Contraintes sur des flux appliquées à la vérification de programmes audio

Anicet Bart[1] **Charlotte Truchet**[2] **Éric Monfroy**[2]

LINA - Laboratoire d'Informatique de Nantes-Atlantique - UMR 6241
[1] Écoles des Mines de Nantes - 4, rue Alfred Kastler 44000 NANTES
[2] Université de Nantes - 2, rue de la Houssinière 44000 NANTES
[1]{prenom.nom}@mines-nantes.fr - [2]{prenom.nom}@univ-nantes.fr

Résumé

La programmation par contraintes s'attaque en général à des problèmes statiques, sans notion de temps. Cependant, les méthodes de réduction de domaines pourraient par exemple être utiles dans des problèmes portant sur des flux. C'est le cas de la vérification de programmes temps-réel, avec des variables dont les valeurs peuvent changer à chaque pas de temps. Dans cet article, nous nous intéressons à la vérification de domaines de variables (flux) dans le cadre d'un langage de diagrammes de blocs. Nous proposons une méthode de réduction de domaines de ces flux, pour encadrer finement les valeurs prises au cours du temps. En particulier, nous proposons un nouvel algorithme pour calculer un point fixe dans le cas des boucles temporelles. Nous présentons ensuite une application au langage FAUST, un langage fonctionnel temps réel pour le traitement audio et nous testons notre approche sur différents programmes FAUST standards.

Abstract

Constraint programming usually deals with static problems. However, domain reduction method could be useful in stream-based problems. This is the case in formal verification of real time programs for which variables can be assigned different values at every single time. In this paper, we focus on domain checking of stream variables in the context of block diagram languages. We propose a reduction algorithm for streams in order to tightly reduce their domains all over the time. Particularly, we propose a new technique to compute fix points of temporal loops. Finally, we apply our method to the FAUST language, which is a real time language for processing and generating audio streams. We also test some standards FAUST programs.

1 Introduction

La programmation par contraintes [10] offre un ensemble de méthodes efficaces pour modéliser et résoudre des problèmes combinatoires, mais pas seulement. Les méthodes de propagation de contraintes, notamment continues [1], peuvent approximer rapidement des ensembles définis dans un langage générique. Cela permet de les croiser naturellement avec les outils d'interprétation abstraite [4], où l'on souhaite vérifier un programme en calculant des sur-approximations de ses traces, comme cela a été fait dans différents travaux récents [7, 15, 14].

On s'intéresse dans cet article à un cas particulier de programmes, des DSP (Digital Signal Process), qui opèrent sur des flux infinis, et en particulier au langage FAUST (Functional Audio Stream) destiné à la synthèse et au traitement de flux audio [13]. Il est important de vérifier les programmes FAUST car le langage est destiné à des non-informaticiens et utilisé dans des concerts, cas où une erreur est inenvisageable. Plus précisément, nous souhaitons encadrer les valeurs prises par les flux calculés, notamment pour éviter des saturations sur les sons produits [1]. Avoir un encadrement des flux calculés permet aussi de pouvoir estimer la mémoire nécessaire au programme lorsqu'il contient des tables pour stocker certaines valeurs.

Un langage de flux comme FAUST contient des boucles, qui consistent à itérer certaines opérations à chaque pas de temps. Traitant de flux infinis, ces boucles n'ont pas de condition d'arrêt et sont toujours

1. une saturation se produit quand un son sort sort de $[-1, 1]$, la limite des sons numériques. Dans ce cas, il est en général coupé pour les valeurs au delà des bornes, ce qui modifie la forme de l'onde sonore et provoque un effet clairement audible.

infinies, ce qui ne facilite pas l'analyse des flux générés. Le compilateur de FAUST embarque un analyseur dans le domaine abstrait des intervalles [5, 6], mais l'opérateur d'élargissement des intervalles, appliqué aux boucles FAUST, renvoie presque toujours un domaine infini, qui ne permet pas de conclure.

Dans cet article, nous utilisons des méthodes de programmation par contraintes pour vérifier des programmes temps-réel travaillant sur des flux. Pour cela, nous proposons une modélisation en contraintes de diagrammes de blocs, qui décrivent ces programmes. La propagation des contraintes ainsi générées permet de calculer des sur-approximations des ensembles de valeurs prises par les flux au cours du temps. Nous introduisons une contrainte spécifique pour traiter des décalages temporels et proposons un algorithme de propagation efficace pour les problèmes incluant cette contrainte. Enfin, nous appliquons les méthodes développées au langage FAUST, un langage temps-réel de traitement du son. Les contraintes sont calculées dans la librairie de contraintes continues Ibex [2]. Nous présentons des tests préliminaires, avec de très bons temps de résolution, y compris pour des problèmes avec des décalages temporels.

Ce travail fait suite à [3], où une première modélisation partielle (uniquement pour certains composants de FAUST) a été proposée. Cet article présente une modélisation plus générale des langages de flux, ainsi qu'un nouvel algorithme efficace pour les boucles. Des travaux antérieurs comme [9, 8] proposent déjà de résoudre des contraintes sur des flux, mais il s'agit de calculer un automate dont les chemins sont les solutions des contraintes, ce qui diffère assez nettement de notre approche où le problème est d'utiliser la propagation de contraintes pour déterminer des propriétés sur les flux.

Cet article est organisé comme suit : la section 2 introduit la notion de diagrammes de blocs. La section 3 donne la modélisation des diagrammes de blocs en contraintes, dont l'algorithme de résolution est donné dans la section 4. Enfin, la section 5 détaille l'application visée et les tests réalisés.

2 Diagrammes de blocs

Dans cette section, nous présentons la notion de diagrammes de blocs qui sert à définir la sémantique des langages considérés.

2.1 Syntaxe

Un bloc est une fonction, appliquant un opérateur à un ensemble d'entrées ordonnées et produisant une ou plusieurs sorties ordonnées.

