils sont utilisés plusieurs fois.

Grâce au canal d'entrée, un domaine peut demander une propriété à propos d'une pré-condition (une propriété sur l'état du programme avant d'effectuer un pas de calcul). Par exemple, lorsque le programme rencontre la première itération d'un filtre de second ordre, il n'y a pas de borne disponible pour la valeur de la forme quadratique associée à ce filtre. Aussi, le domaine des filtres numériques doit demander un intervalle de variations pour les variables de sortie du filtre, et en déduire une borne pour la valeur de la forme quadratique. Par ailleurs, un domaine peut également demander une propriété à propos d'une post-condition (une propriété sur l'état du programme après le pas de calcul courant). Par exemple, dans certains cas, le domaine des octogones ne peut pas calculer l'effet d'une fonction de transfert. Ce serait le cas en particulier si le domaine des octogones devait interpréter l'affectation $\mathtt{x} = \mathtt{y}$, sur un octogone qui porterait sur la variable `x`, mais pas sur la variable `y`. Dans un tel cas, le domaine des octogones utilise le domaine des intervalles pour obtenir l'intervalle de variation de la variable `x` après l'étape de calcul.

Il faut garder à l'esprit que l'ordre des calculs a sont importance : seuls les propriétés qui ont déjà été calculées dans les domaines abstraits sont disponibles sur le canal d'entrée. Aussi, un autre canal est nécessaire pour permettre le raffinement des propriétés d'un domaine par d'autres propriétés qui n'auraient pas été encore inférées. C'est le but du *canal de sortie*. Le canal de sortie peut être utilisé dès qu'une nouvelle propriété est découverte par un domaine, et que ce dernier veut avertir les domaines qui ont déjà effectué leurs calculs. C'est le cas par exemple lors de l'itération d'un filtre. L'intervalle de variation qui est trouvé par le domaine des filtres pour la variable de sortie d'un filtre est toujours meilleur que celui trouvé par le domaine des intervalles, aussi l'intervalle qui est trouvé par le domaine des filtres est donné aux domaines des intervalles grâce au canal de sortie.

La plupart des étapes de réduction peuvent être vues comme des raffinements de l'état abstrait, c'est-à-dire comme un produit réduit approché tel que défini dans la section 2.3.6.1. Notons D le domaine abstrait composite, et γ la fonction de concrétisation qui associe à chaque élément abstrait $d \in D$, l'ensemble des états concrets correspondant. Un opérateur de réduction ρ est une fonction conservative, ce qui signifie que, pour tout élément abstrait $d \in D$: $\gamma(d) \subseteq \gamma(\rho(d))$. Le fait de remplacer d par $\rho(d)$ dans un calcul, est appelé un raffinement de l'état abstrait. Cependant, dans certains cas, la collaboration entre domaines abstraits permet également le raffinement des fonctions de transfert. Par exemple, la plupart des domaines abstraits utilisent des expressions linéaires (avec des intervalles comme coefficients) et, dès

qu'une expression n'est pas linéaire, cette expression est *linéarisée* en remplaçant certaines sous-expressions par leur intervalle de variation. Il n'y a pas de manière canonique de choisir quelles expressions doivent ainsi être remplacées par leur intervalle, aussi l'analyseur ASTRÉE doit recourir à des stratégies. Par exemple, pour linéariser le produit entre les valeurs des variables x et y, l'analyseur ASTRÉE doit choisir de garder l'une des deux variables x et y, et de remplacer l'autre par un intervalle. Ce choix est guidé par les propriétés disponibles sur les variables x et y, comme leur intervalle de variation par exemple, dans les autres domaines (grâce au canal d'entrée).

Enfin, des étapes de raffinement peuvent aussi être nécessaires en sortie des opérateurs d'extrapolation (élargissement et rétrécissement). Cependant, des précautions doivent être prises pour ne pas empêcher l'extrapolation d'un invariant inductif, et ainsi aboutir à la non terminaison de l'analyse. Des exemples de mauvaises interactions entre raffinement et extrapolation peuvent être consultés dans [MIN 04b, p. 85]. De plus, alterner des applications d'opérateur d'élargissement avec des unions classiques pendant des itérations ascendantes, ou même intersecter le résultat de chaque itération par un élément abstrait constant peut conduire à des analyses qui ne terminent pas (toujours). Plus formellement, nous avons donné dans une publication [COU 06, Sect. 7] un exemple de suite $(d_n) \in D^{\mathbb{N}}$ d'éléments abstraits dans un domaine abstrait D muni d'une fonction de concrétisation γ, d'un opérateur d'union $\sqcup$ (vérifiant $\gamma(d_1) \cup \gamma(d_2) \subseteq \gamma(d_1 \sqcup d_2)$), d'un opérateur d'intersection (tel que $\gamma(d_1) \cap \gamma(d_2) \subseteq \gamma(d_1 \sqcap d_2)$), d'un opérateur d'élargissement ∇ (tel que $\gamma(d_1) \cup \gamma(d_2) \subseteq \gamma(d_1 \nabla d_2)$) d'un opérateur de réduction ρ (satisfaisant $\gamma(d) \subseteq \gamma(\rho(d))$), et tel qu'aucune des trois suites (a_n), (b_n), (c_n) définies ci-dessous :

