Vérification de la génération modulaire du code impératif pour Lustre

Timothy Bourke, Pierre-Evariste Dagand, Marc Pouzet, Lionel Rieg

Vérification de la génération modulaire du code impératif pour Lustre

Timothy Bourke[1,2] & Pierre-Évariste Dagand[3,4,1] & Marc Pouzet[3,2,1] & Lionel Rieg[5,6]

1: Inria Paris
2: École normale supérieure, PSL Research University
3: Sorbonne Universités, UPMC Univ Paris 06
4: CNRS, LIP6 UMR 7606
5: Collège de France
6: Yale University

Résumé

Les langages synchrones sont utilisés pour programmer des logiciels de contrôle-commande d'applications critiques. Le langage Scade, utilisé dans l'industrie pour ces applications, est fondé sur le langage Lustre introduit par Caspi et Halbwachs. On s'intéresse ici à la formalisation et la preuve, dans l'assistant de preuve Coq, d'une étape clef de la compilation : la traduction de programmes Lustre vers des programmes d'un langage impératif. Le défi est de passer d'une sémantique synchrone flot de données, où un programme manipule des flots, à une sémantique impérative, où un programme manipule la mémoire de façon séquentielle. Nous spécifions et vérifions un générateur de code simple qui gère les traits principaux de Lustre : l'échantillonnage, les nœuds et les délais. La preuve utilise un modèle sémantique intermédiaire qui mélange des traits flot de données et impératifs et permet de définir un invariant inductif essentiel. Nous exploitons la formalisation proposée pour vérifier une optimisation classique qui fusionne des structures conditionnelles dans le code impératif généré.

1. Introduction

Lustre a été introduit en 1987 comme langage de programmation pour des systèmes numériques de contrôle-commande et de traitement du signal [7]. C'est un langage d'équations flot de données de type schémas-blocs qui a donné naissance au langage industriel Scade [1]. Il peut servir comme cible pour compiler un sous-ensemble synchrone de Simulink [2] [6]. Il y a plusieurs raisons pour lesquelles les langages à la Lustre conviennent aux applications critiques comme les commandes de vol ou la surveillance de centrales électriques : des constructions dédiées à la programmation réactive, une exécution en mémoire et temps bornés statiquement, une sémantique mathématiquement bien définie [7], une compilation traçable et modulaire [3] et la faisabilité de la vérification automatique des programmes [13, 14] et de la certification industrielle. Ces langages permettent aux ingénieurs de développer et valider leurs systèmes au niveau des schémas-blocs qui sont ensuite compilés directement vers du code exécutable.

La compilation traduit un ensemble d'équations définissant des flots de valeurs vers une séquence d'instructions impératives qui manipule la mémoire d'une machine. L'exécution répétée de ces instructions est censée générer les valeurs successives des flots originaux : mais comment le garantir ? Notre réponse est de spécifier formellement les langages, leurs sémantiques et le processus de

1. `http://www.ansys.com/products/embedded-software/ansys-scade-suite`
2. `http://www.mathworks.com/products/simulink/`

compilation dans un assistant de preuve, puis d'énoncer et de démontrer une relation de correction entre les sémantiques des programmes source et cible. Nous décrivons ici la vérification de la passe de compilation qui génère le code impératif.

1.1. Un programme Lustre

Un programme Lustre est composé d'un ensemble de *nœuds*. Un nœud spécifie une fonction entre flots d'entrée et flots de sortie qui est définie par un ensemble d'équations. Considérons l'exemple suivant :

```
node count (ini: int; inc: int; restart: bool) returns (n: int)
  var c: int; f: bool;
let
  n = if (f or restart) then ini else c + inc;
  f = true fby false;
  c = 0 fby n;
tel
```

Ce nœud count prend trois flots d'entrée — deux d'entiers et un de booléens — et rend un flot d'entiers. La première équation définit la sortie n avec une expression qui contient trois opérateurs (if/then/else, or et +). La deuxième équation définit une variable locale f qui n'est vraie qu'au premier instant [3]. La troisième équation définit une variable locale c comme un flot qui est initialement 0 puis égal à la valeur précédente de n. Une fois défini, un nœud peut être instancié dans d'autres nœuds, comme dans l'exemple suivant :

```
node avgvelocity (delta: int; sec: bool) returns (v: int)
  var r, t, h: int;
let
  r = count(0, delta, false);
  t = count(1 when sec, 1 when sec, false when sec);
  v = merge sec ((r when sec) / t) (h whenot sec);
  h = 0 fby v;
tel
```

Considérons les flots de ce nœud pour des valeurs spécifiques des entrées delta et sec :

delta	1	2	1	2	3	0	3	0	$\cdots$
sec	F	F	T	F	T	T	F	F	$\cdots$
r	0	2	3	5	8	8	11	11	$\cdots$
(c_r)	0	0	2	3	5	8	8	11	$\cdots$
t			1		2	3			$\cdots$
(c_t)			0		1	2			$\cdots$
v	0	0	3	3	4	2	2	2	$\cdots$
h	0	0	0	3	3	4	2	2	$\cdots$

La valeur de r est définie par une instance de count qui donne un résultat à chaque instant ; la variable définie à l'intérieur du nœud par c = 0 **fby** n est affichée comme (c_r). L'opérateur **when** échantillonne un flot. Par exemple, r **when** sec ne garde les valeurs de r que lorsque sec est vrai, ce qui donne une séquence commençant par $3, 8, 8$. Dans l'expression h **whenot** sec, l'échantillonnage est fait quand sec est faux. La valeur de t est définie par une instance de count qui n'est activée que lorsque sec est vrai à cause des opérateurs **when** appliqués aux entrées ; la variable interne est affichée comme (c_t). L'opérateur **merge** fusionne deux flots complémentaires. Il est utilisé ici pour garder la valeur de v entre deux calculs. L'opérateur **merge** vient de Lucid Synchrone [21] et SCADE 6. Il remplace l'opérateur `current` de Lustre [7].

3. Elle a pour seul but de simuler l'opérateur d'initialisation -> de Lustre

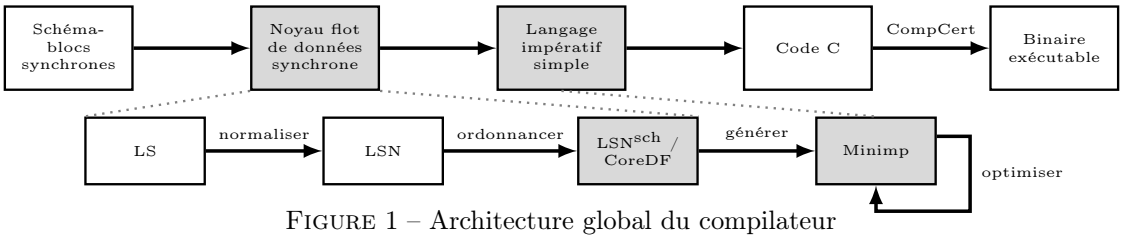

FIGURE 1 – Architecture global du compilateur

1.2. Approche

Nous suivons l'architecture décrite dans [3] et dite « dirigée par les horloges » dont les transformations source à source successives sont présentées dans la figure 1. Tout d'abord, les structures de contrôle comme les automates hiérarchiques sont réduites à un noyau flot de données synchrone [8]. Ce language ('LS') est *normalisé* ('LSN') pour que chaque appel de nœud, chaque **fby** et chaque **merge** soit placé dans une équation distincte. Ensuite, les équations sont *ordonnancées* ('LSN^{sch}') pour respecter leurs interdépendances. Le nœud count de la section 1.1 est normalisé et ordonnancé avec la même sémantique que l'exemple original dans [7, §1.3]. La *génération de code* traduit le code du noyau dans un langage impératif simple qui peut ensuite être transformé en C (ou Ada) pour être compilé par un outil externe. D'une part, ce schéma de compilation est *modulaire* : chaque nœud est compilé en une fonction impérative ; d'autre part, il est *dirigée par les horloges* : chaque équation est typée par une horloge statique qui devient une structure conditionnelle dans le code impératif.

