Analyse de robustesse de systèmes avioniques

Y. Ait Ameur[1], F. Boniol[2], S. Pairault[2] & V. Wiels[2]

1: LISI-ENSMA,
BP 109, Site du Futuroscope
86960 Futuroscope cedex, France
`yamine@ensma.fr`
2: ONERA-CERT/DTIM,
2 avenue E. Belin, BP 4025
31055 Toulouse cedex, France
`{boniol,wiels}@cert.fr, pairault@supaero.fr`

1. Introduction

Les systèmes intégrés constituent une classe de systèmes d'importance grandissante, que ce soit dans les domaines des télécommunications, du nucléaire ou des transports. Parmi ces systèmes, on peut citer les systèmes avioniques embarqués à bord d'aéronefs civils ou militaires tels que les Airbus A320 à A380, le Boeing B777, ou les avions d'arme du type Rafale ou Eurofighter. Au sein de ces systèmes, les fonctions de pilotage et d'asservissement des organes physiques de l'aéronefs, souvent appelés “système de commande de vol”, occupent une place centrale. Le rôle de ce système est, en réponse à une consigne de pilotage exprimée par le pilote (humain ou automatique) et en fonction de la position des gouvernes et de la situation (attitude, altitude, vitesse . . .), de calculer et appliquer des ordres de braquage de gouverne. Depuis le développement des architectures “fly-by-wire”, ces fonctions sont vitales, et donc critiques.

Se pose alors la question centrale de la tolérance aux défaillances du système informatique implantant les lois de pilotage et d'asservissement, notamment face aux erreurs ou aux imprécisions des capteurs. En particulier, la position des gouvernes, l'altitude et l'attitude de l'avion sont mesurées par des instruments physiques inévitablement imprécis, délivrant des valeurs souvent entâchées d'erreur par rapport à la valeur réelle du paramètre mesuré. La question sous-jacente est alors l'évaluation de la robustesse du système informatique des commandes de vol face à de telles imprécisions, et en particulier l'évaluation de l'impact en terme d'erreur sur les ordres de gouverne produits par le système. Compte tenu du caractère critique des systèmes considérés, il est impératif de démontrer que cette erreur est bornée, d'en déterminer la valeur maximale, et de démontrer que cette erreur est acceptable en regard de la dynamique de l'aéronef et de ses gouvernes. Cela revient à étudier une facette non fonctionnelle du système, à savoir la propagation en son sein des erreurs numériques.

Le langage adopté par Airbus pour la programmation du système des commandes de vol est le langage fonctionnel synchrone LUSTRE [CHPR91] implémenté par l'atelier SCADE. La question résultante, examinée dans cet article, est alors l'analyse du comportement d'un programme LUSTRE/SCADE en présence de flots d'entrée numériques imprécis. Une méthode possible pour cette analyse serait d'utiliser le model checking (technique des observateurs [HFP93]). Il faudrait considérer une version exacte du système et une version imprécise et comparer les sorties des deux versions du système (la propriété à vérifier serait que la différence entre les sorties est inférieure à un seuil donné). Cette méthode est utilisée dans [HCW02] pour effectuer une analyse de déviation à l'aide de model checking. Cependant, les auteurs de cet article ne considèrent que de petits exemples comportant uniquement des entiers et des booléens. Les systèmes avioniques que nous souhaitons analyser sont des systèmes comportant de nombreuses variables réelles associées à des comportements dynamiques. Le model checking ne permet pas pour l'instant de vérifier des propriétés sur des systèmes comportant des calculs non linéaires sur les réels et des calculs temporels. Par conséquent, nous avons choisi une approche qui repose sur la technique d'interprétation abstraite, consistant à transformer le programme LUSTRE initial (le système embarqué) en une représentation abstraite, à base d'intervalles, permettant de calculer la propagation d'erreurs au sein du système.

Après une rapide présentation du langage LUSTRE/SCADE considéré dans cette étude (le domaine concret) (section 2), ainsi qu'un bref rappel sur la technique de l'interprétation abstraite (section 3), la section 4 présente l'interprétation abstraite proposée pour la détermination des erreurs propagées sur les flots de sortie d'un nœud LUSTRE.

2. L'approche Lustre/SCADE

Dans cette section, nous présentons brièvement l'environnement SCADE et le langage LUSTRE puis nous identifions les types de variables qui peuvent être perturbées par des erreurs de mesures.

SCADE. L'environnement SCADE[1] a été défini pour assister le développement de systèmes critiques. Cet environnement est constitué de plusieurs outils, notamment un éditeur graphique, un simulateur, un model checker permettant de vérifier certaines propriétés, un générateur de code qui traduit de façon automatique les spécifications en code C.

Le langage SCADE est un langage de spécification graphique à flots de données qui peut être traduit en LUSTRE [CHPR91, HLR92]. Dans la suite de l'article, nous manipulerons uniquement des programmes LUSTRE.

LUSTRE. LUSTRE est un langage à flots de données synchrone. Un système est décrit par l'interconnexion de plusieurs nœuds. Un nœud calcule la valeur de ses sorties en fonction de la valeur de ses entrées. Un nœud est défini par un ensemble d'équations et d'assertions. Une équation `VAR=EXPR` spécifie que la variable `VAR` est égale à `EXPR` à tous les instants. Une assertion `assert BOOLEXPR` signifie que l'expression booléenne `BOOLEXPR` est supposée vraie pendant toute l'exécution du programme.

Chaque variable ou expression représente la séquence infinie de valeurs qu'elle prend pendant toute l'exécution du système et les opérateurs LUSTRE opèrent globalement sur ces séquences. Par exemple, `VAR=EXPR` signifie que les séquences de valeurs associées à `VAR` et `EXPR` sont identiques.

Les expressions sont faites d'identificateurs de variables, de constantes, des opérateurs arithmétiques, booléens et conditionnels classiques et de deux opérateurs temporels spécifiques : previous `pre` et followed-by `->`.

- Si `E` est une expression représentant la séquence $(e_0, e_1, e_2, ...)$, alors `pre(E)` représente la séquence $(nil, e_0, e_1, ...)$ où *nil* est la valeur indéfinie.
- Si `E` et `F` sont deux expressions de même type représentant respectivement les séquences $(e_0, e_1, e_2, ...)$ et $(f_0, f_1, f_2, ...)$, alors `E -> F` est une expression représentant la séquence $(e_0, f_1, f_2, ...)$.

Le langage LUSTRE comprend aussi un calcul d'horloges que nous ne présenterons pas ici. Dans la suite, nous considèrerons un LUSTRE restreint : pas d'horloges, pas d'assertions. De plus, nous considèrerons que seules les variables réelles et booléennes peuvent être perturbées. Dans les programmes Airbus analysés, les entiers sont toujours soit des constantes, soit des variables du système que les erreurs de mesure des capteurs ne concernent pas.