$$\begin{cases} a_1 = d_1 \\ a_{n+1} = (a_n \nabla d_{n+1}) \sqcap d_0 \end{cases} \qquad \begin{cases} b_1 = d_1 \\ b_{n+1} = \rho(b_n \nabla d_{n+1}) \end{cases} \qquad \begin{cases} c_1 = d_1 \\ c_{2\cdot n+2} = c_{2\cdot n+1} \cup d_{2\cdot n+2} \\ c_{2\cdot n+1} = c_{2\cdot n} \nabla d_{2\cdot n+1}, \end{cases}$$

ne stationne à partir d'un certain rang.

Néanmoins, raffiner le résultat des étapes d'élargissement est très important, afin d'éviter des pertes irrémédiables d'information. Pour pallier à ce problème, nous imposons des contraintes supplémentaires sur les domaines abstraits, leurs primitives et l'opérateur de réduction ρ_∇ qui est utilisé après les étapes d'élargissement. Ainsi, nous exigeons que (1) chaque domaine abstrait D soit un produit cartésien fini $\prod_{j\in J} D_j$ de domaines $(D_j, \leq_j)$ totalement ordonnés (par exemple, un intervalle pour une variable donnée devra être vue comme une paire de bornes, un octogone comme une famille de bornes, et ainsi de suite), que (2) l'union, l'intersection et l'opérateur d'élargissement soient définis composante par composante à partir des opérateurs sur les domaines (D_j), que (3) pour chaque domaine D_j, l'union $\sqcup_j$, l'intersection $\sqcap_j$, et l'opérateur d'élargissement ∇_j soient compatibles avec les ordres totaux $\leq_j$ (c'est-à-dire que pour tout $j \in J$ et pour toute paire $(a, b) \in D_j^2$, nous avons : $a_j \leq_j a_j \sqcup_j b_j$, $b_j \leq_j a_j \sqcup_j b_j$, $a_j \sqcap_j b_j \leq_j a_j$, $a_j \sqcap_j b_j \leq_j b_j$, $a_j \leq_j a_j \nabla_j b_j$, et

$b_j \leq_j a_j \nabla_j b_j$), et que (4) il n'y a pas de cycle de réductions entre les domaines D_j — ce qui signifie qu'il existe une relation acyclique $\rightarrow$ sur J qui vérifie que pour toute paire d'éléments abstraits $d, d' \in D$ (nous notons $d = (x_j)_{j \in J}$ et $d' = (x'_j)_{j \in J}$, $\rho_\nabla(d) = (y_j)_{j \in J}$, et $\rho_\nabla(d') = (y'_j)_{j \in J}$), nous avons, pour tout j élément de J, $[x_j = x'_j \text{ et } \forall k \in J : k \rightarrow j \implies x_k = x'_k] \implies y_j = y'_j$. Ainsi, ces hypothèses supplémentaires assurent la terminaison de l'analyse même lorsque l'opérateur de réduction ρ_∇ est appliqué en sortie des étapes d'élargissement, si des étapes d'élargissement sont remplacées par des unions classiques (pourvu que pour chaque domaine D_j, ses éléments soient bien élargies un nombre infini de fois). De plus, il faut noter que ces hypothèses supplémentaires ne portent que sur les raffinements utilisés en sortie d'étape d'extrapolation, et que d'autres raffinement plus permissifs peuvent être utilisés lors du calcul des fonctions de transfert.

2.5. Applications

Nous nous plaçons maintenant du point de vue d'un ingénieur utilisateur d'ASTRÉE. Nous décrivons sa mis en oeuvre générale et donnons plusieurs exemples d'applications industrielles.