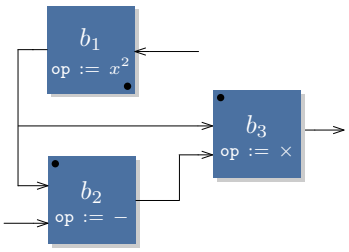


FIGURE 1 – Un diagramme de blocs de $\texttt{DBloc}(\mathbb{R})$

Definition 2.1 (Bloc) Soit E un ensemble non vide. Un bloc sur E est un triplet $b = (\texttt{op}, n, m)$ tel que :
- $n \in \mathbb{N}$ est appelé *nombre d'entrées* ;
- $m \in \mathbb{N}$ est appelé *nombre de sorties* ;
- $\texttt{op} : E^n \to E^m$ est appelé *opérateur*.

De plus, les n *entrées* et les m *sorties* sont ordonnées :
- $[i]b$ correspond à la $i^{\text{ème}}$ entrée $(1 \leq i \leq n)$;
- $b[j]$ correspond à la $j^{\text{ème}}$ sortie $(1 \leq j \leq m)$.

Pour tout bloc, nous écrirons que l'entrée i (resp. sortie j) existe ssi i est un entier compris entre 1 et le nombre d'entrées de ce bloc (resp. j, sorties). Dans la suite, étant donné un ensemble E non vide, $\texttt{Bloc}(E)$ correspond à l'ensemble des blocs sur E.

Definition 2.2 (Connecteur) Soit B un ensemble de blocs. Un connecteur défini sur B est un couple $(b[i], [j]b')$ tel que :
- b et b' sont des blocs de B ;
- la sortie i du bloc b existe ;
- l'entrée j du bloc b' existe.

Definition 2.3 (Diagramme de blocs) Soit E un ensemble non vide. Un diagramme de blocs sur E est un couple $d = (B, C)$ tel que :
- B est un ensemble de blocs sur E ;
- C est un ensemble de connecteurs sur B.

De la même manière que pour les blocs nous utilisons la notation $\texttt{DBloc}(E)$ pour l'ensemble des diagrammes de blocs sur E.

La Figure 1 représente trois blocs manipulant des réels (*i.e.* appartenant à $\texttt{Bloc}(\mathbb{R})$). Il s'agit du bloc b_1 muni de la fonction carrée comme opérateur ; du bloc b_2 avec la soustraction et du bloc b_3 avec la multiplication. L'ordre des entrées et sorties est indiqué par la distance de chacun au point noir. Quant aux connecteurs, ils sont matérialisés par des flèches allant d'un bloc à un autre. Ce diagramme de blocs en possède trois : le connecteur $(b_1[1], [1]b_2)$ allant de b_1 à b_2 ; $(b_1[1], [1]b_3)$ allant de b_1 à b_3 et $(b_2[1], [2]b_3)$ de b_2 à b_3. Le tout (ces trois blocs et ces trois connecteurs) forme un diagramme de blocs manipulant des réels (*i.e.* un diagramme de $\texttt{DBloc}(\mathbb{R})$).

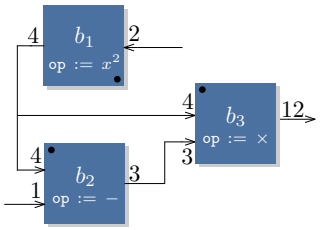


FIGURE 2 – Un diagramme de $\texttt{DBloc}(\mathbb{R})$ interprété

2.2 Sémantique

Après avoir défini ce que sont les diagrammes de blocs, nous définissons leur sémantique.

Definition 2.4 (Interprétation d'un bloc) Soit E un ensemble non vide. Une interprétation I d'un bloc $b = (\texttt{op}, n, m) \in \texttt{Bloc}(E)$ est donnée par l'association de chaque entrée i à un élément de E noté $I([i]b)$ et de chaque sortie j à un élément de E noté $I(b[j])$.

Definition 2.5 (Interprétation d'un diagramme de blocs) Soit E un ensemble non vide. Une interprétation I d'un diagramme de blocs $d = (B, C) \in \texttt{DBloc}(E)$ est telle que I est une interprétation pour chaque bloc b de B.

En reprenant le diagramme de $\texttt{DBloc}(\mathbb{R})$ en exemple dans la Figure 1, donner une interprétation à ce diagramme revient à poser un réel devant chaque entrée et devant chaque sortie des blocs (*e.g.* Figure 2 où nous lisons que $I([1]b_2)$ vaut 4 et que $I(b_3[1])$ vaut 12).

Definition 2.6 (Modèle d'un bloc) Soient E un ensemble non vide, $b = (\texttt{op}, n, m)$ un bloc de $\texttt{Bloc}(E)$ et I une interprétation de b.

$$I \text{ est un modèle de } b \Leftrightarrow$$
$$\texttt{op}(I([0]b), \ldots, I([n]b)) = (I(b[0]), \ldots I(b[m]))$$

Definition 2.7 (Modèle d'un diagramme de blocs) Soient E un ensemble non vide, $d = (B, C)$ un diagramme de blocs de $\texttt{DBloc}(E)$ et I une interprétation de d.

$$I \text{ est un modèle de } d \Leftrightarrow$$
$$\forall b \in B : I \text{ est un modèle de } b$$
$$\text{et } \forall (b[i], [j]b') \in C : I(b[i]) = I([j]b')$$

La Figure 2 propose un modèle au diagramme de blocs de la Figure 1. Remarquons qu'il s'agit d'un modèle parmi l'infinité de ceux que le diagramme admet.

Pour I une interprétation et d un diagramme de blocs, nous noterons $I \models d$ la relation : I est un modèle de d.

2.3 Cas des flux

Jusqu'ici, nous avons construit des diagrammes de blocs sur des ensembles quelconques. Dans la suite, nous prenons comme ensemble de base E un ensemble de *flux*. Pour cela, nous considérons que le temps est discrétisé et nous nous intéressons à des suites de valeurs pouvant changer à chaque pas de temps.

Definition 2.8 (Flux) Soit D un ensemble non vide. Un flux x sur D (appelé aussi un flux de domaine D) est une suite infinie à valeurs dans D.