Les encadrés en gris dans la figure indiquent les éléments abordés dans ce papier. Nous nous concentrons sur la compilation d'un noyau flot de données synchrone normalisé et ordonnancé, appellé CoreDF, vers un langage impératif appellé Minimp, qui ressemble aux langages SOL (de SCADE Suite) et Obc [3]. La vérification formelle de cette étape de génération du code n'a jamais été traitée auparavant. Le travail décrit dans le rapport non publié [2] qui influence notre approche ne résout pas ce problème. Le défi principal de la vérification de cette étape est de faire le lien entre les modèles flot de données et impératif et de traiter le schéma de compilation modulaire employé dans le compilateur KCG de SCADE Suite et le compilateur académique Heptagon [11].

Les autres encadrés dans la figure indiquent les éléments qui ont déjà été abordés auparavant ou qui restent à traiter.

La normalisation et l'ordonnancement ont déjà été vérifiés en Coq dans des travaux antérieurs [1,2]. Leur traitement ne pose pas de problème particulier : la normalisation exploite la transparence référentielle du langage source (c'est-à-dire le fait que l'on puisse remplacer toute variable par la définition donnée par son équation) et l'ordonnancement l'indépendance de l'ordre des équations.

La compilation des structures de contrôle n'a pas encore été traitée dans un assistant de preuve et reste un objectif à long terme. La traduction du langage impératif intermédiaire vers un sous-ensemble de C qui pourrait être compilé ensuite par CompCert [4, 17] est en cours.

Contributions Nous présentons la première preuve mécanisée de la passe de compilation qui traduit des équations flot de données synchrones vers un code impératif. Nous montrons comment traiter ce changement de modèle dans un assistant de preuve. Les autres contributions sont les suivantes :

— Une nouvelle construction sémantique combinant les suites infinies des modèles flot de données avec la manipulation progressive des mémoires qui caractérise un modèle impératif ;

— L'identification des invariants, lemmes et structures de preuve nécessaires pour vérifier la correction de la génération de code dans un assistant de preuve ;

— La vérification d'une optimisation importante sur le code cible qui exploite quelques propriétés du langage source et de la fonction de traduction.

Le texte suivant contient des liens, marqués avec un '🐓', vers les sources Coq 8.4 qui se trouvent à l'adresse `https://hal.inria.fr/hal-01403830/file/index.html`.

2. Langages flot de données et impératif

2.1. Langage flot de données : CoreDF

Syntaxe

Nous présentons maintenant la syntaxe et la sémantique de CoreDF. L'exemple de l'introduction est un programme CoreDF. Il y a six catégories syntaxiques : expressions, expressions de contrôle, horloges, équations, nœuds et programmes.

e :=		**expression**
	x	(variable)
	c	(constante)
	$\mathsf{op}\ o\ \overrightarrow{e}$	(opérateur)
	e **when** x	(échantillonnage sur T)
	e **whenot** x	(échantillonnage sur F)
ce :=		**expression de contrôle**
	e	(expression)
	merge $ck\ ce\ ce$	(merge)

ck :=		**horloge**
	base	(horloge de base)
	ck **on** x	(sous-horloge sur T)
	ck **onot** x	(sous-horloge sur F)
eqn :=		**équation**
	$x =_{ck} ce$	(ordinaire)
	$x =_{ck} c$ **fby** e	(délai)
	$x =_{ck} f(\overrightarrow{e})$	(appel de nœud)
$node$:=		**nœud**
	node $f\ (\overrightarrow{i})$ **returns** o **let** $\overrightarrow{eqn}$ **tel**	

Les détails des types et des opérateurs importent peu au problème de génération du code abordé ici : ils sont passés directement à Minimp. Formellement, un nœud associe un nom à des listes de variables d'entrées et d'équations et à une variable de sortie ; un programme est une liste de nœuds.

Par rapport à Lustre, nous remplaçons l'initialisation (`->`) et le délai (`pre`) par les registres initialisés (`fby`). Ce choix évite une analyse d'initialisation. En outre, les entrées d'un appel de nœud doivent toutes être sur une même horloge, contrairement à Lustre où ils peuvent être sur une sous-horloge de l'horloge de la première entrée. Les formalisations antérieures [1, 2] font les mêmes hypothèses, mais, contrairement à nous, ils traitent le merge généralisé et le reset modulaire [15]. Alors que le merge généralisé n'introduit que quelques difficultés techniques mineures, le reset modulaire pose d'importantes questions sémantiques, même si sa compilation n'est pas compliquée. Nous espérons inclure ces deux constructions dans nos travaux à venir. Dernière précision : les nœuds de la version actuelle n'ont qu'une sortie. Ce n'est pas une limitation fondamentale, juste une simplification technique.

Sémantique

Les flots $\mathsf{stream}A$ sont modélisés par des fonctions des entiers naturels vers un domaine de valeurs A. La n-ième valeur d'un flot s est écrite $s_{(n)}$. Le domaine de valeurs inclut normalement une valeur **abs** pour modéliser les 'trous' dans les flots qui constituent une exécution (comme la grille de l'exemple de la section 1.1). Les autres valeurs sont écrites $\langle c \rangle$ où c est soit un booléen b (T ou F), soit un entier.

La sémantique flot de données est définie en deux parties : une partie *combinatoire* pour les valeurs instantanées et une partie *séquentielle* pour les flots. Les jugements combinatoires sont faits relativement à un *environnement instantané* R qui à chaque variable associe une valeur (présente ou

$$(c\,\mathsf{hold}^{\#}\,xs)_{(0)} = c \qquad \frac{xs_{(n)} = \langle c' \rangle}{(c\,\mathsf{hold}^{\#}\,xs)_{(n+1)} = c'} \qquad \frac{xs_{(n)} = \mathbf{abs}}{(c\,\mathsf{hold}^{\#}\,xs)_{(n+1)} = (c\,\mathsf{hold}^{\#}\,xs)_{(n)}}$$

$$\frac{xs_{(n)} = \mathbf{abs}}{(c\,\mathsf{fby}^{\#}\,xs)_{(n)} = \mathbf{abs}} \qquad \frac{xs_{(n)} = \langle c \rangle}{(c\,\mathsf{fby}^{\#}\,xs)_{(n)} = \langle (c\,\mathsf{hold}^{\#}\,xs)_{(n)} \rangle}$$

FIGURE 2 – La définition de l'opérateur $\mathsf{fby}^{\#}$.

absente) et une variable booléenne b qui indique si le nœud englobant est actif dans l'instant, sans quoi toutes les valeurs sont absentes (intuitivement, le code n'est pas exécuté). Il y a des jugements pour les variables ($R \vdash_{\mathsf{var}} x \Downarrow v$), les horloges ($R \vdash_{\mathsf{ck}}{}^{b}\ ck \Downarrow b'$), les expressions ($R \vdash_{\mathsf{e}}^{b} e \Downarrow v$), les expressions avec horloge ($R \vdash_{\mathsf{e}}^{b} e :: ck \Downarrow v$), les expressions de contrôle ($R \vdash_{\mathsf{ce}}{}^{b}\ ce \Downarrow v$) et les expressions de contrôle avec horloge ($R \vdash_{\mathsf{ce}}{}^{b}\ ce :: ck \Downarrow v$).