2.5.1. *Mise en oeuvre de l'analyseur*

ASTRÉE est un outil entièrement automatique. Il prend en entrée l'ensemble des sources C du programme, le nom d'un point d'entrée et quelques paramètres spécifiant la machine cible (ordre des octets, taille des types de base, comportement en cas de dépassement de capacité arithmétique, etc.). En sortie, nous obtenons une liste des erreurs à l'exécution pouvant se produire, leur nature, point de programme et certaines informations de contexte. Il faut tout de même s'assurer au préalable que le code C appartient au sous-ensemble traité (C99 sans récursion, parallélisme, ni allocation dynamique de mémoire) et ne comporte aucun symbole non-défini. En particuliers, les sources des bibliothèques utilisées doivent être fournis. Il est toutefois possible (et même indispensable, dans le cas de bibliothèques systèmes ou écrites en assembleur) de remplacer certaines procédures par des *stubs* modélisant leur effet de manière abstraite. Ainsi, par exemple, le *stub* suivant :

```
double sqrt(double x) {
  double ret;
  __ASTREE_assert((x>=0));
  __ASTREE_known_fact((ret>=0));
  return ret;
}
```

modélise grossièrement la fonction `sqrt` en indiquant que la valeur de retour `res` est positive, et qu'il faut indiquer une erreur quand l'entrée `x` n'est pas positive. Dans le

cas fréquent où le programme s'exécute dans un environnement avec lequel il interagit (autres programmes, système de fichiers, capteurs physiques, etc.), des directives `__ASTREE_volatile_input` permettent de spécifier les bornes des variables modifiables par l'environnent. Une seule exécution d'ASTRÉE considérera toutes les valeurs spécifiées lors de chaque lecture d'une telle variable, couvrant ainsi tout l'espace des données (contrairement à un test qui choisirait une valeur pour chaque exécution).

S'il est possible, grâce à l'automatisation complète d'ASTRÉE, d'obtenir aisément et rapidement une première analyse, celle-ci peut se révéler imprécise (i.e., comportant des fausses alarmes). ASTRÉE possède de nombreuses options permettant de contrôler finement le rapport coût / précision (e.g., les domaines abstraits actifs, le degré de pliage des tableaux, l'agressivité des élargissements). Il est également possible d'insérer des directives matérialisant les portions du source où une plus grande précision est souhaitée (e.g., directives des partitionement, de paquets booléens et octogonaux). L'expérience [DEL 07] montre qu'une formation légère est suffisante à des ingénieurs non spécialistes afin d'exploiter au mieux ces paramètres. Toutefois, ASTRÉE étant fondamentalement incomplet, il existe des cas où aucun paramètre ne pourra éliminer une fausse alarme. Dans ces cas, il est nécessaire de modifier l'outil lui-même, par exemple en améliorant des fonctions de transferts imprécises, ou en ajoutant un nouveau domaine abstrait. Cette dernière opération ne peut généralement être faite que par un spécialiste d'interprétation abstraite à l'issu d'un travail de recherche théorique, mais est facilitée du point de vue de l'implantation grâce à l'architecture modulaire d'ASTRÉE. Il est important de préciser qu'un nouveau domaine abstrait ne se contente pas d'éliminer une alarme donnée dans un programme donné, mais résout le plus souvent toute une série d'imprécisions du même type dans une famille de programmes de même forme, puisqu'un tel domaine infère une famille infinie de propriétés. Nous faisons ainsi le pari que, après avoir complété ASTRÉE pour le rendre précis sur quelques programmes d'un type donné, la bibliothèque d'abstractions développée est suffisante pour traiter tous les programmes de ce type (en modifiant au besoin quelques paramètres d'analyse simples, mais sans modification supplémentaire de l'outil ni intervention d'un spécialiste d'interprétation abstraite). Les deux sections suivantes justifient cette idée par quelques cas d'étude.

2.5.2. *Applications au domaine avionique*

Une première application d'ASTRÉE est la preuve d'absence d'erreurs à l'exécution dans deux familles de codes de contrôle / commande avioniques industriels. Il s'agit de codes C synchrones, générés par l'outil de spécification graphique SAO (similaire à l'outil SCADE [Est 10]). La structure d'un tel programme est schématisée en figure 2.5.2. Les difficultés d'analyse de ces codes proviennent de leur grande taille (jusqu'à 800 K lignes de code, commentaires et lignes vides exclus), leur grand nombre de variables d'état (jusqu'à 10 K de variables globales, vivantes en tout point du programme), l'absence de structure (le générateur de code ayant tendance à aplatir

```
déclaration des variables d’état, d’entrées, de sorties
initialisation des variables d’état
boucler pendant 10h
    lire les entrées
    mettre à jour les l’état et calculer les sorties
    écrire les sorties
    attendre le prochain coup d’horloge
```