[1] http ://www.esterel-technologies.com

Exemple. Considérons une version (très) simplifiée d'une fonction d'asservissement d'une gouverne, décrite en LUSTRE par le nœud `asserv` ci-dessous :

```
node asserv (Xgouv, Sensor : real) returns (Consigne, Courant : real);
var x, y : real;
let
   Courant = Val_Lim(x*(-3.2), -8.0, 8.0);
   Consigne = (y*0.75) + ((0.0->pre(y))*0.5) - ((0.0 -> pre(0.0 -> pre(y)))*0.25)
   x = Consigne - (Sensor*130.);
   y = Rate_Lim(Xgouv, 2.0, 0.01) ;
tel.
```

La fonction `asserv` est composée, en plus du nœud principal `asserv`, de deux opérateurs de la bibliothèque SCADE `val_Lim` et `Rate_Lim` dont la définition est également donnée en LUSTRE.

Le premier opérateur `val_Lim`, défini ci-dessous, opère comme un limiteur de flot `E1` écrétant celui-ci au dessus ou en dessous d'une valeur maximale `Max` et minimale `Min`. `S1` est le flot écrété renvoyé en sortie.

```
node val_Lim(E1 : real; Min, Max : real) returns (S1 : real) ;
var xmin:real, xmax : real;
let
   (xmax , xmin) = if (Max >= Min) then (Max , Min) else (Min , Max) ;
   S1 = if (xmax <= E1) then xmax
        else (if (E1 > xmin) then E1 else xmin) ;
tel.
```

Le second opérateur `Rate_Lim` agit comme un limiteur de pente écrétant la valeur d'un flot d'entrée `E1` en fonction de sa valeur précédente et selon des pentes maximale et minimale autorisées calculées en fonction des flots `Lim` et `fixer_duree_cycle` en entrée.

```
node Rate_Lim(E1 : real; Lim, fixer_duree_cycle : real;) returns (S1 : real) ;
var x, y : real;
let
   x = E1 - (0.0 -> pre(S1)) ;
   y = Lim * fixer_duree_cycle ;
   S1 = if (x < (-y)) then ((0.0 -> pre(S1)) - y)
        else (if (x > y) then ((0.0 -> pre(S1)) + y) else E1) ;
tel.
```

D'un point de vue fonctionnel, le nœud `asserv` calcule deux flots réels `Consigne`, i.e. la valeur théorique du courant à envoyer à la servo commande de la gouverne, et `Courant`, i.e. la valeur filtrée du courant à envoyer effectivement, en fonction de la position de la gouverne (`Sensor`) et de la position demandée (`Xgouv`). Se pose alors la question de la détermination des erreurs sur `Consigne` et `Courant` résultant de valeurs erronées reçues sur `Xgouv` et `Sensor`.

3. Interprétation abstraite

Les propriétés de précision que nous étudions constituent un cas particulier de propriétés non fonctionnelles. Notre démarche consiste à prendre en compte ce type de propriétés afin d'augmenter la qualité des programmes. Cette démarche consiste à construire un évaluateur de ces propriétés. Ces dernières sont évaluées sur un autre domaine (le domaine représentant les propriétés) ou domaine non standard. Les valeurs de propriétés sont alors utilisées pour améliorer la qualité des programmes. L'interprétation abstraite, technique d'analyse de programmes, est utilisée pour établir formellement le lien entre l'évaluation des propriétés qui constitue une sémantique non standard et la sémantique standard des programmes que nous analysons.

La construction de l'évaluateur sur le domaine non standard peut se faire de deux façons. La première consiste à coder cet évaluateur dans le même langage que celui utilisé pour représenter les programmes. La seconde approche consiste à décrire un évaluateur dans un autre langage et à réutiliser les résultats de l'évaluation dans le langage source. Dans les deux cas, l'interprétation abstraite est utilisée comme technique garantissant la correction des évaluations de propriétés vis-à-vis du programme source analysé.

Dans les travaux que nous présentons dans cet article, nous nous situons dans le second cas. En effet, les spécifications et/ou programmes sont décrits en LUSTRE (sémantique standard) et sont analysés, à l'aide d'un analyseur développé en JAVA (sémantique non standard). L'interprétation abstraite assure la correction entre les deux sémantiques. Les résultats des évaluations de propriétés non fonctionnelles obtenues à partir de l'évaluateur JAVA sont utilisés pour améliorer les spécifications LUSTRE.

L'interprétation abstraite est une technique d'analyse de programmes introduite par [CC77]. Elle constitue une technique formelle de développement et de preuve de programmes, et a été appliquée avec succès à plusieurs styles de programmation (impérative, logique et fonctionnelle)[AH87][BHA86][Mel87]. L'interprétation abstraite consiste à exécuter, sur un domaine abstrait, une version abstraite du programme à analyser, qui en soit une approximation correcte. Le domaine abstrait sur lequel s'appuie cette transformation se caractérise par la formalisation explicite des propriétés que l'on désire analyser, et qui sont implicitement présentes dans le programme original. Ce domaine doit alors remplir certaines conditions afin de garantir la correction du processus de transformation du programme original. Ce processus est appelé abstraction. Le processus inverse, faisant correspondre à une valeur abstraite donnée l'ensemble des éléments représentés par cette valeur, est la concrétisation.

Principe

- L'abstraction associe donc à chaque élément du domaine concret, un élément abstrait représentant ses propriétés.

- La concrétisation caractérise l'ensemble des éléments du domaine concret qui satisfont les propriétés d'un élément donné du domaine abstrait.

- Enfin, un critère de correction est associé à une interprétation abstraite.

Ainsi, une interprétation abstraite est définie par un quadruplet $(D_\perp, A, abs, conc)$ où :
– $D_\perp$ est le domaine concret ou domaine standard ;
– A est le domaine abstrait, un treillis complet ;
– abs est la fonction d'abstraction $(abs : D_\perp \longrightarrow A)$ où l'élément $a = abs(d)$ est l'élément de A constituant une approximation de l'élément d de $D_\perp$. Il définit la propriété a associée à l'élément d ;
– $conc$ est la fonction de concrétisation $(conc : A \longrightarrow \mathcal{P}(D_\perp))$ où $conc(a)$ représente l'ensemble des éléments de $D_\perp$ qui vérifient les propriétés décrites par a. On définit $conc(a) = \{d \in D_\perp \mid abs(d) = a\})$.

Il est alors possible de traduire les fonctions f définies sur $D_\perp$ en fonctions abstraites f_a définies sur A, afin d'analyser les propriétés de leurs images, en comparaison de celles de leurs antécédents.