Les jugements séquentiels sont définis relativement à un flot de valeurs booléennes bk qui donne l'horloge de base du nœud englobant et à un *historique* H qui lie chaque variable à un flot de valeurs. Les valeurs instantanées sont relevés de façon point-à-point en des flots. Par exemple pour les horloges, on a $H \vdash_{\mathsf{ck}}{}^{bk}\ ck \Downarrow bs \triangleq \forall n, H_{(n)} \vdash_{\mathsf{ck}}{}^{bk_{(n)}}\ ck \Downarrow bs_{(n)}$.

Les registres initialisés sont spécifiés par l'opérateur sur des flots $\mathsf{fby}^{\#}$ (figure 2). La définition de $\mathsf{fby}^{\#}$ utilise l'operateur auxiliaire $\mathsf{hold}^{\#}$ qui est initialement égale à la constante c et qui maintient sa valeur actuelle jusqu'à l'instant suivant l'apparition d'une nouvelle valeur sur le flot xs. Cette nouvelle valeur est ensuite maintenue. L'opérateur $\mathsf{fby}^{\#}$ est intégré dans le jugement sémantique de **fby** :

$$\frac{H \vdash_{\mathsf{e}}^{bk} e :: ck \Downarrow ls \qquad H \vdash_{\mathsf{var}} x \Downarrow xs \qquad xs = c\,\mathsf{fby}^{\#}\,ls}{G, H \vdash_{\mathsf{eqn}}{}^{bk}\ x =_{ck} c\ \mathbf{fby}\ e}$$

De la sémantique d'une liste d'équations, on construit la sémantique d'un nœud f dans un programme G qui lie une liste de flots d'entrées xs à un flot de sortie ys :

$$\frac{\left(\begin{array}{l}\mathbf{node}\ f\ (\overrightarrow{i})\ \mathbf{returns}\ o \\ \mathbf{let}\ \ \overrightarrow{eqn}\ \ \mathbf{tel}\end{array}\right) \in G \qquad \begin{array}{c} bk = \mathsf{clock}^{\#}\ \overrightarrow{xs} \\ G, H \vdash_{\mathsf{eqn}}{}^{bk}\ \overrightarrow{eqn} \\ H \vdash_{\mathsf{var}} \overrightarrow{i} \Downarrow \overrightarrow{xs} \\ H \vdash_{\mathsf{var}} o \Downarrow ys \end{array} \qquad \forall n, \overrightarrow{xs}_{(n)} = \mathbf{abs} \Leftrightarrow ys_{(n)} = \mathbf{abs}}{G \vdash_{\mathsf{node}} f(\overrightarrow{xs}, ys)}$$

Le jugement ci-dessus exprime deux propriétés clés. Premièrement, les flots dans un appel de nœud sont seulement activés quand les entrées $\overrightarrow{xs}$ sont présentes. Ceci est garanti en dérivant l'horloge de base bk des valeurs d'entrée : $(\mathsf{clock}^{\#}\,x)_{(n)} \triangleq$ if $x_{(n)} = \mathbf{abs}$ then F else T. Deuxièmement, nous faisons l'hypothèse que les horloges des entrées $\overrightarrow{xs}$ et de la sortie ys sont égales. Un programme doit satisfaire cette hypothèse pour avoir une sémantique. Le contrôle statique des horloges est censé le garantir.

Dans notre sémantique, les horloges servent de contraintes : une équation ne rend une valeur que quand la sémantique de son horloge est vraie. Ceci évite le non-déterminisme dans les équations qui sont 'détachées' de l'environnement comme $\mathsf{x} = 0\ \mathbf{fby}\ \mathsf{x}$ qui n'a pas de contraintes d'horloges et admettrait comme sémantique tout entrelacement de $\langle 0 \rangle$ et d'**abs** sinon.

2.2. Langage impératif : Minimp

Minimp est un sous-ensemble d'Obc [3], un langage impératif simple fait pour représenter un état mémoire et les fonctions de lecture et de modification de cet état. Ses expressions et commandes

lisent et manipulent une paire d'environnements de mémoire. Une *mémoire locale* (*env*) modélise une structure de pile ; elle lie des noms de variables à des valeurs booléennes ou entières. Une *mémoire globale* (*mem*) modélise une mémoire statique et contient deux maps : values des noms de variables aux valeurs et instances des noms de variables aux instances (sous-mémoires des appels de nœuds internes) :

$$\begin{array}{lll} \mathsf{memory}\,(V : \textsc{Set}) & : & \textsc{Set} \\ \mathsf{memory} \quad V & \triangleq & \begin{cases} \mathsf{values} & : \mathsf{ident} \rightharpoonup V \\ \mathsf{instances} & : \mathsf{ident} \rightharpoonup \mathsf{memory}\ V \end{cases} \end{array}$$

Les mémoires (memory) des programmes compilés de CoreDF suivent la structure de l'arbre d'appels de nœuds dans le code source : il y a un élément dans values pour chaque **fby** et un élément dans instances pour chaque appel de nœud. La variable de type V représente le type des valeurs de variables.

Syntaxe

Il y a quatre catégories syntaxiques dans Minimp : les expressions, les commandes, les classes et les programmes.

$$\begin{array}{rll} e & := & \textbf{expression} \\ & \mid\ x & \text{(variable locale)} \\ & \mid\ \mathsf{state}(x) & \text{(variable d'état)} \\ & \mid\ \mathrm{c} & \text{(constante)} \\ & \mid\ \mathsf{op}\ o\ \overrightarrow{e} & \text{(opérateur)} \end{array} \qquad \begin{array}{rll} s & := & \textbf{commande} \\ & \mid\ x := e & \text{(affectation)} \\ & \mid\ \mathsf{state}(x) := e & \text{(affectation d'état)} \\ & \mid\ \textbf{if}\ e\ \textbf{then}\ s\ \textbf{else}\ s & \text{(conditionnelle)} \\ & \mid\ x := f_x.\textbf{step}(\overrightarrow{e}) & \text{(appel à step)} \\ & \mid\ f_x.\textbf{reset}() & \text{(appel à reset)} \\ & \mid\ s\ ;\ s & \text{(composition)} \\ & \mid\ \textbf{skip} & \text{(ne rien faire)} \end{array}$$

$$\begin{array}{rll} cls & := & \textbf{classe} \\ & \mid\ \textbf{class}\ f\ \{ & \\ & \quad \textbf{step}(o)(\overrightarrow{i}) = s & \\ & \quad \textbf{reset} = s\ \} & \end{array}$$

Une classe rassemble un nom de classe (f), les noms des variables d'entrées ($\overrightarrow{i}$), le nom d'une variable de sortie (o), une commande 'step' et une commande 'reset'. Un programme est une liste de classes. L'indice des appels step et reset associe une instance d'un nœud avec sa mémoire ; nous réutilisons le nom de la variable de résultat d'un appel à 'step' pour distinguer plusieurs appels au même nœud. Quelques exemples de programmes Minimp sont présentés dans la section 3.1.

Sémantique

Le jugement sémantique pour une commande s dans le contexte d'un programme *prog* lie les états au début et à la fin de l'instant. Les deux sortes d'affectations évaluent une expression dans une paire de mémoires initiales et contraignent la valeur d'une variable dans une paire de mémoires actualisées.

$$\frac{mem, env \vdash_{\mathsf{e}} e \Downarrow v \qquad env \cup \{x \mapsto v\} = env'}{mem, env \vdash_{\mathsf{st}} x := e \Downarrow mem, env'} \qquad \frac{mem, env \vdash_{\mathsf{e}} e \Downarrow v \qquad mem \cup \{x \mapsto v\} = mem'}{mem, env \vdash_{\mathsf{st}} \mathsf{state}(x) := e \Downarrow mem', env}$$

Un appel à step évalue les expressions des arguments dans les mémoires initiales, cherche le nom de classe dans *prog* et exécute la commande step dans une (sous-) mémoire globale récupérée dans la map instances et une mémoire locale qui associe les variables d'entrée à leurs valeurs. La mémoire d'instance et la variable de résultat sont ensuite mises à jour. Un appel à reset initialise la mémoire globale d'une classe.