Figure 2.1. *Structure générale d’un code de contrôle / commande synchrone.*

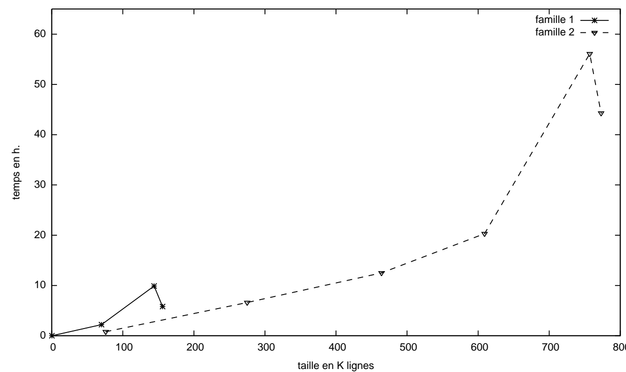


Figure 2.2. *Temps de calcul en fonction de la taille du code (la taille en milliers de lignes exclut les commentaires et les lignes vides)* .

la structure des calculs et à les disperser tout au long du programme), et l’utilisation massive de calculs en virgule flottante. En contrepartie, le code étant généré, il est très régulier, ce qui facilite l’élaboration de stratégies qui, par simples critères syntaxiques, déterminent automatiquement la précision locale nécessaire (partitionement, paquets d’octogones, etc.).

Afin d’obtenir une analyse à la fois efficace et précise, nous sommes parti d’une analyse très simple d’intervalles pour analyser des petits fragments d’une famille de codes avioniques. Nous avons alors raffiné l’embryon d’ASTRÉE par l’ajout de domaines, d’utilité générale en analyse statique (section 2.4.4) ou bien plus spécifiquement liés au domaine du contrôle avionique (section 2.4.5), jusqu’à réduire à zéro le nombre d’alarmes. Nous avons alors itéré ce processus sur des fragments plus gros jusqu’à l’analyse de codes complets (environ 150 K lignes). Ce développement initial a pris trois ans (2001–2003). Nous avons ensuite considéré l’analyse d’une famille de logiciels de contrôle / commande avionique plus récente. Grâce aux acquis du premier

développement et malgré la complexité et la taille accrus du logiciel (800 K lignes) et l'utilisation d'un générateur de code sensiblement différent, l'adaptation d'ASTRÉE s'est déroulée plus rapidement (2003–2004). Le résultat est un analyseur automatique qui prouve l'absence d'erreurs à l'exécution (zéro alarme) sur tous les codes considérés lors du développement, et est de plus exploitable par des utilisateurs industriels [DEL 07] sur de nouveaux codes issus des mêmes familles, moyennant la modification de quelques paramètres simples. La figure 2.5.2 donne l'efficacité de l'analyse de codes pour ces deux familles sur notre serveur intel 2.66 GHz en rapportant le temps de calcul à la taille du code. Pour chaque famille, le code le plus gros correspond à la dernière version de chaque famille, pour laquelle l'analyse est généralement plus optimisée (d'où une baisse du temps de calcul). Notons enfin que nos utilisateurs industriels ont également appliqué ASTRÉE à l'analyse d'autres codes avioniques qui n'appartiennent pas strictement au sous-ensemble de contrôle / commande (il s'agit de logiciels d'initialisation et de test du matériel, et de logiciels de communication sur les bus), mais sur lesquels ASTRÉE donne des résultat satisfaisants, i.e., peu de fausses alarmes (voir [DEL 07]).

2.5.3. *Application au domaine spatial*

Un deuxième application d'ASTRÉE est l'analyse de logiciels de contrôle / commande issus de l'industrie spatiale. Notre cas d'étude, considéré lors d'un projet avec l'Agence spatiale européenne ([BOU 09], 2006–2008), est une version C du module de sûreté (MSU, i.e., *Monitoring and Safing Unit*) du logiciel contrôlant le véhicule automatique de transfert européen (*ATV*) servant à ravitailler la station spatiale internationale. Alors que la version opérationnelle est programmée en Ada, la version développée par Astrium ST pour l'étude est en C et générée par SCADE V6 [Est 10]. Toutefois, la version d'étude est bien représentative du type programmes utilisés dans l'industrie spatiale.