Domaines abstrait et concret Le domaine concret, découlant du domaine original D est muni d'un plus petit élément $\bot_D$ et d'une relation d'ordre partiel. L'élément $\bot_D$ représente l'élément indéfini ; l'ordre partiel défini sur $D_\bot$, noté $\preceq$, vérifie $\forall\ d \in D_\bot,\ \bot_D \preceq d$ et $d \preceq d$

Le domaine abstrait A est défini à partir du domaine standard $D_\bot$, et l'étend de façon à tenir compte d'informations sur le comportement des éléments de $D_\bot$. Ce domaine est un treillis complet, c'est-à-dire un ensemble partiellement ordonné contenant un plus petit élément, $\bot_A$, un plus grand élément, $\top_A$, et deux opérateurs binaires, $\sqcup$ et $\sqcap$.

Les opérateurs binaires, *join* et *meet*, étendent les notions classiques d'union et d'intersection de la théorie des ensembles, et permettent d'associer à deux éléments pris dans A, respectivement un majorant et un minorant de A, en accord avec l'ordre partiel défini ci-dessus.

Critère de correction La définition de la fonction d'abstraction est étendue aux ensembles, de sorte que $Abs : \mathcal{P}(D_\bot) \to A$, tel que $Abs(\delta) = \bigsqcup_{d\in\delta} abs(d)$.

L'interprétation abstraite est alors dite correcte lorsque $Id \subseteq conc \circ abs$ et $Abs \circ conc = Id$, c'est-à-dire lorsque $\forall\ d \in D_\bot,\ d \in conc(abs(d))$ et $\forall\ a \in A,\ Abs(conc(a)) = a$.

De plus, si f est une fonction définie sur $D_\bot$ et f_a la version abstraite de f sur le domaine abstrait A, alors f_a est une approximation correcte de f si et seulement si $f \subseteq conc \circ f_a \circ abs$, soit $\forall\ d \in D_\bot,\ f(d) \in conc\,(f_a\,(abs(d)))$

Ces critères de correction permettent de s'assurer qu'une propriété d'un élément a du domaine abstrait est bien une propriété des éléments du domaine concret qui a été abstrait en a.

4. Analyse de programmes LUSTRE

L'interaction de systèmes embarqués, tels qu'un système de commande de vol, avec leur environnement (l'avion, les gouvernes...) est réalisée par l'intermédiaire de capteurs dont les mesures ne peuvent être définies qu'avec une précision limitée. Ces erreurs de mesures des capteurs se manifestent sous la forme d'incertitudes sur les valeurs des variables d'entrée du programme, se propageant dans les calculs et induisant une incertitude sur les valeurs des sorties. L'analyse de la robustesse d'une fonction LUSTRE consiste alors à mesurer, pour des erreurs bornées sur les entrées, l'incertitude sur la valeur de la sortie, afin de s'assurer que celle-ci ne dépasse pas un seuil fixé.

Cette question nous a conduit à une approche consistant à interpréter les incertitudes par des intervalles réels et les différentes opérations sur les réels par des opérations de l'algèbre des intervalles. Plus formellement, soit F un nœud LUSTRE recevant un flot réel X et produisant un flot réel Y :

$$X \to F \to Y$$

Supposons que le flot d'entrée $X = (x_0, x_1, \cdots, x_n, \cdots)$ est perturbé, c'est-à-dire entaché d'une erreur de mesure due à l'imprécision des capteurs. Chaque valeur du flot d'entrée sera entachée d'une erreur. Notons par ϵ l'erreur générée par l'imprécision des capteurs. Ainsi, chaque élément x_i du flot d'entrée sera un élément de l'intervalle $[x_i - \epsilon, x_i + \epsilon]$ et les résultats y_i du flot de sortie $Y = (y_0, y_1, \cdots, y_n, \cdots)$ seront des éléments de l'intervalle $[y_i - \epsilon', y_i + \epsilon']$ où ϵ' est l'erreur de sortie. L'approche étudiée consiste alors à construire une interprétation abstraite générale du nœud F, notée F_a, comme une fonction sur l'algèbre des intervalles :

$$X_I \to F_a \to Y_I$$

avec $X_I = ([x_0 - \epsilon_0, x_0 + \epsilon_0], [x_1 - \epsilon_1, x_1 + \epsilon_1], \cdots, [x_n - \epsilon_n, x_n + \epsilon_n], \cdots)$ et $Y_I = ([y_0 - \epsilon'_0, y_0 + \epsilon'_0], [y_1 - \epsilon'_1, y_1 + \epsilon'_1], \cdots, [y_n - \epsilon'_n, y_n + \epsilon'_n], \cdots)$.

Le flot Y_I obtenu en sortie caractérise alors une (sur-approximation d'une) plage de valeurs prises par le flot Y lorsque le flot d'entrée X est soumis à une perturbation $(\epsilon_0, \epsilon_1, \cdots, \epsilon_n, \cdots)$.

F étant une fonction LUSTRE, c'est-à-dire opérant sur des flots (i.e. des suites de valeurs), il est alors nécessaire d'appliquer la fonction abstraite F_a de façon iterative pour obtenir une suite d'intervalles $([y_i - \epsilon'_i, y_i + \epsilon'_i])_{i \in N}$ de manière à pouvoir prendre en compte la dynamique des flots engendrée par les opérateurs `pre` et `->` du langage LUSTRE. Il n'est pas garanti que chaque intervalle de cette séquence d'intervalles corresponde à un plus grand intervalle borné (lorsqu'un tel intervalle n'est pas produit, l'interprétation abstraite fournit le plus grand intervalle qui correspond à la totalité du domaine associé à ces valeurs de flots). Cela peut alors signifier soit que la fonction LUSTRE F amplifie les erreurs propagées sur ses flots d'entrée (et donc n'est pas robuste), soit que l'interprétation abstraite est trop approximative.

En revanche, dans l'hypothèse où chaque intervalle de la suite d'intervalles $([y_i - \epsilon'_i, y_i + \epsilon'_i])_{i \in N}$ est un plus grand intervalle fourni par l'interprétation, alors on peut calculer l'erreur maximale produite sur le flot Y comme étant la plus grande largeur des intervalles qui représentent les valeurs des flots de sortie et qui vaut donc $max(\epsilon'_i)_{(i \in N)}$.

La définition précédente est générale et peut être appliquée à tout nœud LUSTRE F qui calcule des flots réels à partir de flots réels en entrée dès lors que l'interprétation abstraite des constructions de base de LUSTRE est définie.

La généralité de cette démarche permet d'envisager différentes utilisations possibles :

1- une approche générale, à l'image de la description précédente, est possible avec :

$$X_I \rightarrow F_a \rightarrow Y_I$$

avec $X_I = ([x_0 - \epsilon_0, x_0 + \epsilon_0], [x_1 - \epsilon_1, x_1 + \epsilon_1], \cdots, [x_n - \epsilon_n, x_n + \epsilon_n], \cdots)$ et $Y_I = ([y_0 - \epsilon'_0, y_0 + \epsilon'_0], [y_1 - \epsilon'_1, y_1 + \epsilon'_1], \cdots, [y_n - \epsilon'_n, y_n + \epsilon'_n], \cdots)$

Ici la précision est variable pour toutes les valeurs du flot d'entrée et sera donc variable pour le flot de sortie.