Dans ce qui suit, nous appelons $x(t)$ la valeur au temps t du flux x ($t \in \mathbb{N}$). Pour un ensemble D non vide, nous notons $\mathbb{S}(D)$ l'ensemble des flux de domaine D. Parmi les blocs sur les flux, nous distinguons deux catégories.

Definition 2.9 (Bloc fonctionnel) Soit D un ensemble non vide et $b = (\texttt{op}, n, m) \in \texttt{Bloc}(\mathbb{S}(D))$.

$$b \text{ est fonctionnel } \Leftrightarrow$$
$$\exists f : D^n \to D^m \text{ telle que}$$
$$\forall s_1, \ldots, s_n, s'_1, \ldots, s'_m \in \mathbb{S}(D)$$
$$\text{avec } \texttt{op}(s_1, \ldots, s_n) = (s'_1, \ldots, s'_m) :$$
$$\forall t \in \mathbb{N} : f(s_1(t), \ldots, s_n(t)) = (s'_1(t), \ldots, s'_m(t))$$

Un bloc fonctionnel peut être calculé à chaque pas de temps indépendamment. Les blocs non fonctionnels ont des dépendances temporelles.

Definition 2.10 (Bloc temporel) Soit D un ensemble non vide et $b = (\texttt{op}, n, m) \in \texttt{Bloc}(\mathbb{S}(D))$.

$$b \text{ est temporel } \Leftrightarrow b \text{ n'est pas fonctionnel}$$

Parmi tous les blocs temporels, nous en identifions un particulier : le bloc *fby* (*followed by*).

Definition 2.11 (Bloc *fby*) Soit D un ensemble non vide. Le bloc $fby = (\texttt{op}, 2, 1)$ de $\texttt{Bloc}(\mathbb{S}(D))$ est défini comme suit :

$$\texttt{op} : \mathbb{S}(D) \times \mathbb{S}(D) \to \mathbb{S}(D)$$
$$(s_1, s_2) = s_3$$

$$\text{avec } s_3(t) = \begin{cases} s_1(0) & \text{si } t = 0 \\ s_2(t-1) & \text{sinon} \end{cases}$$

Remarque Un diagramme de blocs sur des flux peut contenir des cycles (équivalent d'une boucle en programmation). En pratique, si ces boucles ne contiennent pas de *fby*, elles provoquent un calcul infini à chaque pas de temps sauf dans des cas très particuliers (fonctions nulles par exemple). Mais dès lors qu'elles contiennent au moins un bloc *fby* dans le cycle, ce phénomène ne se produit pas.

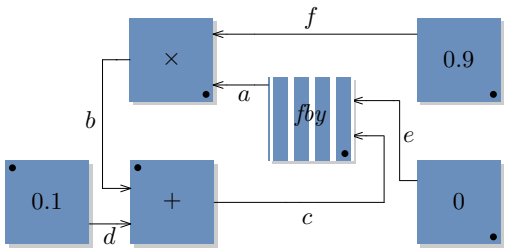


FIGURE 3 – Un diagramme de blocs de $\mathtt{DBloc}(\mathbb{S}(\mathbb{R}))$

t	0	1	2	3	4	...
$a(t)$	0	0.1	0.19	0.27	0.34	...
$b(t)$	0	0.09	0.17	0.24	0.31	...
$c(t)$	0.1	0.19	0.27	0.34	0.41	...
$d(t)$	0.1	0.1	0.1	0.1	0.1	...
$e(t)$	0	0	0	0	0	...
$f(t)$	0.9	0.9	0.9	0.9	0.9	...

TABLE 1 – Modèle du diagramme de la Figure 3 pour les 5 premiers temps.

Dans toute la suite, on supposera que les diagrammes de blocs considérés contiennent toujours au moins un *fby* dans chacun de leurs cycles.

La Figure 3 illustre un diagramme de blocs qui, entre autres, possède un cycle contenant un bloc *fby*.[2] Dans ce diagramme, les blocs 0, 0.1 et 0.9 correspondent aux opérateurs constants (*i.e.* $\forall t \in \mathbb{N} : 0.9(t) = 0.9$). La Table 1 propose un modèle pour les 5 premiers temps. Nous pouvons par exemple y lire le décalage d'un temps entre a et c dû au bloc *fby*.

3 Vérification des flux par contraintes

Les diagrammes de blocs sur des flux peuvent représenter des programmes. Dans ce cas, il est intéressant de connaître certaines propriétés sur les flux générés, qu'il s'agisse des sorties du programme ou des flux intermédiaires (variables locales). Nous présentons ici une modélisation, sous la forme d'un problème de contraintes, du problème suivant : encadrer les valeurs prises par les flux d'un diagramme de blocs.

3.1 Abstraction du temps

La première étape de notre modélisation consiste à abstraire le temps, de façon à considérer les enveloppes des flux générés.

Definition 3.1 (Abstraction temporelle) L'abstraction temporelle d'un flux x est notée $\dot{x}$ et vaut l'ensemble des valeurs prises par ce flux :

$$\dot{x} = \bigcup_{t \in \mathbb{N}} x(t)$$

Il est important de souligner que les abstractions temporelles peuvent contenir un nombre infini d'éléments. De fait, une représentation en extension n'est pas envisageable. Nous nous intéresseront dans la suite à des sur-approximations des flux dans les intervalles.

2. Dans la représentation graphique des diagrammes de blocs sur des flux, nous hachurons les blocs temporels pour les différencier des blocs fonctionnels.

3.2 Abstraction en intervalles

Dans ce qui suit, nous faisons l'hypothèse que D est un ensemble totalement ordonné et nous notons $\mathbb{I}_D$ l'ensemble des intervalles à valeurs dans D. Pour tout intervalle I, nous notons $\lceil I \rceil$ sa borne supérieure et $\lfloor I \rfloor$ sa borne inférieure.