Le logiciel MSU est modeste (14 K lignes) et montre des similarités avec les logiciels avioniques analysés précédemment (tous deux sont des logiciels de contrôle / commande fortement numériques). Par conséquent, une analyse avec la version d'ASTRÉE spécialisée pour les logiciels avioniques s'est révélée relativement précise, avec un nombre de fausses alarmes assez faible. Toutefois, certains calculs utilisés dans le contrôle spatial sont d'une nature spécifique, très différente des calculs du contrôle avionique, telle l'utilisation de quaternions. L'ajout d'un domaine numérique pour les quaternions (section 2.4.5.3) s'est révélé nécessaire afin d'éliminer les alarmes restantes et prouver l'absence d'erreurs à l'exécution. L'analyse précise avec ce nouveau domaine prend environ une heure.

Cette expérience montre qu'il est possible d'adapter un analyseur déjà spécialisé tel qu'ASTRÉE à un domaine d'application connexe afin d'obtenir une analyse à la fois précise et rapide, au prix d'un travail de recherche et d'implantation modéré. D'autres

domaines d'application (industrie automobile, de l'énergie, etc.) restent encore à étudier.

2.5.4. *Industrialisation*

ASTRÉE a débuté par un prototype en 2001 qui a connu un succès académique [BLA 03] et a progressivement évolué vers un outil utilisable dans un contexte industriel [DEL 07]. Bien qu'initialement spécialisé pour l'analyse de logiciels de contrôle / commande synchrones avioniques, sur lesquels il donne d'excellents résultats, il s'est révélé également utile pour analyser en l'état des codes synchrones d'autres sortes et dans d'autres domaines d'application. Ce succès s'est traduit par son industrialisation par AbsInt Angewandte Informatik GmbH [Abs 10], entreprise spécialisée dans les logiciels d'analyse statique basés sur l'interprétation abstraite (par exemple, l'analyse de temps d'exécution [HEC 04]). ASTRÉE est commercialisé depuis 2009 [KÄS 10].

2.6. Conclusion

ASTRÉE permet une analyse complètement automatique de programmes C, qui produit une sur-approximation des erreurs d'exécution possibles. Sur plusieurs familles importantes de logiciels industriels, nous avons obtenu une preuve d'absence totale d'erreurs, l'analyse nécessitant un temps de calcul raisonnable. Cette approche répond donc aux critères industriels classiques, qui spécifient en particulier que l'analyse doit se fondre dans le processus de développement (i.e., ne pas nécessiter un temps de calcul prohibitif, qui interromprait le processus) sans exiger un temps de travail important de la part des utilisateurs. Elle répond également aux exigences des normes qui régissent le développement des logiciels critiques, tels que le DO 178 B [AVI 99] (l'analyse statique par interprétation abstraite fera explicitement partie des approches recommandées dans la version "C" de cette norme, qui doit paraître en 2010).

D'un autre côté, l'analyse n'est pas complète, et il faut donc s'attendre à ce que l'analyse produise des fausses alarmes. Nous avons vu que cette limitation théorique n'est pas un problème sérieux en pratique, car il est possible de concevoir et paramétrer l'analyseur de manière à ce qu'il soit particulièrement précis sur certaines familles spécifiques de programmes, comme nous l'avons fait dans les domaines aéronautique et spatial.

Pour parvenir à ces résultats, nous nous sommes fondés sur la théorie de l'interprétation abstraite [COU 77b], qui permet de décrire formellement l'approximation sûre d'une sémantique concrète par une sémantique abstraite calculable. Ce fondement théorique solide a également permis de concevoir l'analyse de manière modulaire, c'est-à-dire en isolant les composants fondamentaux (itérateurs abstrait, domaines

abstraits), ce qui a aidé significativement la conception et l'implantation d'ASTRÉE, tout en garantissant la possibilité de l'étendre à l'analyse de nouvelles familles d'applications.

2.7. Bibliographie

[Abs 10] ABSINT, ANGEWANDTE INFORMATIK, ASTRÉE Run-Time Error Analyzer, `http://www.absint.com/astree/`, Jan. 2010.

[AVI 99] TECHNICAL COMMISSION ON AVIATION R., DO-178B, Rapport, Software Considerations in Airborne Systems and Equipment Certification, 1999.

[BER 10] BERTRANE J., COUSOT P., COUSOT R., FERET J., MAUBORGNE L., MINÉ A., MONNIAUX D., RIVAL X., « Static Analysis by Abstract Interpretation of Embedded Critical Software », *AIAA (American Institute of Aeronautics and Astronautics) Infotech Aerospace*, Atlanta (Georgia, USA), avril 2010, 2010.