2- Lorsque l'erreur sur les valeurs du flot d'entrée est modélisée par une fonction (un nœud LUSTRE par exemple) $erreur(x)$, l'interprétation définie peut être utilisée de la façon suivante :

$$X_I \rightarrow F_a \rightarrow Y_I$$

avec $X_I = ([x_0 - erreur(x_0), x_0 + erreur(x_0)], [x_1 - erreur(x_1), x_1 + erreur(x_1)], \cdots, [x_n - erreur(x_n), x_n + erreur(x_n)], \cdots)$ et $Y_I = ([y_0 - \epsilon'_0, y_0 + \epsilon'_0], [y_1 - \epsilon'_1, y_1 + \epsilon'_1], \cdots, [y_n - \epsilon'_n, y_n + \epsilon'_n], \cdots)$.

Si la fonction *erreur* est constante, on supposera que les perturbations sont constantes. Si un modèle de calcul de l'erreur sur les entrées (modèle d'usure des capteurs par exemple) est connu, alors un nœud *erreur* est en mesure de fournir une perturbation des entrées.

3- Enfin, une utilisation particulière consiste à fournir un flot d'entrées à valeur constante. Cette approche permet d'étudier la convergence, du calcul perturbé, lorsque le flot est constant. Cependant, il faut remarquer que dans ce cas, nous ne maîtrisons pas la convergence de ce calcul vers un intervalle représentant l'erreur. Soit $e\_X(x_0)$ et $E\_X(x_0)$ les erreurs minimale et maximale véhiculées par X lorsque celui-ci vaut x_0. On construit alors le nœud F_a comme :

$$[x_0 + e\_X(x_0), x_0 + E\_X(x_O)] \rightarrow F_a \rightarrow [yp_0, YP_0]$$

Si la suite d'intervalles $[yp_0^i, YP_0^i]_{(i \in N)}$ converge vers un plus grand intervalle $[yp_0, YP_0]$ alors on peut calculer l'erreur maximale produite sur le flot Y. Notons alors $y_0 = F(x_0)$ la valeur du flot Y retournée par F lorsque le flot X vaut x_0 et n'est entaché d'aucune erreur.

Les utilisations que nous avons effectuées dans le cadre de cette étude correspondent au cas 2 avec une précision constante sur les entrées.

Deux interprétations ont été définies. La première, exposée en section 4.1, produit des approximations trop grandes et non significatives. Une seconde abstraction, permettant de réduire considérablement le niveau d'approximation, a alors été définie et présentée en section 4.2. L'intérêt de cette seconde abstraction est de permettre la détection des sources de propagation des incertitudes.

4.1. Première analyse

La première analyse que nous avons développée est issue de l'algèbre d'intervalles classique.

Définition du domaine concret. La définition du domaine concret découle de la caractérisation du domaine d'exécution de LUSTRE : il doit être composé de l'ensemble des variables réelles et booléennes du programme. Cependant, cet ensemble doit être complété afin de constituer un domaine. Les éléments $\perp_{D_{\mathcal{B}}}$, la variable booléenne indéfinie, et $\perp_{D_{\mathcal{R}}}$, la variable réelle indéfinie, sont donc introduits.

Le domaine $D_{\mathcal{B}}$ des variables booléennes est alors défini comme l'union des variables booléennes du programme et de l'élément $\perp_{D_{\mathcal{B}}}$; similairement, l'élément $\perp_{D_{\mathcal{R}}}$ se joint à l'ensemble des variables réelles pour donner le domaine $D_{\mathcal{R}}$ des variables réelles.

La somme amalgamée de ces deux domaines constitue enfin le domaine concret $D_{\perp}$.

Par la suite, les éléments du domaine concret seront désignés par des lettres minuscules, d désignant généralement un élément quelconque de $D_{\perp}$, b un booléen, et r un réel.

Définition des booléens abstraits. La structure de l'ensemble $A_{\mathcal{B}}$ des variables booléennes abstraites est un treillis, comportant quatre éléments : $Undef$, $True$, $False$, et $Maybe$.

- La première valeur, $Undef$, est la transposition de l'élément $\perp_{D_{\mathcal{B}}}$ du domaine concret, c'est à dire l'élément $\perp_{A_{\mathcal{B}}}$.
- Les constantes $True$ et $False$ correspondent aux valeurs classiques des booléens.
- La dernière valeur, $Maybe$, est l'élément maximal $\top_{A_{\mathcal{B}}}$ de la représentation abstraite. Elle lui est propre, et correspond précisément aux cas où le booléen est perturbé, sa valeur effective, dans le programme original, pouvant alors être aussi bien `true` que `false`, et une détermination plus précise étant impossible.

Enfin, cette structure est complétée par la définition de l'ordre partiel $\sqsubseteq$, et des opérateurs binaires $\sqcup$ et $\sqcap$ sur les booléens.

Définition des réels abstraits. La structure de l'ensemble $A_{\mathcal{R}}$ des variables réelles abstraites est également un treillis, composé des réels abstraits, de l'élément $\perp_{A_{\mathcal{R}}}$, et de l'élément $\top_{A_{\mathcal{R}}}$.

- Les variables réelles sont abstraites sous la forme d'un intervalle. Cet intervalle inclut l'ensemble des valeurs possibles pour la variable perturbée considérée.
- L'élément $\perp_{A_{\mathcal{R}}}$ est la transposition de l'élément $\perp_{D_{\mathcal{R}}}$ du domaine concret, et correspond à un intervalle indéfini.
- L'élément $\top_{A_{\mathcal{R}}}$ est l'élément maximal, regroupant toutes les valeurs possibles des variables concrètes. Il correspond donc à l'intervalle $[\,MIN_{\mathcal{R}}\,;\,MAX_{\mathcal{R}}\,]$, où $MIN_{\mathcal{R}}$ est la plus petite valeur possible pour une variable réelle ou $-\infty$, et $MAX_{\mathcal{R}}$ la plus grande valeur possible ou $+\infty$.