3. Génération du code

L'étape principale de la génération de code consiste à traduire un programme flot de données vers un programme impératif pour que les sémantiques des deux programmes s'accordent. Dans cette section, nous présentons la traduction, nous définissons le sens de « s'accorder », nous développons une sémantique intermédiaire qui relie les sémantiques flot de données et impérative, et nous décrivons la preuve de correction.

La traduction distingue les variables définies avec des **fby** de celles définies avec des expressions de contrôle ou des appels de nœud. Pour une liste d'équations associée à un nœud, nous rassemblons les noms de ses variables définies par des **fby** dans un ensemble mems.

La traduction n'est correcte que si les équations d'un nœud sont ordonnancées suivant leurs dépendances. Ceci est formalisé par le prédicat isWellSch :

$$\frac{}{\mathrm{isWellSch}^{args}_{\mathsf{mems}}([])} \qquad \frac{\begin{array}{l}\mathrm{isWellSch}^{args}_{\mathsf{mems}}(eqns) \quad x \in \mathrm{Def}(eqn) \Rightarrow x \notin \mathrm{Def}(eqns) \\ \forall i \in \mathrm{Free}(eqn), \begin{cases} i \notin \mathrm{Def}(eqns) & \text{if } i \in \mathsf{mems} \\ i \in \mathrm{Var}(eqns) \vee i \in args & \text{if } i \notin \mathsf{mems} \end{cases}\end{array}}{\mathrm{isWellSch}^{args}_{\mathsf{mems}}(eqn :: eqns)}$$

Ce prédicat lie une équation *eqn* avec celles qui la suivent dans la liste *eqns*, avec l'idée que les traductions des *eqns* sont exécutées avant celles d'*eqn*. Nous avons trouvé plus commode de travailler avec cet ordre inversé car on peut facilement lier les variables libres dans l'équation à la tête de la liste avec les variables définies ou non dans sa queue. Chaque variable ne peut être définie qu'une fois dans un nœud : x ne peut pas être déjà défini dans *eqns*. Les variables libres du membre droit de *eqn* définies par des **fby** ne doivent pas se trouver avant x, c'est-à-dire, dans *eqns*, car il faut qu'elles soient lues avant d'être mises à jour avec leurs futures valeurs. Inversement, toute autre variable libre doit être écrite avant d'être lue, donc soit être une entrée, soit être définie dans *eqns* par une expression de contrôle ou un appel de nœud.

3.1. Traduction

La traduction transforme une liste de nœuds flot de données en une liste de classes impératives. Nos définitions suivent celles de [3]. La fonction trexp traduit les expressions. Les variables flot de données sont traduites par des variables impératives globales ou locales :

$$\begin{array}{lll} \mathsf{var}\,(x : \mathsf{ident}) & : & \mathsf{exp} \\ \mathsf{var} \quad x & \triangleq & \mathsf{if}\ x \in \mathsf{mems}\ \mathsf{then}\ \mathbf{state}(x)\ \mathsf{else}\ x \end{array}$$

Les constantes et les opérateurs sont propagés directement et les expressions e **when** c sont simplement remplacées par e.

$$\begin{array}{llll} \mathsf{trexp} & (e : \mathsf{exp}) & : & \mathsf{iexp} \\ \mathsf{trexp} & x & \triangleq & \mathsf{var}(x) \\ \mathsf{trexp} & c & \triangleq & c \\ \mathsf{trexp} & (\mathsf{op}\ o\ \overrightarrow{e}) & \triangleq & \mathsf{op}\ o\ (\mathrm{map}\ \mathsf{trexp}\ \overrightarrow{e}) \\ \mathsf{trexp} & (\ \mathbf{when}\ e\ x) & \triangleq & \mathsf{trexp}\ e \\ \mathsf{trexp} & (\ \mathbf{whenot}\ e\ x) & \triangleq & \mathsf{trexp}\ e \end{array}$$

Les expressions de contrôle sont traduites par un arbre de structures conditionnelles dans lequel chaque feuille est une affectation à la même variable (nécessairement locale) :

$$\begin{array}{llll} \mathsf{trcexp}\,(x : \mathsf{ident}) & (ce : \mathsf{cexp}) & : & \mathsf{stmt} \\ \mathsf{trcexp} \quad x & (\mathbf{merge}\ y\ t\ f) & \triangleq & \mathbf{if}\ (\mathsf{var}\ y)\ \mathbf{then}\ (\mathsf{trcexp}\ x\ t)\ \mathbf{else}\ (\mathsf{trcexp}\ x\ f) \\ \mathsf{trcexp} \quad x \quad e & & \triangleq & x := \mathsf{trexp}\ e \end{array}$$

Dans une traduction dirigée par les horloges, *les horloges du langage source sont transformées en structures de contrôle dans le langage cible* [3]. Ce principe est concrétisé par une fonction qui enveloppe une commande s dans des structures conditionnelles suivant la structure d'une horloge ck :

$$
\begin{array}{lll}
\mathsf{ctrl}\ (ck : \mathsf{clock})\ (s : \mathsf{stmt}) : & \mathsf{stmt} & \\
\mathsf{ctrl} \quad \textbf{base} & s & \triangleq\ s \\
\mathsf{ctrl}\ \ (ck\ \textbf{on}\ x) & s & \triangleq\ \mathsf{ctrl}\ ck\ (\textbf{if}\ (\mathsf{var}\ x)\ \textbf{then}\ s\ \textbf{else skip}) \\
\mathsf{ctrl}\ \ (ck\ \textbf{onot}\ x) & s & \triangleq\ \mathsf{ctrl}\ ck\ (\textbf{if}\ (\mathsf{var}\ x)\ \textbf{then skip else}\ s)
\end{array}
$$

Ces fonctions suffisent pour traduire une équation en une affectation gardée d'une variable locale, d'un appel de fonction step ou d'une variable globale :

$$
\begin{array}{lll}
\mathsf{treqn}\ (eqn : \mathsf{equation}) & : & \mathsf{stmt} \\
\mathsf{treqn} \quad (x =_{ck} ce) & \triangleq & \mathsf{ctrl}\ ck\ (\mathsf{trcexp}\ x\ ce) \\
\mathsf{treqn}\ (x =_{ck} f(es)) & \triangleq & \mathsf{ctrl}\ ck\ (x := f_x.\textbf{step}(\mathsf{map}\ \mathsf{trexp}\ es)) \\
\mathsf{treqn}\ (x =_{ck} \mathrm{c}\ \textbf{fby}\ e) & \triangleq & \mathsf{ctrl}\ ck\ (\textbf{state}(x) := \mathsf{trexp}\ e)
\end{array}
$$

Les équations ordinaires deviennent des affectations dans la mémoire locale. Les appels de nœuds deviennent des appels à step qui mettent à jour la mémoire locale et la mémoire globale. Les **fby** deviennent des affectations dans la mémoire globale.

Une liste d'équations satisfaisant isWellSch est traduite en une séquence d'affectations gardées qui devient le corps de la fonction step correspondante :

$$
\begin{array}{l}
\mathsf{treqns}\ (eqns : \mathsf{list\ equation}) : \mathsf{stmt} \\
\mathsf{treqns}\ eqns\ \triangleq\ \mathsf{foldl}\ (\lambda\ acc\ eqn.\ \mathsf{treqn}\ eqn\ ;\ acc)\ eqns\ \textbf{skip}
\end{array}
$$

Le corps d'une fonction reset est également une séquence d'affectations globales (pour les **fby**) et d'appels à reset (pour les appels de nœud).