[BLA 02] BLANCHET B., COUSOT P., COUSOT R., FERET J., MAUBORGNE L., MINÉ A., MONNIAUX D., RIVAL X., « Design and Implementation of a Special-Purpose Static Program Analyzer for Safety-Critical Real-Time Embedded Software », MOGENSEN T., SCHMIDT D., SUDBOROUGH I., Eds., *The Essence of Computation : Complexity, Analysis, Transformation. Essays Dedicated to Neil D. Jones*, vol. 2566 de *LNCS*, p. 85–108, Springer, Heidelberg, 2002.

[BLA 03] BLANCHET B., COUSOT P., COUSOT R., FERET J., MAUBORGNE L., MINÉ A., MONNIAUX D., RIVAL X., « A Static Analyzer for Large Safety-Critical Software », *Proc. of the ACM SIGPLAN 2003 Conf. on Programming Language Design and Implementation (PLDI'03)*, San Diego, USA, 7–14 juin 2003, ACM Press (New York), p. 196–207, 2003.

[BOU 92] BOURDONCLE F., « Abstract Interpretation by Dynamic Partitioning », *Journal of Functional Programming*, vol. 2 (4), p. 407–423, 1992.

[BOU 09] BOUISSOU O., CONQUET E., COUSOT P., COUSOT R., FERET J., GOUBAULT E., GHORBAL K., LESENS D., MAUBORGNE L., MINÉ A., PUTOT S., RIVAL X., TURIN M., « Space Software Validation using Abstract Interpretation », *Proc. of the Int. Space System Engineering Conference, Data Systems In Aerospace (DASIA'09)*, Istanbul, Turkey, 26–29 mai 2009, ESA publications, p. 1–7, 2009.

[BRA 28] BRAHMAGUPTA, Brahma Sphuta Siddhanta, (voir Plofker, K. « Mathematics in India », Princeton University Press, 2008), 628.

[BRY 86] BRYANT R. E., « Graph-Based Algorithms for Boolean Function Manipulation », *IEEE Transactions on Computers*, vol. 35, p. 677–691, 1986.

[BUR 74] BURSTALL R. M., « Program proving as hand simulation with a little induction », ROSENFELD J. L., Ed., *Information Processing 74*, Amsterdam, Holland, p. 308–312, 1974.

[CLA 00] CLARKE E. M., GRUMBERG O., JHA S., LU Y., VEITH H., « Counterexample-Guided Abstraction Refinement », *Proc. of the 12th Int. Conf. on Computer Aided Verification (CAV'00)*, vol. 1855 de *LNCS*, Springer, Heidelberg, p. 154–169, Jul. 2000.

[COU 77a] COUSOT P., Asynchronous iterative methods for solving a fixed point system of monotone equations in a complete lattice, Research report R.R. 88, Laboratoire IMAG, Université scientifique et médicale de Grenoble, Grenoble, september 1977.

[COU 77b] COUSOT P., COUSOT R., « Abstract interpretation : a unified lattice model for static analysis of programs by construction or approximation of fixpoints », *Conf. Rec. of the 4th Annual ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages (POPL'77)*, Los Angeles, USA, janvier 1977, ACM Press (New York), p. 238–252, 1977.

[COU 78a] COUSOT P., Méthodes itératives de construction et d'approximation de points fixes d'opérateurs monotones sur un treillis, analyse sémantique de programmes, Thèse d'État ès sciences mathématiques, Université scientifique et médicale de Grenoble, Grenoble, France, 1978.

[COU 78b] COUSOT P., HALBWACHS N., « Automatic discovery of linear restraints among variables of a program », *Conf. Rec. of the 5th Annual ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages (POPL'78)*, Tucson, USA, janvier 1978, ACM Press (New York), p. 84–97, 1978.

[COU 79] COUSOT P., COUSOT R., « Systematic design of program analysis frameworks », *Conf. Rec. of the 6th Annual ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages (POPL'79)*, San Antonio, USA, janvier 1979, ACM Press (New York), p. 269–282, 1979.

[COU 81] COUSOT P., « Semantic Foundations of Program Analysis », MUCHNICK S., JONES N., Eds., *Program Flow Analysis : Theory and Applications*, Chapitre 10, p. 303–342, Prentice-Hall, Inc., Englewood Cliffs, 1981.

[COU 92] COUSOT P., COUSOT R., « Abstract Interpretation Frameworks », *Journal of Logic and Computation*, vol. 2 (4), p. 511–547, Oxford University Press, Oxford, 1992.

[COU 97] COUSOT P., « Types as Abstract Interpretations », *Conf. Rec. of the 24th Annual ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages (POPL'97)*, Paris, France, 15–17 janvier 1997, ACM Press (New York), p. 316–331, 1997.