L'ordre partiel $\sqsubseteq$ de ce treillis est caractérisé par les relations :

$$\forall\, R \in A_{\mathcal{R}},\ \perp_{A_{\mathcal{R}}} \sqsubseteq R, \quad R \sqsubseteq \top_{A_{\mathcal{R}}} \quad \text{et}\,,$$

$$\forall\, (R_i, R_j) \in A_{\mathcal{R}}^2,\ \text{dont les intervalles sont } \boldsymbol{I} \text{ et } \boldsymbol{J},\ \text{avec } \boldsymbol{I} = \left[\,\underline{i}\,;\,\overline{i}\,\right] \text{ et } \boldsymbol{J} = \left[\,\underline{j}\,;\,\overline{j}\,\right]\,,$$

$$R_i \sqsubseteq R_j\text{"} <=> \text{"}\underline{j} \leq \underline{i} \text{ et } \overline{i} \leq \overline{j}\ .$$

Avec les mêmes notations, sont ensuite définis les opérateurs *join* $\sqcup$ et *meet* $\sqcap$:

$$\begin{array}{rcl} R_i \sqcup R_j & = & \top_{A_\mathcal{R}}, \quad \text{si } \min\{\underline{i}, \underline{j}\} = MIN_\mathcal{R} \text{ et } \max\{\overline{i}, \overline{j}\} = MAX_\mathcal{R} \\ & = & \bot_{A_\mathcal{R}} \quad \text{si } R_i = \bot_{A_\mathcal{R}} \text{ et } R_j = \bot_{A_\mathcal{R}} \\ & = & R_i \quad \text{si } R_j = \bot_{A_\mathcal{R}}, \quad \text{ou } R_j \quad \text{si } R_i = \bot_{A_\mathcal{R}} \\ & = & R_k \quad \text{avec } \boldsymbol{K} = [\min\{\underline{i}, \underline{j}\}\,;\, \max\{\overline{i}, \overline{j}\}] \quad \text{sinon.} \end{array}$$

$$\begin{array}{rcl} R_i \sqcap R_j & = & \bot_{A_\mathcal{R}}, \quad \text{si } R_i = \bot_{A_\mathcal{R}} \text{ ou } R_j = \bot_{A_\mathcal{R}} \text{ ou } \max\{\underline{i}, \underline{j}\} > \min\{\overline{i}, \overline{j}\} \\ & = & \top_{A_\mathcal{R}}, \quad \text{si } R_i = \top_{A_\mathcal{R}} \text{ et } R_j = \top_{A_\mathcal{R}} \\ & = & R_k \quad \text{avec } \boldsymbol{K} = [\max\{\underline{i}, \underline{j}\}\,;\, \min\{\overline{i}, \overline{j}\}] \quad \text{sinon.} \end{array}$$

Interprétation des constructions LUSTRE. Le domaine abstrait étant précisé, il reste alors à définir l'interprétation des opérateurs LUSTRE dans ce domaine abstrait. Par souci de concision, nous nous limitons dans cette présentation aux opérations d'addition, de comparaison ($<$) ainsi qu'au traitement de la conditionnelle :

- l'addition : elle est définie par $I + J = [\,\underline{I} + \underline{J}\,;\, \overline{I} + \overline{J}\,]$
- la comparaison : considérons le booléen sur le domaine abstrait des intervalles $B = I < J$, avec deux intervalles I et J. La valeur de B est alors :
  - $Undef$ si $I = \bot_A$ ou $J = \bot_A$,
  - $True$ si $\overline{I} < \underline{J}$,
  - $False$ si $\overline{J} < \underline{I}$,
  - $Maybe$ sinon .
- la conditionnelle s'écrit I = if B then J else K, où B est un booléen faisant intervenir des comparaisons d'intervalles, et I, J et K sont des intervalles correspondant à des variables réelles abstraites. Conformément à la définition classique du choix conditionnel,
  – si $B = True$, alors $I = J$,
  – si $B = False$, alors $I = K$,
  – enfin, si B n'est pas défini (et vaut $Maybe$), alors $I = [min(\underline{I}, \underline{J}), max(\overline{I}, \overline{J})]$

A cause de l'interprétation de la conditionnelle qui renvoie un intervalle englobant dans le cas où la condition est $Maybe$, cette analyse fournit des approximations correctes, mais trop larges.

En effet, après évaluation des programmes abstraits, les intervalles décrivant les incertitudes en sorties sont trop grands pour être significatifs. Les propriétés ainsi calculées ne permettent ni de localiser les portions de code qui entraînent cette approximation grossière, ni de proposer des transformations de programmes permettant de réduire cette incertitude.

4.2. Améliorations

La seconde analyse reprend les domaines de concrétisation et d'abstraction de la première, en enrichissant la définition des réels abstraits d'informations contextuelles et en redéfinissant l'interprétation des constructions du langage, de manière à tirer parti de ces informations supplémentaires.

Informations contextuelles. Les informations contextuelles incluses dans une variable réelle abstraite sont un ensemble de paires d'intervalles. Chacune est associée à un booléen du programme : étant donnée une variable réelle r_i, abstraite en R_i, d'intervalle $\boldsymbol{I}$, les intervalles contextuels $\boldsymbol{I}'_n$ et $\boldsymbol{I}''_n$ sont associés au booléen concret b_n, abstrait en B_n. Ces intervalles sont définis formellement par :

$$\boldsymbol{I}'_n = \left\{ i \in \boldsymbol{I} \;\middle|\; \begin{array}{c} \exists\ (e_1,\ e_2,\ \ldots,\ e_t) \in conc_{\mathcal{R}}(E_1) \times conc_{\mathcal{R}}(E_2) \times \cdots \times conc_{\mathcal{R}}(E_t) \\ \text{tel que } b_n(e_1,\ e_2,\ \ldots,\ e_t) = \texttt{ true} \text{ et } r_i(e_1,\ e_2,\ \ldots,\ e_t) = i \end{array} \right\},$$

$$\boldsymbol{I}''_n = \left\{ i \in \boldsymbol{I} \;\middle|\; \begin{array}{c} \exists\ (e_1,\ e_2,\ \ldots,\ e_t) \in conc_{\mathcal{R}}(E_1) \times conc_{\mathcal{R}}(E_2) \times \cdots \times conc_{\mathcal{R}}(E_t) \\ \text{tel que } b_n(e_1,\ e_2,\ \ldots,\ e_t) = \texttt{ false} \text{ et } r_i(e_1,\ e_2,\ \ldots,\ e_t) = i \end{array} \right\},$$

où $E_1,\ \ldots,\ E_t$ sont les variables d'entrée du programme abstrait, et $conc_{\mathcal{R}}$ est la fonction de concrétisation ; la notation $var(e_1,\ e_2,\ \ldots,\ e_t)$ indique que la variable concrète var à été calculée par l'algorithme original en prenant $e_1,\ e_2,\ \ldots,\ e_t$ comme valeur pour les variables d'entrée.

Plus intuitivement, l'intervalle $\boldsymbol{I}'_n$ peut être défini comme l'ensemble des valeurs de la variable réelle R_i compatibles avec une valeur `true` du booléen b_n. Cet ensemble de valeurs est toujours un intervalle inclus dans l'intervalle des valeurs possibles $\boldsymbol{I}$ de la variable R_i. Similairement, $\boldsymbol{I}''_n$ est inclus dans $\boldsymbol{I}$, et regroupe les valeurs compatibles avec $b_n =$ `false`.