Exemples

Le nœud count de la section 1.1 (avec la liste d'équations inversées) est traduit par la commande step suivante :

```
n:= if (state(f) or restart) then ini else (state(c) + inc);
state(f) := false;
state(c) := n;
skip
```

et la commande reset suivante :

```
state(f) := true; state(c) := 0; skip
```

De même, la traduction de avgvelocity donne le code suivant :

```
r := count_r.step(0, delta, false);
if sec then t := count_t.step(1, 1, false) else skip;
if sec then v := (r / t) else v := h;
state(h) := v;
skip
```

Dans cet exemple, il y a deux instances du nœud count. On appelle donc deux fois la fonction count.step. Le premier appel agit sur la mémoire globale instances r et son résultat est transféré dans la variable locale r. Le nom r est utilisé pour spécifier sans ambiguïté à la fois la mémoire d'instance et la variable de résultat. Les instances sont spécifiées de la même façon dans la fonction de reset :

```
count_r.reset(); count_t.reset(); state(h) := 0 ; skip
```

3.2. Correction

Dans le contexte d'un programme flot de données, la sémantique d'un nœud lie les flots d'entrée à un flot de sortie. Le code impératif produit par la traduction du programme doit satisfaire une propriété fondamentale : répéter l'exécution avec les valeurs successives des flots d'entrée génère les valeurs successives du flot de sortie. Nous formalisons les exécutions répétées par un prédicat step :

$$\frac{\emptyset, \emptyset \vdash_{\mathsf{st}} f_r.\mathbf{reset}() \Downarrow env, mem}{\mathrm{step}(0, r, f, \overrightarrow{xs}, env, mem)} \qquad \frac{\mathrm{step}(n, r, f, \overrightarrow{xs}, env, mem) \quad env, mem \vdash_{\mathsf{st}} r := f_r.\mathbf{step}(\overrightarrow{xs}_{(n)}) \Downarrow env', mem'}{\mathrm{step}(n+1, r, f, \overrightarrow{xs}, env', mem')}$$

Ce prédicat 'exécute' n fois la commande step de f à partir d'un environnement créé par la commande reset et rend les environnements env' et mem', en passant les valeurs appropriées des entrées $\overrightarrow{xs}$ à chaque instant.

Nous imposons que G soit bien formé, noté $\mathrm{Welldef}(G)$, ce qui signifie que : (1) les équations de chaque nœud satisfont isWellSch, (2) elles ne redéfinissent pas les entrées, (3) elles définissent la sortie, (4) elles n'instancient que des nœuds définis, (5) et ce, de façon non circulaire. Nous pouvons alors énoncer et démontrer la propriété fondamentale que doit satisfaire la fonction de traduction trans.

Proposition 1. *Soit G un programme CoreDF bien formé, tel que* $\mathrm{Welldef}(G)$*, qui contient un nœud f avec la sémantique $G \vdash_{\mathsf{node}} f(\overrightarrow{xs}, ys)$. La traduction de G en un programme impératif qui itère n fois la commande step de f avec les valeurs successives de $\overrightarrow{xs}$ donne un environnement contenant la n-ième valeur de ys si et seulement si cette dernière est présente :*

$$\exists env\ mem, \quad \mathrm{step}(n+1, r, f, \overrightarrow{xs}, env, mem) \\ \wedge\ \forall o,\ ys_{(n)} = \langle o \rangle \iff env(r) = o.$$

Cette proposition exprime ce qu'il faut pour que la traduction soit correcte : la relation entre $\overrightarrow{xs}$ et ys définie par f est respectée par les exécutions répétées du code impératif généré. Le fait que env et mem existent veut dire que le code impératif a une sémantique : chaque pas termine. Le déterminisme de la sémantique de Minimp garantit que les sorties antérieures sont également correctes. Les éléments principaux de la preuve de cette proposition sont donnés sur les pages suivantes, à savoir un modèle intermédiaire (lemme 1), la preuve qu'un pas d'une commande step est correct (proposition 2) et, pour cette dernière, le lemme clef sur la correction de la compilation des équations d'un nœud (lemme 2).

La proposition 1 est trop faible pour être démontrée directement par récurrence sur n, parce qu'elle ne spécifie rien sur la mémoire globale. Remarquons que le programme généré manipule un arbre d'éléments mem qui reflète la structure des appels de nœuds dans le programme original. Pour un appel donné, il y a dans mem.instances un sous-arbre pour chaque appel interne et dans mem.values un registre pour chaque équation **fby**. Il faut donc définir un invariant qui lie la séquence de valeurs prise par chaque flot **fby** aux valeurs lues et écrites successivement dans le registre associé. Pour l'exécution de l'exemple de la section 1.1, les flots du deuxième appel de nœud, n et c = 0 **fby** n, sont affichés dans les deux premiers rangs ci-dessous.

n			1		2	3		⋯
c			0		1	2		⋯
state(c)	0	0	0 ; 1	1	1 ; 2	2 ; 3	3	⋯

Le troisième rang montre les valeurs dans le registre créé pour c, au début d'un pas du programme impératif et à la fin de ce pas lorsque le registre est écrit. Dans les itérations où n et c sont absents, un pas d'avgvelocity est exécuté, mais on n'exécute pas le deuxième count (car sec est faux) et **state**(c) garde sa valeur. Le premier 0 est placé dans **state**(c) par la commande reset de count. À chaque pas de cet appel de nœud, la valeur qui est déjà dans **state**(c) sert à calculer une valeur pour n qui est ensuite écrite dans **state**(c) pour le prochain appel de step.

3.3. Sémantique flot de données intermédiaire avec mémoire

Le jugement $G \vdash_{\mathsf{node}} f(\overrightarrow{xs}, ys)$ ne donne que le comportement entrées/sortie d'un nœud. Les valeurs des flots internes, comme $\mathsf{c_r}$ et $\mathsf{c_t}$ dans l'exemple, sont cachées. En outre, à la différence des registres impératifs, ces flots n'ont pas forcément une valeur à chaque instant : ils peuvent être absents. Nous traitons les deux problèmes en introduisant un nouveau jugement sémantique $G \vdash_{\mathsf{mnode}} f(\overrightarrow{xs}, M, ys)$. Ce jugement révèle un arbre de memory M qui est isomorphe à celui du code traduit, mais où la variable de type V est instanciée avec des flots de constantes. Nous reprenons les règles de la sémantique instantanée des expressions. Les règles pour les nœuds, les équations ordinaires et les appels de nœud sont redéfinis presque trivialement et cette dernière doit chercher le sous-arbre M' dans $mem.\mathsf{instances}$ qui correspond à cet appel. La nouvelle règle de **fby** est la suivante :

$$\frac{\begin{array}{cc} ms = M.\mathsf{values}(x) & H \vdash_{\mathsf{e}}^{bk} e :: ck \Downarrow ls \\ ms_{(0)} = v_0 & H \vdash_{\mathsf{var}} x \Downarrow xs \end{array} \qquad \forall n, \begin{cases} ms_{(n+1)} = ms_{(n)} \wedge xs_{(n)} = \mathbf{abs} & \text{if } ls_{(n)} = \mathbf{abs} \\ ms_{(n+1)} = v \wedge xs_{(n)} = \langle ms_{(n)} \rangle & \text{if } ls_{(n)} = \langle v \rangle \end{cases}}{G, M, H \vdash_{\mathsf{meqn}}^{bk} x =_{ck} v_0 \textbf{ fby } e}$$

Le flot ms associé à la mémoire de la variable x est récupéré dans $M.\mathsf{values}$. Au départ, il prend la valeur v_0. Par la suite, sa valeur est maintenue si le flot en argument est absent ; sinon sa prochaine valeur est la valeur qui est présente sur ce flot. Le flot xs associé à x est absent si l'argument l'est et présent avec la valeur actuelle de la mémoire sinon. Le comportement de ce modèle ressemble fortement à celui du code généré.