Definition 3.2 (Sur-approximation) Soit D un ensemble non vide muni d'un ordre total $\leq$ et S un ensemble inclus dans D. Une sur-approximation dans les intervalles de S est un intervalle englobant noté $[S]$ appartenant à $\mathbb{I}_{\overline{D}}$.

$$[S] = \{a \leq x \leq b \mid \forall s \in S : a \leq s \leq b\}$$
avec a, b et x dans $\overline{D}$ [3]

Definition 3.3 (Sur-approximation minimale) Soit D un ensemble non vide muni d'un ordre total $\leq$ et S un ensemble inclus dans D. La sur-approximation minimale de S notée $[\![S]\!]$ est la plus petite sur-approximation pour l'inclusion ensembliste.

Proposition 3.1 (Treillis des sur-approximations) Soit D un ensemble non vide muni d'un ordre total $\leq$ et S un ensemble inclus dans D. L'ensemble des sur-approximations $[S]$ muni de l'inclusion comme relation d'ordre forme un treillis dont la borne supérieure est l'union ensembliste et la borne inférieure l'intersection ensembliste. Le plus petit élément correspond à la sur-approximation minimale $[\![S]\!]$ et le plus grand à l'ensemble étendu $\overline{D}$.

Corollaire 3.2 Soit D un ensemble non vide muni d'un ordre total $\leq$. Pour tout S inclu dans D, la sur-approximation minimale existe et est unique.

La démonstration est évidente et laissée au lecteur.

3.3 Modélisation par contraintes

Un diagramme de blocs calcule des sorties en fonction de ses entrées. Pour calculer une sur-approximation des flux générés par un diagramme de

3. $\overline{D}$ appelé ensemble étendu de D correspond à l'union de D et de ses limites (*e.g.* $\overline{\mathbb{R}} = \mathbb{R} \cup \{-\infty, +\infty\}$)

a	:= fby(e, c)	d	:= 0.1
b	:= ×(f, a)	e	:= 0
c	:= +(b, d)	f	:= 0.9

FIGURE 4 – Modèle en contraintes sur les flux

a	:= [fby](e, c)	d	:= [0.1]
b	:= [×](f, a)	e	:= [0]
c	:= [+](b, d)	f	:= [0.9]

FIGURE 5 – Modèle en contraintes sur les intervalles

bloc (B, C) de $\texttt{DBloc}(\mathbb{S}(D))$, nous associons à chaque entrée et à chaque sortie des blocs de B une *variable* de domaine $\mathbb{S}(D)$ et nous considérons l'opérateur de chaque bloc comme une *contrainte* liant les sorties aux entrées. Par exemple, l'opérateur + du diagramme de la Figure 3 calcule c en fonction de b et de d, de sorte que : $c = b + d$. De plus, nous ajoutons une contrainte d'égalité pour les variables qui composent chaque connecteur de C. Dans notre exemple, nous avons unifié les variables ainsi connectées (*e.g.* $[1]+ = *[1] = \text{b}$). La Figure 4 présente un modèle en contraintes sur les flux de l'exemple de la Figure 3.

Nous supposons donc que chaque opérateur du diagramme de blocs peut être traduit dans le langage de contraintes. A partir de ce modèle de contraintes sur les flux, nous considérons le modèle sur les intervalles contenant exactement les mêmes formulations de contraintes que celui sur les flux, en remplaçant chaque fonction par son extension aux intervalles comme défini dans [11]. Dans ce modèle, le domaine des variables n'est plus $\mathbb{S}(D)$ mais $\mathbb{I}_D$.

Definition 3.4 (Extension aux intervalles) Soit D un ensemble non vide et $f : \mathbb{S}(D)^n \to \mathbb{S}(D)^m$ une fonction $(n, m \in \mathbb{N})$. Une extension aux intervalles de f, notée $[f]$ est telle que :

$$[f] : (\mathbb{I}_D)^n \to (\mathbb{I}_D)^m$$
$$X_1, \ldots, X_n \mapsto Y_1, \ldots, Y_m \quad \text{avec}$$
$$Y_i = [\{y_i \mid x_j \in X_j, y_1, \ldots, y_m = f(x_1, \ldots, x_n)\}]$$

La Figure 5 présente un modèle en contraintes sur les intervalles de l'exemple de la Figure 4.

En propageant les contraintes pour le modèle sur les flux, avec la bonne définition des fonctions étendues aux intervalles pour le modèle sur les intervalles, nous pouvons calculer des sur-approximations pour chaque flux. Ainsi, nous avons traduit un diagramme de blocs en un problème de contraintes, dans un langage *ad hoc*, tel qu'en propageant les contraintes, on obtient un encadrement (sur-approximation) des valeurs prises par les flux du diagramme de bloc.

Variable	a	b	c	d	e	f
Solution	$[0; 1]$	$[0; 0.9]$	$[0; 1]$	$[0.1]$	$[0]$	$[0.9]$

TABLE 2 – Solutions au modèle de contraintes sur les intervalles de la Figure 5

Le Tableau 2 présente la solution minimale au modèle de contrainte sur intervalles de la Figure 5.

4 Résolution

Dans cet article, nous résolvons uniquement les diagrammes de blocs utilisant exclusivement le bloc *fby* comme bloc temporel. Cette restriction n'est pas trop forte puisque ce bloc suffit à exprimer tous les diagrammes de blocs avec une mémoire finie.

4.1 Graphe des dépendances

Pour résoudre les problèmes de contraintes issus des diagrammes de blocs, nous utilisons les dépendances entre les flux, devenus des variables du CSP.

Definition 4.1 (Graphe des dépendances) Soient E un ensemble non vide et $d = (B, C)$ un diagramme de blocs de $\texttt{DBloc}(E)$. La relation de dépendances notée $\to$ est la relation sur l'ensemble des entrées et sorties des blocs de B contenant exactement les couples suivants :

$$\forall (b[i], [j]b') \in C : b[i] \to [j]b'$$
$$\forall b = (\texttt{op}, n, m) \in B : [i]b \to b[j]$$
$$(1 \leq i \leq n) \text{ et } (1 \leq j \leq m)$$

Du point de vue de la programmation par contraintes, le graphe dessiné par cette relation est le graphe de dépendance des contraintes (dont les nœuds sont les variables du problème), mais dont les arêtes sont orientées par la dépendance induite par les blocs/-contraintes. La Figure 6 correspond au graphe des dépendances de l'exemple de la Figure 3. [4]

Le calcul des composantes fortement connexes sur ce graphe a un intérêt double. Premièrement, il permet de découper le problème global en sous-problèmes, ces sous-problèmes pouvant être traités dans l'ordre des arêtes qui les relient. Deuxièmement, l'algorithme standard de [16] retourne les composantes dans l'ordre de leurs dépendances dans le graphe orienté, ce que nous utilisons pour choisir l'ordre de la propagation des contraintes (*e.g.* pour la Figure 6, nous traiterons d'abord les composantes du pourtour avant de s'intéresser à celle du centre).