Enfin, la représentation naturelle des informations contextuelles est un tableau d'intervalles, comportant 2 lignes et une colonne par booléen dans le programme. Par la suite, l'intervalle des valeurs de R_i, compatibles avec une valeur `true` ou `false` du booléen b_n sera noté $R_i.tab[n][\texttt{true}]$ ou $R_i.tab[n][\texttt{false}]$.

Interprétation des opérateurs booléens. Les opérations élémentaires à redéfinir sur les booléens abstraits sont la négation ($\neg_a$) et la conjonction ($\wedge_a$) abstraites.

Négation : $B_n = \neg_a B_p$

Si $B_p = Undef$, alors $B_n = Undef$
Si $B_p = True$, alors $B_n = False$
Si $B_p = False$, alors $B_n = True$
Si $B_p = Maybe$, alors $B_n = Maybe$

Par ailleurs, le tableau d'intervalles de chaque variable réelle abstraite R du programme doit être mis à jour :

$$\begin{array}{rcl} R.tab[n][\texttt{true}] & = & R.tab[p][\texttt{false}] \\ R.tab[n][\texttt{false}] & = & R.tab[p][\texttt{true}] \end{array}$$

Conjonction : $B_n = B_p \wedge_a B_q$

Si $B_p = Undef$ ou $B_q = Undef$, alors $B_n = Undef$
Si $B_p = True$ et $B_q = True$, alors $B_n = True$
Si $B_p = False$ ou $B_q = False$, alors $B_n = False$
Sinon $B_n = Maybe$

Le tableau d'intervalles de chaque variable réelle R est également mis à jour :

$$\begin{array}{rcl} R.tab[n][\texttt{true}] & = & R.tab[p][\texttt{true}] \sqcap R.tab[q][\texttt{true}] \\ R.tab[n][\texttt{false}] & = & R.tab[p][\texttt{false}] \sqcup R.tab[q][\texttt{false}] \end{array}$$

Disjonction : $B_n = B_p \vee_a B_q$

Si $B_p = Undef$ ou $B_q = Undef$, alors $B_n = Undef$
Si $B_p = True$ ou $B_q = True$, alors $B_n = True$
Si $B_p = False$ et $B_q = False$, alors $B_n = False$
Sinon $B_n = Maybe$

Là encore, le tableau est mis à jour :

$$\begin{array}{rcl} R.tab[n][\texttt{true}] & = & R.tab[p][\texttt{true}] \sqcup R.tab[q][\texttt{true}] \\ R.tab[n][\texttt{false}] & = & R.tab[p][\texttt{false}] \sqcap R.tab[q][\texttt{false}] \end{array}$$

Interprétation des opérateurs réels. Les opérations élémentaires d'addition, de soustraction, de multiplication et de division sont définies comme précédemment, sur le domaine abstrait. Cependant, lors d'un calcul $R_k = R_i \star_a R_j$, le même calcul est effectué sur les intervalles des tableaux de R_i et R_j, afin d'obtenir les intervalles du tableau de R_k, pour tout booléen B_n :

$$R_k.tab[n][b] = R_i.tab[n][b] \star_{\boldsymbol{a}} R_j.tab[n][b]$$

Interprétation des comparaisons. On considère le booléen abstrait $B_n = R_i <_a R_j$, avec R_i et R_j deux variables réelles abstraites dont les intervalles sont $\boldsymbol{I} = \left[\,\underline{i}\,;\,\overline{i}\,\right]$ et $\boldsymbol{J} = \left[\,\underline{j}\,;\,\overline{j}\,\right]$. Comme précédemment, la valeur de B_n est alors :

- $Undef$ si $\boldsymbol{I} = \perp_{A_\mathcal{R}}$ ou $\boldsymbol{J} = \perp_{A_\mathcal{R}}$,
- $True$ si $\overline{i} < \underline{j}$,
- $False$ si $\overline{j} < \underline{i}$,
- $Maybe$ sinon .

Cependant, lorsque la valeur calculée pour B_n est $Maybe$, il est possible de déterminer une valeur plus précise de ce booléen pour certains intervalles de valeurs inclus dans les intervalles de référence de R_i ou R_j. Réciproquement, en supposant connue la valeur de b_n calculée par le programme original, il est possible de calculer des intervalles restreints représentant les valeurs de R_i ou R_j compatibles avec cette valeur de b_n. Ainsi, si $\overline{j} < \overline{i}$, seules les valeurs de l'intervalle $\boldsymbol{I}'_n = \left[\,\underline{i}\,;\,\overline{j}\,\right] \subseteq \boldsymbol{I}$ sont compatibles avec une valeur `true` de b_n : $\forall\, r_i \in \left[\,\overline{j}\,;\,\overline{i}\,\right]$ et $\forall\, r_j \in \boldsymbol{J}$, $r_i < r_j$ retourne `false`.

De façon plus générale, les restrictions des intervalles de référence compatibles avec des valeurs précises de B_n sont résumées dans le tableau suivant :

b_n	$\boldsymbol{I}$	$\boldsymbol{J}$
`true`	$\boldsymbol{I}'_n = \left[\,\underline{i}\,;\,\min\left\{\,\overline{i}\,,\,\overline{j}\,\right\}\right]$	$\boldsymbol{J}'_n = \left[\max\left\{\,\underline{i}\,,\,\underline{j}\,\right\}\,;\,\overline{j}\,\right]$
`false`	$\boldsymbol{I}''_n = \left[\max\left\{\,\underline{i}\,,\,\underline{j}\,\right\}\,;\,\overline{i}\,\right]$	$\boldsymbol{J}''_n = \left[\,\underline{j}\,;\,\min\left\{\,\overline{i}\,,\,\overline{j}\,\right\}\right]$

Ces intervalles restreints sont calculés, et mémorisés dans les tableaux d'intervalles des variables R_i et R_j, dans la colonne qui correspond au booléen B_n :

$$\begin{array}{cc} R_i.tab[n][\texttt{true}] = \boldsymbol{I}'_n & R_j.tab[n][\texttt{true}] = \boldsymbol{J}'_n \\ R_i.tab[n][\texttt{false}] = \boldsymbol{I}''_n & R_j.tab[n][\texttt{false}] = \boldsymbol{J}''_n \end{array}$$

Interprétation de la conditionnelle. L'expression considérée dans ce paragraphe est la forme abstraite du choix conditionnel, et s'écrit

$$R_i = \ \mathbf{if}_a\ B_n\ \mathbf{then}_a\ R_j\ \mathbf{else}_a\ R_k,$$

où B_n est un booléen abstrait, et R_i, R_j et R_k sont des variables réelles abstraites auxquelles sont respectivement associés les intervalles $\boldsymbol{I}$, $\boldsymbol{J}$ et $\boldsymbol{K}$.