À partir du fait qu'un nœud possède une sémantique, on peut montrer qu'il possède également une sémantique avec mémoire. En pratique, cela permet de raisonner sur les valeurs des mémoires dans des preuves de propriétés exprimées uniquement avec la sémantique flots de données classique. Formellement, nous démontrons le lemme suivant.

Lemme 1. *Soit un programme G tels que* $\mathrm{Welldef}(G)$ *et $G \vdash_{\mathsf{node}} f(\overrightarrow{xs}, ys)$, alors il y a une mémoire M qui satisfait $G \vdash_{\mathsf{mnode}} f(\overrightarrow{xs}, M, ys)$.*

Comme la mémoire M d'un nœud est exposée, il est possible de la lier directement à une mémoire globale mem du programme impératif à l'instant n. Nous formalisons cette relation dans le prédicat $\mathrm{MemCorres}_n(M, mem)$ où G est laissé implicite. Ce prédicat exige que les équations pour chaque nœud de G satisfont MemCorresEqn. C'est trivialement le cas pour les équations ordinaires. Le prédicat est défini pour les appels de nœud par la règle suivante :

$$\frac{\begin{array}{c} M_x = M.\mathsf{instances}(x) \\ mem_x = mem.\mathsf{instances}(x) \quad \mathrm{MemCorres}_n(M_x, mem_x) \end{array}}{\mathrm{MemCorresEqn}_n(M, mem, x =_{ck} f(e))}$$

et pour les **fby** par la règle suivante :

$$\frac{ms_x = M.\mathsf{values}(x) \qquad mem.\mathsf{values}(x) = (ms_x)_{(n)}}{\mathrm{MemCorresEqn}_n(M, mem, x =_{ck} v_0 \textbf{ fby } e)}$$

Avec ces définitions, nous sommes en mesure d'énoncer et de démontrer l'invariant inductif principal de notre preuve de correction.

3.4. Preuve de correction

La correction de trans est démontrée par trois niveaux imbriqués de raisonnement par récurrence : sur les instants n, sur les appels de nœuds G et sur les équations à l'intérieur d'un nœud *eqns*. Il y a ensuite deux distinctions de cas imbriquées : sur les trois types d'équations et sur le fait que chacun peut être exécuté ou non. La correction de la n-ième exécution de la fonction step générée est énoncée pour les programmes bien formés en utilisant la sémantique avec mémoire :

Proposition 2. *Soit G un programme tel que* Welldef(G) *et f un nœud de G qui vérifie $G \vdash_{\mathsf{mnode}} f(\overrightarrow{xs}, M, ys)$. Si mem est une mémoire globale satisfaisant* MemCorres$_n(M, mem)$ *à l'instant n, alors il existe une mémoire globale mem' telle que*

$$mem \vdash_{\mathsf{step}} r := f_r.\textbf{\textit{step}}(\overrightarrow{xs}_{(n)}) \Downarrow mem', ys_{(n)} \\ \wedge \text{MemCorres}_{n+1}(M, mem'),$$

où la commande step de f provient du programme généré.

Cette proposition est centrale dans la preuve de la proposition 1, où elle fournit l'invariance de correspondance des mémoires : le code impératif met à jour correctement la mémoire globale.

Un autre lemme montre MemCorres$_0(M, mem_0)$ pour la mémoire mem_0 créée par l'exécution de la commande reset de f. Ensemble, ces deux faits permettent un raisonnement par récurrence sur n dans la proposition 1. Un lemme supplémentaire est nécessaire dans la preuve de la proposition 2.

Lemme 2. *Soient $eqns_{all}$ une liste d'équations, eqns un suffixe de cette liste—c'est-à-dire, $\exists eqns', eqns_{all} = eqns' +\!\!+ eqns$—et* mems *l'ensemble de variables dans $eqns_{all}$ définies avec un* fby*. Soient G un programme qui vérifie* Welldef(G)*, H un environnement de flots, M une mémoire flot de données et bk un flot d'horloge tels que $G, H, M \vdash_{\mathsf{meqns}}^{bk} eqns_{all}$. On impose en outre* isWellSch$^{args}_{mems}(eqns)$ *pour un ensemble de noms de variable args et que les entrées ne doivent pas être redéfinies par les équations : $\forall i \in args, i \notin \text{Def}(eqns_{all})$.*

Soit env une mémoire locale qui contient une valeur pour chaque entrée égale à la valeur de cette variable dans H à l'instant n (et rien de plus),

$$\forall i \in args, H_{(n)} \vdash_{\mathsf{var}} i \Downarrow \langle c \rangle \Leftrightarrow env(i) = c,$$

et soit mem une mémoire globale qui s'accorde avec M pour toutes les variables définies par $eqns_{all}$, c'est-à-dire qui satisfait MemCorresEqns$_n(M, mem, eqns_{all})$.

Alors, il est vrai que :

1. *l'exécution de la traduction des équations eqns dans les deux environnements rend une mémoire locale env' et une mémoire globale mem',*

$$env, mem \vdash_{\mathsf{st}} \textbf{\textit{treqns}}\ eqns \Downarrow env', mem',$$

2. *les valeurs calculées dans env' s'accordent avec celles de H à l'instant n, $\forall x \in \text{Def}(eqns) \setminus$ mems,*

$$H_{(n)} \vdash_{\mathsf{var}} x \Downarrow \langle c \rangle \Leftrightarrow env'(x) = c, \text{ et}$$

3. *mem' s'accorde avec M à l'instant $n+1$, pour les variables définies dans eqns,*

$$\text{MemCorresEqns}_{n+1}(M, mem', eqns).$$

Pour l'utilisation du lemme dans la preuve de la proposition 2, $eqns_{all}$ et *eqns* sont tous les deux instanciés avec la liste d'équations du nœud (c'est-à-dire, $eqns' = []$). La preuve formelle n'est pas courte. Le lecteur qui désire en savoir plus peut consulter les sources Coq. Le cas le plus subtil est l'appel de nœud lorsque son horloge est fausse. Comme la fonction impérative associée n'est pas exécutée, nous ne pouvons pas nous servir de l'hypothèse de récurrence sur G. Il faut plutôt remarquer que la mémoire de l'instance ne change pas et la formulation du modèle intermédiaire joue un rôle crucial pour cela.

4. Optimisation

Pour certains programmes, la fonction de traduction génère un code avec trop de structures conditionnelles, une passe d'optimisation est donc normalement appliquée pour fusionner les structures conditionnelles adjacentes. Pour avgvelocity, par exemple, le code optimisé de la fonction step est le suivant (cf. page 8) :

```
r := count_r.step(0, delta, false);
if sec then (t := count_t.step(1, 1, false); v := (r / t))
       else v := h;
state(h) := v
```

Cette optimisation est plus efficace si l'ordonnanceur statique place ensemble les équations qui ont des horloges similaires. Nous définissons l'optimisation par deux fonctions. La première fonction divise simplement une composition séquentielle en deux parties :

$$\begin{array}{lll} \mathsf{fuse}\,(s\!:\!\mathsf{stmt}) & : & \mathsf{stmt} \\ \mathsf{fuse}\,(s_1\ ;\ s_2) & \triangleq & \mathsf{zip}\ s_1\ s_2 \\ \mathsf{fuse}\quad s & \triangleq & s \end{array}$$