[COU 05] COUSOT P., COUSOT R., FERET J., MAUBORGNE L., MINÉ A., MONNIAUX D., RIVAL X., « The ASTRÉE analyser », SAGIV M., Ed., *Proc. of the 14th European Symposium on Programming Languages and Systems (ESOP'05)*, vol. 3444 de *LNCS*, p. 21–30, Springer, Heidelberg, 2005.

[COU 06] COUSOT P., COUSOT R., FERET J., MAUBORGNE L., MINÉ A., MONNIAUX D., RIVAL X., « Combination of Abstractions in the ASTRÉE Static Analyzer », OKADA M., SATOH I., Eds., *Proc. of the 11th Annual Asian Computing Science Conference (ASIAN'06)*, vol. 4435 de *LNCS*, Tokyo, Japan, 6–8 décembre 2006, Springer, Heidelberg, p. 272–300, 2006.

[COU 07a] COUSOT P., COUSOT R., FERET J., MAUBORGNE L., MINÉ A., MONNIAUX D., RIVAL X., « Varieties of Static Analyzers : A Comparison with ASTRÉE », HINCHEY M., JIFENG H., SANDERS J., Eds., *Proc. of the First Symp. on Theoretical Aspects of Software Engineering (TASE'07)*, Shanghai, China, 6–8 juin 2007, IEEE Comp. Soc. Press, p. 3–17, 2007.

[COU 07b] COUSOT P., P.GANTY, RASKIN J.-F., « Fixpoint-Guided Abstraction Refinements », *Proc. of the 14th Int. Symp. on Static Analysis, (SAS'07)*, vol. 4634 de *LNCS*, Springer, Heidelberg, p. 333–348, Aug. 2007.

[DEL 07] DELMAS D., SOUYRIS J., « ASTRÉE : from Research to Industry », FILÉ G., RIIS-NIELSON H., Eds., *Proc. of the 14th Int. Static Analysis Symposium (SAS'07)*, vol. 4634 de *LNCS*, p. 437–451, Springer, Heidelberg, Kongens Lyngby, Denmark, 2007.

[Est 10] ESTEREL TECHNOLOGIES, SCADE Suite™, The Standard for the Development of Safety-Critical Embedded Software in the Avionics Industry, `http://www.esterel-technologies.com/`, 2010.

[Euc 00] EUCLIDE D'ALEXANDRIE, Elementa Geometriæ, Book XII, proposition 17, `http://aleph0.clarku.edu/~djoyce/java/elements/bookXII/propXII17.html`, environ -300.

[FER 04] FERET J., « Static Analysis of Digital Filters », SCHMIDT D., Ed., *Proc. of the 13th European Symp. on Programming Languages and Systems (ESOP'04)*, vol. 2986 de *LNCS*, Springer, Heidelberg, 27–31 mars 2004, p. 33–48, 2004.

[FER 05a] FERET J., « The Arithmetic-Geometric Progression Abstract Domain », COUSOT R., Ed., *Proc. of the 6th Int. Conf. on Verification, Model Checking and Abstract Interpretation (VMCAI'05)*, vol. 3385 de *LNCS*, Paris, France, 17–19 janvier 2005, Springer, Heidelberg, p. 42–58, 2005.

[FER 05b] FERET J., « Numerical Abstract Domains for Digital Filters », *Proc. of the First Int. Workshop on Numerical & Symbolic Abstract Domains (NSAD'05)*, Paris, France, 21 janvier 2005, 2005.

[FLO 67] FLOYD R. W., « Assigning meanings to programs », *Proc. of the American Mathematical Society Symposia on Applied Mathematics*, vol. 19, Providence, USA, p. 19–32, 1967.

[GOU 01] GOUBAULT E., « Static analyses of floating-point operations », *Proc. of the 8th Int. Static Analysis Symposium (SAS'01)*, vol. 2126 de *LNCS*, Springer, Heidelberg, p. 234–259, 2001.

[GRA 89] GRANGER P., « Static Analysis of Arithmetical Congruences », *Int. J. Comput. Math.*, vol. 30 (3 & 4), p. 165–190, 1989.

[GRA 91] GRANGER P., « Static Analysis of Linear Congruence Equalities among Variables of a Program », ABRAMSKY S., MAIBAUM T., Eds., *Proc. of the Int. Joint Conf. on Theory and Practice of Software Development (TAPSOFT'91), Volume 1 (CAAP'91)*, vol. 493 de *LNCS*, Brighton, UK, Springer, Heidelberg, p. 169–192, 1991.