Conformément à la définition classique du choix conditionnel,

- si $B_n = True$, alors $\boldsymbol{I} = \boldsymbol{J}$,
- si $B_n = False$, $\boldsymbol{I} = \boldsymbol{K}$,
- enfin, si B_n n'est pas défini, alors R_i ne l'est pas non plus, et $\boldsymbol{I} = \perp_{A_\mathcal{R}}$.

Cependant, lorsque $B_n = Maybe$, l'ensemble des valeurs possibles pour $\boldsymbol{I}$ sont les valeurs de $\boldsymbol{J}$ compatibles avec une valeur `true` de b_n réunies aux valeurs de $\boldsymbol{K}$ compatibles, elles, avec une valeur `false` du booléen. Cet ensemble de valeurs est l'union de deux intervalles $\boldsymbol{J}'_n$ et $\boldsymbol{K}''_n$ inclus respectivement dans les intervalles $\boldsymbol{J}$ et $\boldsymbol{K}$. Ces intervalles, $\boldsymbol{J}'_n$ et $\boldsymbol{K}''_n$, sont justement ceux mémorisés dans le tableau des variables R_j et R_k : $\boldsymbol{J}'_n = R_j.tab[n][\texttt{true}]$ et $\boldsymbol{K}''_n = R_k.tab[n][\texttt{false}]$.

Toutefois, dans la structure de donnée choisie, R_i est représentée par un intervalle unique, et cette union d'intervalles ne peut se représenter sous cette forme de manière exacte que si les intervalles $\boldsymbol{J}'_n$ et $\boldsymbol{K}''_n$ ne sont pas disjoints. Ainsi la valeur retenue pour $\boldsymbol{I}$ est le plus petit intervalle contenant ces deux intervalles, ce qui introduit, lorsqu'ils sont disjoints, une sur-approximation pour la valeur de R_i. Ce plus petit intervalle est donné par l'opérateur *join* $\sqcup$ défini sur les réels abstraits. Finalement,

– si $B_n = Maybe$, alors $\boldsymbol{I} = R_j.tab[n][\texttt{true}] \sqcup R_k.tab[n][\texttt{false}]$.

La preuve de la correction de l'abstraction ainsi définie est donnée dans [Pai02]. De même, on trouvera dans [Pai02] une description JAVA de l'approche proposée.

4.3. Application à l'exemple

L'interprétation abstraite de programmes LUSTRE proposée précédemment, appliquée à l'exemple présenté en section 3, a permis de montrer que, sous l'hypothèse que les erreurs minimales et maximales de `Xgouv` et `Sensor` sont constantes et définies par

$$e\_Xgouv = -0.5 \quad \text{et} \quad E\_Xgouv = 0.5$$

$$e\_Sensor = -0.04 \quad \text{et} \quad E\_Sensor = 0.04$$

la suite des intervalles obtenue par application itérative de la fonction abstraite $\texttt{asserv}_a$ converge, et permet de calculer les erreurs minimales et maximales :

$$e\_Consigne = -2.816 \quad \text{et} \quad E\_Consigne = 2.816$$

$$e\_Courant = -0.75 \quad \text{et} \quad E\_Courant = 0.75$$

.

Il est à noter cependant que les erreurs ainsi calculées restent des sur-approximations de l'erreur réelle qui est sur cet exemple

$$e\_Consigne_{vraie} = -2.048 \quad \text{et} \quad E\_Consigne_{vraie} = 2.048$$
$$e\_Courant_{vraie} = -0.51 \quad \text{et} \quad E\_Courant_{vraie} = 0.51$$

5. Conclusion et perspectives

Cet article a présenté une interprétation abstraite du langage LUSTRE (sans horloge) pour l'étude de la propagation d'erreurs de mesure. Cette approche s'avère comparable à l'abstraction proposée dans [AA97]. Celle-ci s'applique aux programmes numériques, et étudie la précision des calculs de ces programmes en analysant leurs propriétés non-fonctionnelles. La représentation des nombres flottants qui y est choisie se compose d'un intervalle, et des tailles nécessaires au codage de la mantisse et de l'exposant. Cependant, si, d'un côté, l'analyse présentée ici est moins précise, puisqu'elle ne formalise pas la représentation des réels sous la forme de leur mantisse et de leur exposant, elle apparaît également plus riche, par l'introduction de valeurs contextuelles.

De nombreux développements futurs sont envisageables. La première extension possible, et peut-être la plus fructueuse, serait d'enrichir la représentation des variables réelles abstraites, afin de gagner en précision. A cette fin, une représentation des valeurs possibles pour les variables réelles, par une liste d'intervalles (figurant l'union des intervalles de la liste), serait une amélioration conséquente, puisqu'elle permettrait de représenter les unions d'intervalles disjoints de manière exacte. L'opérateur *join*, sur les réels abstraits, devrait donc être redéfini, ainsi que toutes les opérations faisant intervenir les intervalles, dans l'abstraction actuelle. Le fondement de la modification de ces opérations consisterait, en fait, à réaliser le même calcul que précédemment, pour chaque intervalle de la liste de la variable réelle représentée. Cet aspect de la modification entraine donc, potentiellement, une explosion combinatoire de la taille des listes représentant les réels, lorsque deux variables réelles interviennent dans une opération. Par ailleurs, il serait également nécessaire de définir une forme normalisée (i.e., une liste classée d'intervalles disjoints) pour garantir l'unicité de la représentation d'un ensemble de valeurs donné.

Selon un autre axe de développement possible, des transformations du programme avant son interprétation pourrait également conduire à des gains de précision. Ainsi, le développement ou la factorisation d'une expression arithmétique peut modifier la précision du calcul réalisé ultérieurement avec des intervalles. En effet la définition donnée pour les opérateurs abstraits $+_a$ et $\times_a$ garantit leur commutativité et leur associativité, mais la distributivité n'est pas vérifiée. L'exemple suivant illustre simplement cette remarque :

$$
\begin{array}{rrcl}
\text{Soit l'expression} & \eta(x) & = & x^2 - x \\
\text{alors} & \eta(x) & = & x(x-1)\ , \quad \text{en factorisant.} \\
\text{Si } \boldsymbol{X} = [1;2]\,, & \eta(\boldsymbol{X}) & = & (\boldsymbol{X} \times_a \boldsymbol{X}) -_a \boldsymbol{X} = [1;4] -_a [1;2] \\
 & & = & [-1;3] \\
\text{et} & \eta(\boldsymbol{X}) & = & \boldsymbol{X} \times_a (\boldsymbol{X} -_a \mathbf{1}) = [1;2] \times_a [0;1] \\
 & & = & [0;2]
\end{array}
$$

La deuxième expression, factorisée, donne donc ici un résultat plus précis que l'expression développée. Cependant, il est, a priori, impossible de prévoir quelle forme d'une expression donnera le résultat le plus précis. Il serait donc sans doute nécessaire d'implémenter un algorithme soumettant successivement à l'interprétation abstraite plusieurs formes différentes de chaque expression arithmétique du programme. Le résultat retenu serait alors l'intersection des différents intervalles obtenus.