La deuxième fonction intègre itérativement les commandes de la deuxième partie dans la première partie et exécute l'optimisation de façon récursive :

$$\begin{array}{llll} \mathsf{zip} & (s_1\!:\!\mathsf{stmt}) & (s_2\!:\!\mathsf{stmt}) & : \mathsf{stmt} \\ \mathsf{zip}\,(\textbf{if}\ e\ \textbf{then}\ s_1\ \textbf{else}\ s_2) & (\textbf{if}\ e\ \textbf{then}\ t_1\ \textbf{else}\ t_2) & \triangleq & \textbf{if}\ e\ \textbf{then}\ \mathsf{zip}\ s_1\ t_1\ \textbf{else}\ \mathsf{zip}\ s_2\ t_2 \\ \mathsf{zip} & (s_1\ ;\ s_2) & t & \triangleq s_1\ ;\ (\mathsf{zip}\ s_2\ t) \\ \mathsf{zip} & s & (t_1\ ;\ t_2) & \triangleq \mathsf{zip}\ (\mathsf{zip}\ s\ t_1)\ t_2 \\ \mathsf{zip} & s & \textbf{skip} & \triangleq s \\ \mathsf{zip} & \textbf{skip} & t & \triangleq t \\ \mathsf{zip} & s & t & \triangleq s\ ;\ t \end{array}$$

Alors que la première règle de zip ne respecte pas la sémantique de $s_1\ ;\ s_2$ en général (considérez **if** x **then** x := false **else** x := true ; **if** x **then** $\cdots$ **else** $\cdots$.), elle le fait bien pour le code produit par la fonction trans. Pour le démontrer, il faut caractériser la propriété qui assure la correction de l'optimisation, démontrer que le code produit par trans la satisfait et faire de même pour les transformations successives de fuse. Nous définissons pour cela un prédicat Fusible sur les commandes qui a une seule règle non triviale :

$$\frac{\mathrm{Fusible}(s_1) \qquad \mathrm{Fusible}(s_2) \qquad \forall x \in \mathrm{Free}(e), \neg\mathrm{MayWrite}(x, s_1) \wedge \neg\mathrm{MayWrite}(x, s_2)}{\mathrm{Fusible}(\textbf{if}\ e\ \textbf{then}\ s_1\ \textbf{else}\ s_2)}$$

où $\mathrm{MayWrite}(x, s)$ est vrai si et seulement si s contient une affectation à x ou à $\textbf{state}(x)$.

Pour les équations qui satisfont isWellSch, le code généré pour les formes $x =_{ck} ce$ et $x =_{ck} f(\overrightarrow{e})$ satisfait Fusible, car les variables doivent être écrites avant d'être lues. Autrement dit, x ne peut jamais être libre dans ck ou dans les **merge** de ce à partir desquels les structures de contrôle sont générées. Par contre, ce n'est pas le cas pour $x =_{ck} c\ \textbf{fby}\ e$. Plutôt que de proposer un prédicat plus compliqué (les variables d'état écrites à gauche d'un '**;**' ne sont jamais libre à sa droite), qui rendrait plus compliquées les preuves autour de fuse, nous démontrons qu'une variable x ne peut jamais être libre sur sa propre horloge dans un nœud bien cadencé. Quelques détails techniques sont nécessaires pour traiter le cas général, mais pour l'essentiel, x ne peut pas avoir l'horloge $ck\ \textbf{on}\ x$, puisque de telles horloges ne sont bien formées que si x a l'horloge ck.

Pour démontrer que fuse respecte Fusible, nous définissons une relation d'équivalence observationnelle $s_1 \approx s_2$ sur les commandes qui transforment des états de mémoire de manière identique puis nous l'étendons à la relation 'conditionnelle' comme suit.

$$s_1 \approx_{\text{fuse}} s_2 \triangleq s_1 \approx s_2 \wedge \text{Fusible}(s_1) \wedge \text{Fusible}(s_2).$$

Nous démontrons quelques lemmes de congruence pour $\approx_{\text{fuse}}$ qui nous permettent de démontrer que Fusible(s) implique fuse $s \approx s$.

5. Travaux connexes

Les travaux connexes peuvent être divisés en deux catégories : ceux qui se concentrent sur la formalisation des sémantiques des langages dans un assistant de preuve et ceux qui abordent plus directement la correction de la compilation. Dans cette brève revue, nous nous focalisons sur les travaux qui, comme nous, traitent des particularités des langages synchrones. La correction des compilateurs d'usage général reste cependant un sujet proche.

Plusieurs langages synchrones ont été formalisés dans un assistant de preuve : un sous-ensemble de Lustre en Coq avec des types coinductifs [9] ; un langage à la Esterel dans HOL avec un accent sur la preuve des programmes [23] ; un plongement léger de Lucid Synchrone dans Coq avec une sémantique manipulant des suites infinis avec présence, absence et échec, et qui exprime les contraintes d'horloges dans le typage de Coq avec pour garantie que les fonctions sont totales [5] ; une sémantique dénotationnelle des réseaux de Kahn dans Coq [19]. Les travaux qui traitent un compilateur synchrone dans un assistant de preuve sont restés dans le domaine du modèle flot de données : un rapport non publié [12] sur le compilateur Scade 3 se concentre sur les définitions de la sémantique et des horloges ; un des résultats du projet Gene-Auto est la preuve de correction de l'ordonnancement des équations pour un générateur de code C pour Simulink [16]. Aucun de ses travaux ne traite de la génération de code impératif à partir de programmes synchrones flot de données.

La validation de traduction est une technique complémentaire à la vérification d'un compilateur. Elle a été appliquée aux langages synchrones il y a deux décennies [20] et plus récemment à un sous-ensemble de Simulink et son compilateur optimisant RTW [22]. Il existe également un travail en cours sur un compilateur Signal existant [18]. Cette approche est attrayante parce qu'elle découple la compilation de la preuve avec plusieurs avantages pratiques, même s'il est généralement nécessaire d'adapter le compilateur pour fournir plus d'information au validateur. La validation de traduction peut donner des garanties formelles très fortes si le validateur est vérifié [17], mais ce n'est pas le cas pour le travail cité ci-dessus.

Une des motivations de la vérification d'un compilateur Lustre est d'assurer que les propriétés vérifiées sur les modèles sont aussi vraies pour le code généré. Une approche alternative est de compiler les propriétés et de les vérifier à nouveau sur le code exécutable [10]. Cette idée est intéressante, mais elle présente deux inconvénients : (1) il faut faire confiance à la vérification des propriétés et à leur compilation, (2) et une vérification peut réussir sur le modèle, mais échouer sur le code généré.

6. Discussion

Nous avons présenté une formalisation dans Coq d'une version normalisée de Lustre, une fonction de génération de code qui fait le lien entre modèles flot de données et impératif et une preuve de sa correction. Les définitions des deux langages et la fonction de traduction sont adaptées directement de travaux précédents [1–3], mais deux détails méritent d'être notés. Premièrement, nous définissons les flots comme des fonctions sur les entiers naturels et non comme des listes finies. Par rapport aux formalisations précédentes [1, 2], cela évite quelques conditions de bord sans intérêt et des obligations

de preuves sur les longueurs relatives des listes. Mais cela nous oblige à travailler avec des objets infinis. En particuler, notre définition en cinq règles de **fby** (la figure 2) est plus complexe que sa définition habituelle à deux règles sur les suites ([2, figure 9]). Il est intéressant de noter que ce défaut est corrigé dans la sémantique avec mémoire où les éléments de M pour les **fby** correspondent aux indices utilisés dans les preuves sur papier [2]. Deuxièmement, la preuve de la traduction jusqu'au code impératif révèle l'importance pratique de la forme des définitions des nœuds et de leurs appels. Il a fallu notamment énoncer et démontrer plusieurs propriétés d'invariance de mémoire qui dépendent du fait que les horloges des équations à l'intérieur d'un nœud sont des sous-horloges des horloges des entrées du nœud.