[HEC 04] HECKMANN R., FERDINAND C., « Worst-Case Execution Time Prediction by Static Program Analysis », *Proc. of the 18th Int. Parallel and Distributed Processing Symposium (IPDPS'04)*, IEEE Computer Society, p. 26–30, 2004.

[IEE 85] IEEE COMPUTER SOCIETY, IEEE Standard for Binary Floating-Point Arithmetic, Rapport, ANSI/IEEE Std. 745-1985, 1985.

[ISO 07] ISO/IEC JTC1/SC22/WG14 WORKING GROUP, C standard, Rapport 1124, Rapport, ISO & IEC, 2007.

[JEA 07] JEANNET B., MINÉ A., The `Apron` Numerical Abstract Domain Library, `http://apron.cri.ensmp.fr/library/`, 2007.

[JEA 09] JEANNET B., MINÉ A., « `Apron` : A library of numerical abstract domains for static analysis », *Proc. of the 21st Int. Conf. on Computer Aided Verification (CAV'09)*, vol. 5643 de *LNCS*, Grenoble, France, Springer, Heidelberg, p. 661–667, 2009.

[KÄS 10] KÄSTNER D., WILHELM S., NENOVA S., COUSOT P., COUSOT R., FERET J., MAUBORGNE L., MINÉ A., RIVAL X., « ASTRÉE : Proving the Absence of Rutime Errors », *Proc. of Embedded Real-Time Software and Systems (ERTS'10)*, Toulouse, France, 19–21 mai 2010, p. 1–5, 2010.

[LAR 97] LARSEN K., LARSSON F., PETTERSSON P., YI W., « Efficient Verification of Real-Time Systems : Compact Data Structure and State-Space Reduction », *Proc. of the 18th IEEE Real-Time Systems Symp. (RTSS'97)*, IEEE CS Press, p. 14–24, 1997.

[LIO 96] LIONS J.-L., ET AL., « ARIANE 5, Flight 501 Failure, Report by the Inquiry Board », 1996.

[MAU 05] MAUBORGNE L., RIVAL X., « Trace Partitioning in Abstract Interpretation Based Static Analyzer », SAGIV M., Ed., *Proc. of the 14th European Symp. on Programming Languages and Systems (ESOP'05)*, vol. 3444 de *LNCS*, p. 5–20, Springer, Heidelberg, Edinburg, UK, 2005.

[MIN 04a] MINÉ A., « Relational Abstract Domains for the Detection of Floating-Point Run-Time Errors », SCHMIDT D., Ed., *Proc. of the 13th European Symp. on Programming Languages and Systems (ESOP'04)*, vol. 2986 de *LNCS*, Springer, Heidelberg, 27 mars –4 avril 2004, p. 3–17, 2004.

[MIN 04b] MINÉ A., Weakly Relational Numerical Abstract Domains, Thèse de doctorat en informatique, École polytechnique, Palaiseau, France, 2004.

[MIN 06a] MINÉ A., « Field-Sensitive Value Analysis of Embedded C Programs with Union Types and Pointer Arithmetics », *ACM SIGPLAN/SIGBED Conf. on Languages, Compilers, and Tools for Embedded Systems (LCTES'06)*, Ottawa, Ontario, Canada, ACM Press, p. 54–63, June 2006.

[MIN 06b] MINÉ A., « The Octagon Abstract Domain », *Higher-Order and Symbolic Computation*, vol. 19, p. 31–100, 2006.

[MON 05] MONNIAUX D., « The parallel implementation of the ASTRÉE static analyzer », *Proc. of the 3rd Asian Symp. on Programming Languages and Systems (APLAS'05)*, vol. 3780 de *LNCS*, Springer, Heidelberg, 3–5 novembre 2005, p. 86–96, 2005.

[MOO 66] MOORE R. E., *Interval Analysis*, Prentice Hall, Englewood Cliffs N. J., USA, 1966.

[NEU 89] NEUMANN, P.G. (moderator), « The risks digest », *Forum on Risks to the Public in Computers and Related Systems, ACM Committee on Computers and Public Policy*, vol. 8 (49), 5 April 1989, `http://catless.ncl.ac.uk/risks/8.49.html#subj2`.

[RIV 07] RIVAL X., MAUBORGNE L., « The trace partitioning abstract domain », *ACM Trans. Program. Lang. Syst.*, vol. 29 (5), 2007.

[SKE 92] SKEEL R., « Roundoff Error and the Patriot Missile », *SIAM News*, vol. 25 (4), 1992.

[TAR 55] TARSKI A., « A lattice theoretical fixpoint theorem and its applications », *Pacific Journal of Mathematics*, vol. 5, p. 285–310, 1955.