Par ailleurs, il est aisé de voir que les problèmes soulevés par la sous-distributivité des opérateurs arithmétiques abstraits, sont amplifiés lorsqu'une variable possède plusieurs occurrences dans une même expression. Ces problèmes sont dus à la perte du lien entre les valeurs de ces différentes occurrences. Ainsi, dans l'exemple précédent, les deux occurrences de la variable x ont, dans le même calcul, implicitement pris des valeurs différentes dans leur intervalle. Une façon de résoudre ce problème pourrait être d'étendre la notion de contexte déjà utilisée pour les réels : chaque variable se verrait alors associées des valeurs contextuelles, consistant en des ensembles de valeurs plus réduits que l'ensemble général de valeurs caractérisant la variable. Chacune de ces valeurs contextuelles serait elle-même associée à une autre variable, dont la première serait dépendante (existence d'un "parent" commun).

Une autre approche, en cours d'exploration par G. Bel [Bel01] pour ce même problème, consiste à utiliser les formalismes et les outils de l'optimisation combinatoire, et notamment la programmation par contrainte. Les fonctions étudiées étant exprimées en LUSTRE / SCADE, l'idée naturelle revient à traduire chaque flot X de la spécification LUSTRE en deux suites de variables : une première suite $X[i]$ où i est un entier compris entre 0 et un horizon h, et une deuxième suite $Xb[i]$ définie sur le même horizon. $Xb[i]$ est la valeur bruitée de $X[i]$. Chaque équation de la spécification LUSTRE telle que

```
Consigne = (y*0.75)
           + ((0 ;0  -> pre(y))*0.5)
           - ((0.0 -> pre(0.0 -> pre(y)))*0.25)
```

est alors traduite deux ensembles de contraintes linéaires, par exemple :

```
Consigne[i] = (y[i]*0.75) + (y[i-1]*0.5) - (y[i-2]*0.25) pour 1<i<h
Consigne[1] = (y[1]*0.75) + (y[0]*0.5)
Consigne[0] = (y[0]*0.75)
```

et

```
Consigneb[i] = (yb[i]*0.75) + (yb[i-1]*0.5) - (yb[i-2]*0.25) pour i>1
Consigneb[1] = (yb[1]*0.75) + (yb[0]*0.5)
Consigneb[0] = (yb[0]*0.75)
```

A chaque flot d'entrée E de la spécification LUSTRE est également associé une erreur erE variant dans une plage de valeur donnée. Les entrées bruitées de la spécification sont alors définies par la contrainte

```
Eb[i] = E[i] + erE[i] pour tout 0<=i<h
```

La détermination des erreurs sur les sorties sera obtenue par la contrainte

```
erS[i] = S[i] - Sb[i]
```

Le système de contraintes étant construit, la vérification de la robustesse de la fonction LUSTRE, c'est-à-dire que l'erreur propagée sur les sorties de la fonction est toujours inférieure à une valeur maximale souhaitée, se fera par l'ajout de la contrainte

```
erS[h] > val_max
```

et par la recherche d'une configuration des variables (et donc des valeurs des flots au cours du temps) qui respecte les contraintes issues de la spécification LUSTRE et qui respecte la contrainte ci-dessus. Si une telle configuration est trouvée, alors la fonction pourra être déclarée non robuste. En revanche, si aucune configuration n'est trouvée, il n'existe aucune suite de valeurs respectant la spécification LUSTRE et conduisant à une erreur supérieur à `val_max`. La fonction est alors robuste.

Cette méthode présente l'avantage d'être très proche de la spécification LUSTRE initiale et de conduire au calcul de la borne maximale exacte des erreurs propagées sur les sorties de la spécification. En revanche, cette méthode conduit rapidement à une explosion combinatoire qui risque de s'avérer rédhibitoire. En effet, plus la dynamique de la spécification est grande (retards, temporisations... élevés), plus l'horizon est grand et plus le nombre de variables de la modélisation par contrainte (les $X[i]$ et $Xb[i]$) est élevé. Une piste envisageable, et complémentaires aux axes de développements déjà énoncés, serait alors de coupler cette technique par contraintes et la technique exposée dans le présent article, plus rapide mais sur-approximative.

Références

[AA97] Y. Ait-Ameur. A Contribution to Program Comprehension by Program Analysis : Application to Numerical Programs. In *Proceedings of 12th IEEE International Automated Software Engineering Conference*, pages 134–150. IEEE Computer Society Press, 11 1997.

[AH87] S. Abramsky and C. Hankin. *Abstract Interpretation of Declarative Languages.* Ellis Horwood Limited, 1987.

[Bel01] G. Bel. *Etude de la propagation d'erreurs de capteurs dans une spécification de commande de vol SCADE : une approche par contraintes* Notes de travail, septembre 2001.

[BHA86] G.L. Burn, C.L. Hankin, and S. Abramsky. Strictness Analysis of Higher Order Functions. *Science of Computer Programming*, 7 :249–278, 1986.

[CHPR91] P. Caspi, N. Halbwachs, P. Pilaud, and P. Raymond. The synchronous dataflow programming language Lustre. *Proceedings of IEEE, Another Look at Real-time programming*, 79(9) :1305–1319, September 1991.

[CC77] P. Cousot and R. Cousot. Abstract Interpretation : A unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. *4th POPL*, pages 238–252, 1977.

[HFP93] N. Halbwachs, Lagnier F., and Raymond P. Synchronous observer and the verification of reactive systems. In *Third International Conference on Algebraic Methodology and Software Technology, AMAST'93*, Twente, June 93.

[HLR92] N. Halbwachs, F. Lagnier, and C. Ratel. Programming and verifying real time systems by means of the synchronous data-flow language Lustre. *IEEE Transactions on Software Engineering, special issue on the specification and analysis of real time systems*, september 1992.

[HCW02] Mats P.E. Heimdahl, Yunja Choi, and Mike Whalen. Deviation analysis through model checking. In *Proceedings of ASE 2002*, 2002.

[Mel87] C. Mellish. Abstract Interpretation of PROLOG Programs. In S. Abramsky and C. Hankin. Ellis Horwood, editors, *Abstract Interpretation of Declarative Languages*, pages 181–198, 1987.

[Pai02] S. Pairault. Méthode d'analyse automatique de la robustesse d'un programme LUSTRE-SCADE. Rapport de stage, Ecole Polytechnique, juillet 2002.