La nécessité d'un modèle sémantique intermédiaire ne nous apparaissait pas évidente au début. Auger [1, chapitres 8 et 9] se débat avec le problème du passage des flots de données au modèle impératif en introduisant une nouvelle passe de compilation ('LSNI'), mais sans les preuves de simulation correspondantes. À l'inverse, nous introduisons une nouvelle sémantique flot de données que nous validons avec la sémantique de référence (lemme 1). Cette sémantique facilite la formulation et la preuve des lemmes de correction (proposition 2 et lemme 2) et n'implique aucun changement dans l'implémentation du générateur de code. En outre, cette sémantique intermédiaire englobe les deux autres modèles : en supprimant les mémoires, on retrouve la sémantique flot de données ; en prenant un « cliché » instantané, on retrouve la sémantique impérative. Elle est aussi une alternative valable, presque opérationnelle, pour la formalisation d'un langage flot de données (normalisé et ordonnancé).

En plus de la sémantique intermédiaire, il nous a fallu plusieurs itérations pour trouver les définitions précises des prédicats et lemmes présentés dans cet article. Les formes des prédicats isWellSch et MemCorres sont typiques ; il est important, pour le travail pratique dans un assistant de preuve, de trouver des définitions simples qui fonctionnent bien ensemble. De même, la structure du lemme de correction principal avec ses trois récurrences imbriquées (temps, appels de nœud et équations) est le résultat d'un long processus de réflexion et d'expérimentation.

L'inclusion de l'optimisation fuse est presque obligatoire dans l'approche de compilation « dirigée par les horloges ». Sa vérification dans le cadre de Minimp illustre deux avantages de notre développement : il permet d'exploiter les propriétés du langage source et de la fonction de traduction pour justifier une optimisation sur le code impératif. Le langage impératif s'est avéré utile pour l'implémentation et la vérification de cette optimisation. Il serait possible de formaliser fuse, Fusible et d'effectuer la preuve de fuse $s \approx s$ directement dans le langage Clight de CompCert. Cependant, il y a plus de cas à traiter et son modèle de mémoire est beaucoup plus riche que le nôtre : la tâche serait donc plus ardue. Même si nous enrichissons progressivement notre langage avec les types, opérateurs et fonctions externes de Clight, nous ne nous attendons pas à revisiter ce choix.

Notre travail continue en suivant l'architecture globale décrite dans la figure 1. En particulier, nous n'avons pas encore vérifié les systèmes de type et d'horloge. Cela permettrait de dériver plutôt que de supposer les propriétés de synchronisation qui sont nécessaires pour la preuve de correction. Le développement du lien avec CompCert impliquera l'adaptation de notre traitement des types et des opérateurs ainsi que la compilation de nos arbres de mémoire en enregistrements imbriqués.

Remerciements Nous souhaitons remercier Adrien Guatto pour des discussions intéressantes et ses commentaires pertinents ainsi que Guillaume Baudart, Louis Mandel et les rapporteurs des JFLA pour leurs relectures attentives. Ce travail a été soutenu par le projet ITEA 3 14014 ASSUME et le programme Emergence(s) de la Ville de Paris.

Références

[1] C. Auger : *Compilation certifiée de SCADE/LUSTRE.* Thèse de doctorat, Univ. Paris Sud 11, avr. 2013.

[2] C. Auger, J.-L. Colaço, G. Hamon et M. Pouzet : A formalization and proof of a modular Lustre code generator. En préparation, 2016.

[3] D. Biernacki, J.-L. Colaço, G. Hamon et M. Pouzet : Clock-directed modular code generation for synchronous data-flow languages. *In LCTES'08*, p. 121–130. ACM, juin 2008.

[4] S. Blazy, Z. Dargaye et X. Leroy : Formal verification of a C compiler front-end. *In FM'06*, vol. 4085 de *Lecture Notes in Comp. Sci.*, p. 460–475. Springer, août 2006.

[5] S. Boulmé et G. Hamon : Certifying synchrony for free. *In LPAR'01*, vol. 2250 de *Lecture Notes in Comp. Sci.*, p. 495–506. Springer, déc. 2001.

[6] P. Caspi, A. Curic, A. Maignan, C. Sofronis, S. Tripakis et P. Niebert : From Simulink to SCADE/Lustre to TTA : a layered approach for distributed embedded applications. *In LCTES'03*, p. 153–162. ACM, 2003.

[7] P. Caspi, D. Pilaud, N. Halbwachs et J. Plaice : LUSTRE : A declarative language for programming synchronous systems. *In POPL'87*, p. 178–188. ACM, jan. 1987.

[8] J.-L. Colaço, B. Pagano et M. Pouzet : A conservative extension of synchronous data-flow with state machines. *In EMSOFT'05*, p. 173–182. ACM, sept. 2005.

[9] S. Coupet-Grimal et L. Jakubiec : Hardware verification using co-induction in Coq. *In TPHOLs'99*, vol. 1690 de *Lecture Notes in Comp. Sci.*, p. 91–108. Springer, sept. 1999.

[10] A. Dieumegard, P.-L. Garoche, T. Kahsai, A. Taillar et X. Thirioux : Compilation of synchronous observers as code contracts. *In SAC'15*, p. 1933–1939. ACM, avr. 2015.

[11] L. Gérard, A. Guatto, C. Pasteur et M. Pouzet : A modular memory optimization for synchronous data-flow languages : application to arrays in a Lustre compiler. *In LCTES'12*, p. 51–60. ACM, juin 2012.

[12] E. Gimenez et E. Ledinot : Certification de SCADE V3. Rapport final du projet GENIE II, Verilog SA, jan. 2000.

[13] G. Hagen et C. Tinelli : Scaling up the formal verification of Lustre programs with SMT-based techniques. *In FMCAD'08*, p. article 15. IEEE, nov. 2008.

[14] N. Halbwachs, F. Lagnier et C. Ratel : Programming and verifying real-time systems by means of the synchronous data-flow language LUSTRE. *IEEE Trans. Software Engineering*, 18(9):785–793, sept. 1992.

[15] G. Hamon et M. Pouzet : Modular resetting of synchronous data-flow programs. *In PPDP'00*, p. 289–300. ACM, sept. 2000.

[16] N. Izerrouken, X. Thirioux, M. Pantel et M. Strecker : Certifying an automated code generator using formal tools : Preliminary experiments in the GeneAuto project. *In ERTS'08*. Société des Ingénieurs de l'Automobile, jan./fév. 2008.

[17] X. Leroy : Formal verification of a realistic compiler. *Comms. ACM*, 52(7):107–115, 2009.

[18] V.-C. Ngo, J.-P. Talpin, T. Gautier, L. Besnard et P. Le Guernic : Modular translation validation of a full-sized synchronous compiler using off-the-shelf verification tools. *In SCOPES'15*, p. 109–112. ACM, juin 2015.

[19] C. Paulin-Mohring : A constructive denotational semantics for Kahn networks in Coq. *In From Semantics to Computer Science : Essays in Honour of Gilles Kahn*, p. 383–413. CUP, 2009.

[20] A. Pnueli, M. Siegel et O. Shtrichman : Translation validation for synchronous languages. *In ICALP'98*, vol. 1443 de *Lecture Notes in Comp. Sci.*, p. 235–246. Springer, 1998.

[21] M. Pouzet : *Lucid Synchrone, version 3. Tutorial and reference manual.* Université Paris-Sud, LRI, avr. 2006.

[22] M. Ryabtsev et O. Strichman : Translation validation : From Simulink to C. *In CAV'09*, vol. 5643 de *Lecture Notes in Comp. Sci.*, p. 696–701. Springer, juin 2009.

[23] K. Schneider : Embedding imperative synchronous languages in interactive theorem provers. *In ACSD'01*, p. 143–154. IEEE Computer Society, juin 2001.