4. Comme dans la représentation graphique des diagrammes de blocs, nous distinguons les blocs temporels des blocs fonctionnels en hachurant cette fois-ci les flèches des blocs temporels.

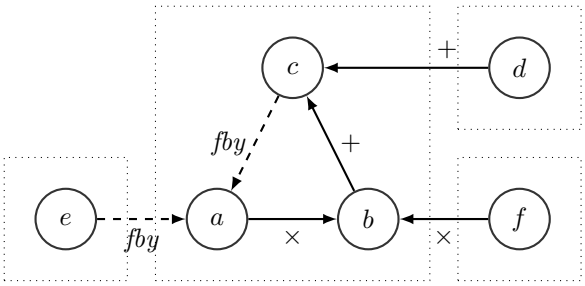


FIGURE 6 – Dépendances inter-connecteurs

Remarque Les cycles du graphe des dépendances sont produits par les boucles du diagramme de blocs.

4.2 Fonction de transfert

L'algorithme 1 propose une méthode de résolution des composantes fortements connexes du graphe des dépendances des contraintes. Nous rappelons l'hypothèse de la section 2.3 : chaque cycle du diagramme de blocs contient au moins un bloc *fby*.

Lorsqu'un bloc *fby* appartient à un cycle, cela implique que son *entrée* dépend de sa *sortie* et donc a fortiori que le flux de *sortie* au temps t dépend de sa valeur au temps précédent (cf. Définition 2.11 introduisant la sémantique du bloc *fby*). Un tel flux de sortie x peut alors se définir récursivement comme suit :

$$\forall t \in \mathbb{N} : x(t+1) = f(x(t))$$
$$\text{et } x(0) \text{ vaut la valeur au temps 0 de } [0]fby$$

Pour trouver cette fonction, appelée *fonction de transfert de la boucle*, nous partons du diagramme de blocs générant la composante fortement connexe considérée et nous retirons les connecteurs amenant à des blocs *fby* (lignes 10 à 17 de l'Algorithme 1). De ce nouveau diagramme, un parcours en profondeur du dual de son graphe des dépendances donne exactement la fonction de transfert de la boucle en notation polonaise inversée. [5]

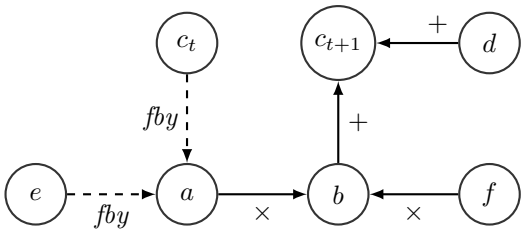


FIGURE 7 – Graphe d'une fonction de transfert

5. Il est nécessaire de respecter l'ordre des entrées imposées par le diagramme de blocs pour la selection des fils (tous les opérateurs ne sont pas forcément commutatifs).

```
1:  fonction RÉSOLUTIONCOMPOSANTE(
2:          C : Set<Contrainte>,
3:          domaine : Map<Variable, I_D>)
4:      retourne Map<Variable, I_D>
5:
6:      // Retrait des connexions vers fby
7:      entrees : List<Variable>
8:      sorties : List<Variable>
9:      compteur ← 0 : entier
10:     pour chaque contrainte b[i] = [j]b' dans C faire
11:         si b' égale fby alors
12:             C ← C \ {b[i] = [j]b'}
13:             entrees.AJOUTER([j]b')
14:             sorties.AJOUTER(b[i])
15:             compteur ← compteur + 1
16:         fin si
17:     fin pour chaque
18:
19:     // Fonction de transfert
20:     f : (I_D)^compteur → (I_D)^compteur
21:     f ← FONCTIONTRANSFERT(C, domaine,
22:                           entrees, sorties)
23:     approximation : Liste<I_D>
24:     approximation ← SURAPPROXIMATION(f)
25:
26:     //  MAJ domaines des variables
27:     pour chaque i de 1 à compteur faire
28:         domaine[entrees[i]] ← approximation[i]
29:         domaine[sorties[i]] ← approximation[i]
30:     fin pour chaque
31:     PROPAGER(C, domaine, entrees ∪ sorties)
32:
33:     retourne domaine
34: fin fonction
```

Algorithm 1 – Résolution des composantes fortement connexes

La Figure 7 représente le graphe des dépendances du diagrammes de blocs de la composante centrale de la Figure 6 où le connecteur menant au bloc *fby* a été supprimé. Par lecture de ce graphe, la fonction de transfert de la boucle en notation infixée est $c_{t+1} = f \times fby(e, c_t) + d$.

Ensuite, en utilisant les extensions aux intervalles des opérateurs des blocs (cf. Définition 3.4) et en remplaçant les variables n'appartenant pas à la composante connexe par leur domaine, nous obtenons la fonctions de transfert sur les intervalles de la boucle (dans l'Algorithme 1, cette fonction est récupérée via l'appel à FONCTIONTRANSFERT de la ligne 21). Dans notre exemple, elle vaut $F(X) = [0.9] \times [fby]([0], X) + [0.1]$ avec $X \in \mathbb{I}_{\mathbb{R}}$.

