Arguments cadencés dans un compilateur Lustre vérifié

Timothy Bourke, Marc Pouzet

Arguments cadencés dans un compilateur Lustre vérifié

Timothy Bourke[1,2] et Marc Pouzet[3,2,1]

[1] Inria Paris
[2] École normale supérieure, PSL University
[3] Sorbonne Universités, UPMC Univ. Paris 06

Résumé

Lustre est un langage synchrone pour programmer des systèmes avec des schémas-blocs desquels un code impératif de bas niveau est généré automatiquement. Des travaux récents utilisent l'assistant de preuve Coq pour spécifier un compilateur d'un noyau de Lustre vers le langage Clight de CompCert pour ensuite générer du code assembleur. La preuve de correction de l'ensemble relie la sémantique de flots de Lustre avec la sémantique impérative du code assembleur.

Chaque flot dans un programme Lustre est associé avec une « horloge » statique qui représente ses instants d'activation. La compilation transforme les horloges en des instructions conditionnelles qui déterminent quand les valeurs associées sont calculées. Les travaux précédents faisaient l'hypothèse simplificatrice que toutes les entrées et sorties d'un bloc partagent la même horloge. Cet article décrit une façon de supprimer cette restriction. Elle exige d'abord d'enrichir les règles de typage des horloges et le modèle sémantique. Ensuite, pour satisfaire le modèle sémantique de Clight, on ajoute une étape de compilation pour assurer que chaque variable passée directement à un appel de fonction a été initialisée.

1 Introduction

Les langages basés sur les schémas-blocs sont couramment utilisés dans la programmation des logiciels contrôle-commande critiques. Le langage Scade 6 [9] en est un exemple emblématique ainsi que son prédécesseur académique le langage synchrone flot de données Lustre [11]. Les compilateurs des langages schéma-blocs traduisent des modèles abstraits en du code impératif de bas niveau, typiquement en C ou en Ada. Le compilateur Scade est doté de plusieurs qualifications aux normes industrielles. L'avantage majeur de ces qualifications est que dans un processus de développement certifié où Scade est utilisé, les exigences ne doivent être tracées qu'aux modèles de conception de haut niveau plutôt qu'au code de bas niveau. L'inconvénient est que la qualification d'un compilateur est pénible et coûteuse.

Les assistants de preuve permettent d'encoder les spécifications formelles et les algorithmes et de vérifier leur cohérence et leurs propriétés sur ordinateur. Des travaux récents montrent comment appliquer de tels outils pour spécifier et raisonner sur les compilateurs de langages impératifs de bas niveau comme C, dans le compilateur CompCert [14], et de langages avec gestion de mémoire automatique comme Standard ML, dans le compilateur CakeML [13]. Dans le projet Vélus [6, 5], nous développons un prototype d'un compilateur pour le noyau d'un langage synchrone flot de données en utilisant l'assistant de preuve Coq [15]. Le but ultime est de fournir un schéma directeur pour appliquer un assistant de preuve à la spécification et à la vérification de langages schéma-bloc et d'avoir un aperçu de leur utilité pour faciliter la qualification d'outils industriels.

Les deux éléments principaux d'un langage schéma-bloc sont les « blocs », pour représenter les unités fonctionnelles comme des compteurs ou des filtres, et les « signaux », pour lier les entrées et les sorties des blocs. En Lustre, un signal est modélisé par un flot de valeurs et un bloc, appelé un *nœud*, est modélisé par une fonction de flots en flots. Les flots d'un programme Lustre

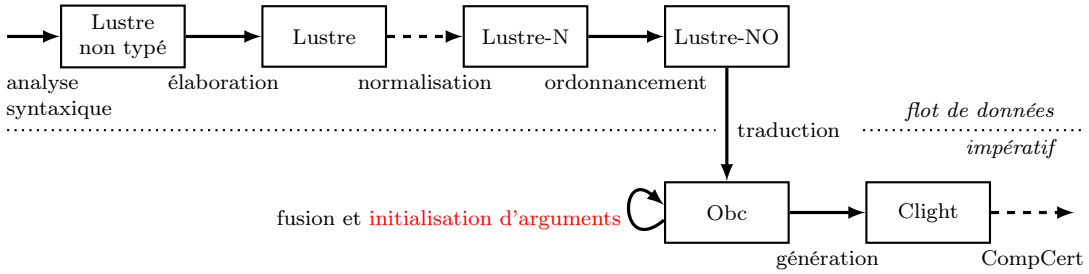

FIGURE 1 – Architecture d'un compilateur du code « modulaire et dirigé par les horloges ».

sont synchronisés les uns aux autres par rapport à une *horloge de base* qui est finalement réalisée par l'exécution répétée du code généré. Lustre fournit des opérateurs pour échantillonner ou sur-échantillonner un flot, ce qui permet l'activation conditionnelle de sous-parties d'un programme et, en particulier, la compilation efficace des constructions de haut niveau comme les machines à état hiérarchiques. L'utilisation des opérateurs d'échantillonnage est limitée par un système de typage [10] qui assure que les programmes acceptés peuvent être exécutés en mémoire bornée [7].

Les opérateurs d'échantillonnage et le typage par horloges statiques sont implémentés dans la première version du compilateur Vélus [6, 5] avec la restriction que les entrées et sorties d'un même nœud doivent toutes avoir la même horloge statique. Dans cet article, nous expliquons nos travaux récents pour enlever cette restriction et traiter les arguments cadencés.