La fonction de transfert sur les intervalles et la clé de voûte pour la résolution de notre problème de part la Proposition 4.1.

```
fonction surApproximation(f : D^n → D^n)
    retourne Liste<I_D>

    etat : Liste<String>
    courant, min, max, image : Liste<I_D>
    pour chaque i de 1 à n faire
        etat[i] ← "Widening"
        courant[i] ← ∅
        min[i] ← ∅
        max[i] ← D
    fin pour chaque

    tant que continuerBoucle(courant, min, max)
    faire
        image ← f(courant)
        pour chaque i de 1 à n faire
            si etat[i] = "Widening" et
                    image[i] ⊆ courant[i] alors
                etat[i] ← "Narrowing"
                max[i] ← courant[i]
            sinon si etat[i] = "Narrowing" et
                    image[i] ⊈ courant[i] alors
                etat[i] ← "Widening"
                min[i] ← courant[i]
            fin si

            courant[i] = courant[i] ∪ image[i]
            si etat[i] = "Widening" alors
                courant[i] ←
          selectIntervalBetween(courant[i], max[i])
            sinon
                courant[i] ←
          selectIntervalBetween(min[i], courant[i])
            fin si
        fin pour chaque
    fin tant que
    retourne max
fin fonction
```

Algorithm 2 – Fonction de recherche aléatoire

Proposition 4.1 Soient D un ensemble non vide et $d = (B, C)$ un diagramme de blocs de $\texttt{DBloc}(\mathbb{S}(D))$. Pour tout bloc *fby* $b \in B$ appartenant à une boucle de d dont la fonction de transfert sur les intervalles est F, si un intervalle S de $\mathbb{I}_D$ est stable par F, alors S est une sur-approximation valide des flux en sortie de b.

L'Algorithme 2 propose une méthode héritée de l'interprétation abstraite pour déterminer des ensembles stables par la fonction de transfert sur les intervalles. Cette fonction ne prétend pas retourner systématiquement la sur-approximation minimale mais simplement une sur-approximation acceptable.

Pour chaque variable d'entrée i de la fonction de transfert nous associons un espace de recherche délimité par les intervalles $min[i]$ et $max[i]$ qui sont respectivement initialisés à l'ensemble vide et à l'ensemble étendu du domaine considéré. A chaque tour de boucle, $courant[i]$ est sélectionné tel qu'il contienne $min[i]$ et soit inclus dans $max[i]$ (*i.e.* $min[i] \subseteq courant[i] \subseteq max[i]$ est un invariant de boucle).

```
1:  fonction réductionDomaines((B, C) : DBloc(S(D)))
2:      retourne Map<Variable, I_D>
3:
4:      // Initialisation
5:      domaine : Map<Variable, I_D>
6:      b : Bloc(S(D))
7:      pour chaque b ∈ B faire
8:          pour chaque entrée [i]b de b faire
9:              domaine[[i]b] ← [D]
10:         pour chaque sortie b[j] de b faire
11:             domaine[b[j]] ← [D]
12:     fin pour chaque
13:
14:     // Pré-traitement
15:     graphe : GrapheOrienté<Contrainte>
16:     graphe ← grapheDépendances(B, C)
17:     composante : Set<Connecteur>
18:     composantes : Pile<Set<Contrainte>>
19:     composantes ← composantesFortConnexes(
20:                                         graphe)
21:
22:     // Corps de la fonction
23:     tant que composantes non vide faire
24:         composante ← dépiler(composantes)
25:         domaine ← résolutionComposante(
26:                             composante, domaine)
27:     fin tant que
28:
29:     retourne domaine
30: fin fonction
```

Algorithm 3 – Algorithme global

Nous associons également à chaque variable d'entrée i une variable d'état $state[i]$ qui a pour valeur "Widening" ou "Narrowing". Il y a passage du *widening* au *narrowing* lorsque l'intervalle $courant[i]$ est stable par la fonction de transfert et passage du *narrowing* au *widening* dans le cas contraire.

Concernant la fonction selectIntervalBetween, la méthode de sélection est libre mais nous expliciterons dans nos expérimentations les choix réalisés (*i.e.* aléatoire, dichotomiques, etc.). De même pour la fonction continuerBoucle qui généralement possèdera un compteur pour limiter le nombre d'itérations. Dans les expérimentations, nous appellons selectIntervalBetween l'heuristique de selections et continuerBoucle l'heuristique de fin de boucle.

Finalement, l'Algorithme 3 correspond à la résolution complète du problème.

5 Faust

Nous appliquons ici les outils développés au langage FAUST, un langage développé par le Grame [6].

5.1 Présentation

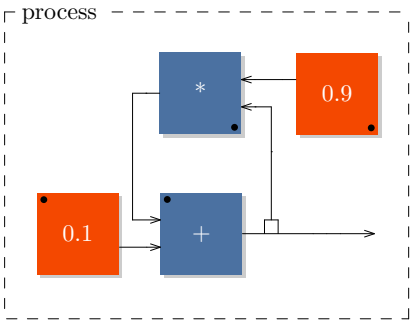


FIGURE 8 – Un diagramme de blocs généré par FAUST

FAUST (Functional Audio Stream) est un langage fonctionnel pour la synthèse et le traitement temps-réel de flux audio. L'une de ses particularités est d'avoir une sémantique bien définie, dans un langage de diagrammes de blocs [12]. Le compilateur FAUST peut générer pour chaque programme le diagramme de blocs qui en est la sémantique.

Un programme FAUST est un processeur de signal numérique (DSP : Digital Signal Processor). Le concept des DSP n'est pas propre à FAUST, ni au traitement audio : il est utilisé dans toutes les applications qui reçoivent et traitent des signaux numériques : modems, appareils multimédia, récepteurs GPS, systèmes vidéo, ... FAUST propose un langage complètement spécifié pour écrire les DSPs.

```
1 A = + (0.1);
2 B = * (0.9);
3 process = A ~ B;
```

FIGURE 9 – Exemple de DSP pour FAUST

Chaque DSP ainsi écrit passe dans le compilateur FAUST et produit un programme C++ optimisé, qui est ensuite compilé pour une plateforme visée. Ce processus de compilation en deux étapes permet à un même programme FAUST de pouvoir être compilé pour des téléphones, navigateurs web, dispositifs de concerts, etc. Nous nous intéressons ici à la première étape de compilation, qui permet de produire le code C++. En particulier, la première phase d'analyse syntaxique ré-écrit le programme sous une forme normale unique, dans laquelle sont notamment éliminées les formules redondantes (par exemple $x - x$). Cette forme normale nous permet de travailler sur des expressions avec peu d'occurences multiples pour les variables. Une fois cette étape réalisée, le compilateur calcule le diagramme de blocs résultant du programme. Le programme de la Figure 9 donne ainsi le diagramme présenté Figure 8 (copie d'écran de la sortie du compilateur FAUST).

6. `http://faust.grame.fr`

Diagramme FAUST Sémantique FAUST

$$\begin{cases} c(0) & = a(0) \\ c(t+1) & = b(t), \text{ si } t > 0 \end{cases}$$

Diagramme de blocs équivalent

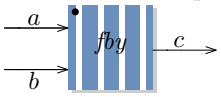
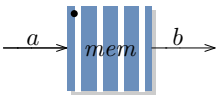


FIGURE 10 – Bloc *prefix* de FAUST

Diagramme FAUST Sémantique FAUST



$$\begin{cases} b(0) & = 0 \\ b(t+1) & = a(t), \text{ si } t > 0 \end{cases}$$

Diagramme de blocs équivalent

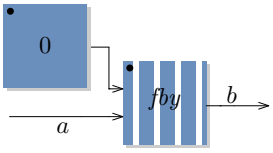


FIGURE 11 – Bloc *mem* de FAUST

5.2 Modélisation

Notre algorithme ne peut résoudre que des diagrammes de blocs de flux utilisant exclusivement le bloc *fby* comme bloc temporel. Or, il existe dans le langage des diagrammes de blocs de FAUST trois blocs temporels : *prefix*, *mem* et *delay*. Les figures 10, 11 et 12 rappellent la sémantique de ces blocs et proposent des diagrammes de blocs équivalents (possédant exactement les mêmes modèles) utilisant le bloc *fby*.

En utilisant l'équivalence proposée du bloc *delay* à la Figure 12, la complexité spatiale du pré-traitement peut devenir importante. Cependant, pour notre problème de sur-approximation des flux, le fait que ces derniers soient infinis alors que le nombre de *fby* ajoutés ne l'est pas, nous permet d'écrire un seul bloc *fby* pour avoir des solutions équivalentes (*i.e.* de par l'abstraction temporelle de la Définition 3.1). Dès lors, nous

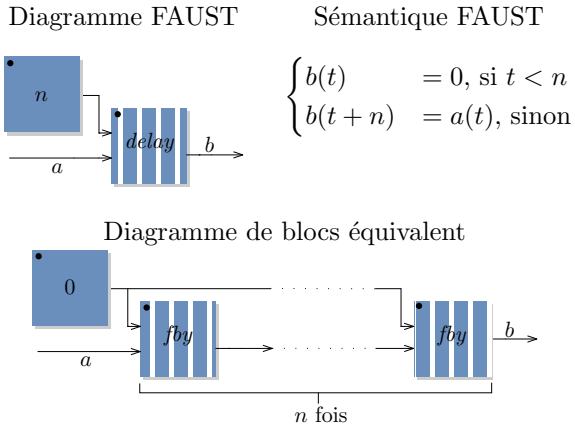


FIGURE 12 – Bloc *delay* de FAUST

utilisons la réécriture du bloc *mem* de la Figure 11 pour le bloc *delay*.

Une extension aux intervalles de l'opérateur *fby* en accord avec sa sémantique des équivalences proposées pour les blocs temporels est la suivante : $[\![fby]\!](A, B) = [A \cup B]$. Le reste du langage FAUST est principalement composé des fonctions arithmétiques usuelles, de fonctions importées de C++ (dont *sin*; *cos*, *exp*...) et d'opérateurs de comparaison. [7] Tous ces opérateurs s'étendent aux intervalles à la façon de [11], et la traduction par contraintes se fait naturellement dans un langage usuel de contraintes continues, excepté les opérateurs bit à bit.

5.3 Expériences

Avant de passer aux expérimentations, nous introduisons la notion de distance sur les intervalles afin de pouvoir mesurer la qualité des solutions retournées par notre algorithme.

Definition 5.1 (Distance étendue) Soit D un ensemble non-vide. La fonction de distance étendue $\overline{d}$ pour deux éléments x et y de $\overline{D}$ est donnée par :

$$\overline{d}(x, y) = 0 \qquad \text{si } x = y,$$
$$\overline{d}(x, y) = |x - y| \qquad \text{sinon.}$$

Cette définition permet d'évaluer la distance avec les infinis. Par exemple $\overline{d}(-\infty, 0)$ vaut $-\infty$ et $\overline{d}(+\infty, +\infty)$ retourne 0.

Definition 5.2 (Distance d'intervalles) Soit D un ensemble non-vide. La fonction de distance pour deux intervalles X et Y de $\mathbb{I}_{\overline{D}}$ est donnée par :

$$d(X, Y) = \overline{d}(\lfloor X \rfloor, \lfloor Y \rfloor) + \overline{d}(\lceil X \rceil, \lceil Y \rceil) \qquad \text{si } X \subseteq Y,$$
$$d(X, Y) = +\infty \qquad \text{sinon.}$$

7. voir `http://faust.grame.fr/index.php/documentation/references` pour une présentation complète.

Proposition 5.1 Soit un ensemble S inclus dans E, une sur-approximation $[S]$ est minimale ssi la distance entre $[S]$ et $[\![S]\!]$ est nulle.

Une bonne approximation est donc une approximation ayant la distance la plus petite possible à la sur-approximation minimale.

Nous avons testé notre approche sur cinq programmes FAUST : EXEMPLE-PAPIER correspond à l'exemple qui a été déroulé tout au long de ce papier ; COMPTEUR correspond à un flux commençant à 0 s'incrémentant de 1 à chaque temps (dont la borne supérieure est infinie) ; BRUIT-BLANC correspond à la génération d'un bruit (*i.e.* suite de valeurs aléatoires) ; SON-SINUS correspond à la génération d'un son pur et SON-ECHO à la création d'un écho simple sur une entrée audio.

Pour chaque exécution, l'heuristique de fin de boucle autorise un maximum de 5 branches de recherches avec pour chaque branche une limite de 500 selections d'intervalles (*i.e.* 500 tours de boucles). L'heuristique de selection des intervalles alterne entre trois : choisir aléatoirement une nouvelle borne inférieure entre $\lfloor min \rfloor$ et $\lfloor max \rfloor$, choisir aléatoirement une nouvelle borne supérieure entre $\lceil min \rceil$ et $\lceil max \rceil$, ou alors réaliser les deux en même temps.

La Figure 13 présente les résultats obtenus en appliquant notre algorithme de résolution pour une sélection de 5 programmes fondamentaux en FAUST. L'algorithme a été exécuté 10 fois pour chaque problème. Nous présentons à chaque fois les moyennes de temps d'exécution, de distance à la sur-approximation minimale et du nombre d'itérations.

Ces tests préliminaires, qui seront complétés par d'autres programmes de la librairie standard de FAUST, montrent que notre algorithme retourne toujours la plus petite sur-approximation des flux pour ce jeu d'essais. Le temps de calcul, qui doit être très rapide pour que notre outil puisse être embarqué dans le compilateur FAUST, est encourageant : le temps de calcul est de l'ordre d'une milli-seconde, et varie peu.

5.4 Conclusion

Nous avons présenté une modélisation en contraintes d'un problème de vérification pour des programmes temps-réel. Le but à terme est d'implémenter l'outil dans le compilateur FAUST, ce qui suppose que les temps de calcul soient très petits (négligeables devant le temps de compilation). Les premiers tests réalisés sont encourageant, et nous engagent à poursuivre le développement de l'outil. Dans un futur proche, ce travail sera implémenté de façon à pouvoir être intégré dans le compilateur FAUST, et les méthodes seront

Programme	#blocs (dont *fby*)	Temps (10^{-3}s)			Distance			#itérations		
		Min	Avg	Max	Min	Avg	Max	Min	Avg	Max
EXEMPLE-PAPIER	6 (1)	0,828	0,8727	1,016	0	0	0	851	911,2	1176
COMPTEUR	4 (1)	1,174	1,5072	1,887	0	0	0	1646	2124	2500
BRUIT-BLANC	8 (1)	0,784	0,9888	1,568	0	0	0	1007	1007	1007
SON-SINUS	5 (1)	0,326	0,3906	0,899	0	0	0	13	13	13
SON-ECHO	7 (1)	0,134	0,1376	0,14	0	0	0	1	1	1

FIGURE 13 – Résultats pour la résolution du jeu d'essais

testées intensivement pour améliorer le réglage des paramètres.

Par ailleurs, ce travail montre l'intérêt de l'utilisation de techniques de programmation par contraintes dans des cadres exotiques, ici, sur des variables flux. L'algorithme développé étant générique, il pourrait avoir des utilisations à d'autres langages de signaux, ce que nous examinerons par la suite.

Références

[1] Frédéric Benhamou and William J. Older. Applying interval arithmetic to real, integer and boolean constraints. *Journal of Logic Programming*, 32(1) :1–24, 1997.

[2] Gilles Chabert and Luc Jaulin. Contractor programming. *Artificial Intelligence*, 173(11) :1079–1100, 2009.

[3] Yann Orlarey Charlotte Truchet, Julie Laniau. Avoiding saturation in sound processes with constraint programming. In *Constraint Programming meets Verification Workshop at CP'14*, 2014.

[4] P. Cousot and R. Cousot. Static determination of dynamic properties of programs. In *Proceedings of the Second International Symposium on Programming*, pages 106–130. Dunod, Paris, France, 1976.

[5] P. Cousot and R. Cousot. Abstract interpretation : a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles, California, 1977.

[6] P. Cousot and R. Cousot. Abstract interpretation frameworks. *Journal of Logic and Computation*, 2(4) :511–547, 1992.

[7] Tristan Denmat, Arnaud Gotlieb, and Mireille Ducassé. An abstract interpretation based combinator for modeling while loops in constraint programming. In *In Proceedings of Principles and Practices of Constraint Programming (CP'07), Springer Verlag, LNCS 4741*, pages 241–255, 2007.

[8] Arnaud Lallouet, Yat Chiu Law, Jimmy HM Lee, and Charles FK Siu. Constraint programming on infinite data streams. In *Proceedings of the Twenty-Second international joint conference on Artificial Intelligence-Volume Volume One*, pages 597–604. AAAI Press, 2011.

[9] JasperC.H. Lee and JimmyH.M. Lee. Towards practical infinite stream constraint programming : Applications and implementation. In Barry O'Sullivan, editor, *Principles and Practice of Constraint Programming*, volume 8656 of *Lecture Notes in Computer Science*, pages 449–464. Springer International Publishing, 2014.

[10] Ugo Montanari. Networks of constraints : Fundamental properties and applications to picture processing. *Information Science*, 7(2) :95–132, 1974.

[11] Ramon Edgar Moore. *Interval Analysis*. Prentice-Hall, Englewood Cliffs N. J., 1966.

[12] Y. Orlarey, D. Fober, and S. Letz. An algebra for block diagram languages. In *International Computer Music Conference*, 2002.

[13] Y. Orlarey, D. Fober, and S. Letz. Syntactical and semantical aspects of faust. *Soft Computing*, 8(9) :623–632, 2004.

[14] Marie Pelleau, Charlotte Truchet, and Frédéric Benhamou. The octagon abstract domain for continuous constraints. *Constraints*, 19(3) :309–337, 2014.

[15] Olivier Ponsini, Claude Michel, and Michel Rueher. Refining abstract interpretation based value analysis with constraint programming techniques. In *Principles and Practice of Constraint Programming - 18th International Conference, CP 2012, Québec City, QC, Canada, October 8-12, 2012. Proceedings*, pages 593–607, 2012.

[16] Micha Sharir. A strong-connectivity algorithm and its applications in data flow analysis. *Computers & Mathematics with Applications*, 7(1) :67–72, 1981.