L'architecture globale du compilateur Vélus est illustrée à la figure 1. Elle suit l'approche dite « modulaire et dirigée par les horloges » [1]. Le résultat de l'analyse syntaxique d'un fichier source est un arbre de syntaxe abstrait auquel l'*élaboration* ajoute des annotations validées pour les types et les horloges statiques. Une étape de *normalisation* est appliquée pour simplifier la forme des expressions et équations pour produire des programmes Lustre-N. La normalisation n'est pas encore implémenté, nous rejetons donc les programmes qui ne sont pas déjà en forme normale. Dans cet article, nous nous concentrons sur le langage Lustre-N. Les modifications qu'il a fallu apportées au langage Lustre-N et à ses règles d'horloge sont décrites dans la section 2. Les équations dans chaque nœud sont ordonnées par l'étape d'*ordonnancement* pour produire des programmes Lustre-NO, c'est-à-dire, Lustre-N et un prédicat sur l'ordre d'équations. Le prédicat est indispensable à l'étape suivante de *traduction* qui transforme les programmes Lustre-NO en un langage intermédiaire qui s'appelle Obc. Une optimisation de *fusion* factorise les instructions conditionnelles dans le code intermédiaire avant que l'étape de *génération* traduise un programme Obc en Clight pour ensuite être compilé par CompCert. La section 3 présent une vue d'ensemble de la génération du code impératif et décrit les obligations de preuves supplémentaires engendrées par notre introduction des arguments cadencés et l'utilisation de Clight. En particulier, il est nécessaire d'adapter Obc pour permettre l'utilisation de variables non initialisées dans les appels de méthode, comme le décrit la section 4, et puis d'assurer l'initialisation de ces variables en ajoutant une étape de compilation, indiquée en rouge dans la figure 1 et décrit dans la section 5, avant la génération de Clight.

2 Lustre normalisé et ses horloges

Un programme Lustre-N est une liste de nœuds. Chaque nœud définit une fonction entre une liste de variables d'entrée et une liste de variables de sortie. Un exemple est présenté dans la figure 2. Les définitions dans l'exemple ont pour seul but de présenter les caractéristiques du

```
node n1(c : bool; x : bool when c)
returns (v : int; o : int when c); (* ... *)

node n2(c1       : bool;
        c2, w, y : bool when c1;
        x        : bool when c2;
        z        : bool when not c1)
returns (o : int); (* ... *)

node f(m : bool; h : bool when m; x : bool)
returns (o1 : int; o2 : int when m);
var e0, e3 : bool when h;
    w      : int  when m;
    y      : int  when h;
    r1, e1 : bool when m;
    r2, e2 : bool when not m;
let
  e0 = (not (x when m)) when h;
  e1 = r1 and (x when m);
  e2 = r2 and (x when not m);
  e3 = e0 or ((x when m) when h);
  r1 = false fby not r1;
  r2 = true fby not r2;
  (w, y) = n1(h, e3);
  o1 = n2(m, h, (not x) when m, e1, e0, e2);
  o2 = merge h y (0 when not h);
tel
```

FIGURE 2 – Exemple : source Lustre

```
class f {
  instance i1 : n1;
  instance i2 : n2;
  memory r1, r2 : bool;

  method reset() { ··· }

  method step(m, h, x : bool)
  returns (o1, o2 : int32)
  var e0, e1, e2, e3 : bool; w, y : int32
  {
    e0 := 0;
    if (m) {
      e2 := 0;
      e1 := state(r1) & x;
      state(r1) := ! state(r1);
      if (h) {
        e0 := ! x;
        e3 := e0 | x
      } else {
        e3 := 0;
        skip
      };
      y := n1(i1).step(⟨h⟩, ⟨e3⟩);
      if (h) {
        o2 := y
      } else {
        o2 := 0
      }
    } else {
      o2 := 0;
      e1 := 0;
      e2 := state(r2) & x;
      state(r2) := ! state(r2)
    };
    o1 := n2(i2).step(⟨m⟩, ⟨h⟩, ! x,
                      ⟨e1⟩, ⟨e0⟩, ⟨e2⟩)
  }
}
```

FIGURE 3 – Exemple : Obc généré

langage et de sa compilation. Le programme est composé de trois nœuds : `n1`, `n2` et `f`. Nous nous concentrons sur le dernier nœud `f` dans lequel on instancie les autres nœuds, `n1` et `n2`, dont les définitions ne sont pas données.

Le nœud `f` a trois variables d'entrée de type `bool` : `m`, `h` et `x`. L'annotation « `bool when m` » à droite de `h` indique que cette variable représente un flot de valeurs booléennes qui ne sont *présentes* que quand `m` est `true`. Considérons, par exemple, les trois flots d'entrée suivants.

m	false	false	true	false	···	false	true	false	true	true	···
h			false		···		true		false	false	···
x	false	true	false	true	···	true	true	false	false	false	···

Les valeurs des flots `m` et `x` sont toujours présentes, mais celle du flot `h` est *absente* dans les colonnes où `m` est `false`. L'alignement des valeurs dans les colonnes se justifie par le modèle synchrone du temps. L'absence d'une valeur est indiquée par un espace vide.

Le corps d'un nœud est une liste d'équations. Il y en a une pour chaque variable de sortie, celles déclarées après `returns`, et chaque variable locale, celles déclarées après `var`. La signifi-

cation d'un nœud est insensible à l'ordre de ses équations. Les expressions qui définissent les équations sont construites à partir de constantes, variables, l'opérateur `when`, les opérateurs point par point unaires (p. ex., `not`) et binaires (p. ex., `and` et `or`) et l'opérateur `merge`[1]. L'opérateur `when` échantillonne un flot. Dans les flots d'entrée donnés en exemple ci-dessus, `h` égale « `x when m` ». Il est également possible de spécifier le flot complémentaire, par exemple, « `x when not m` », et de fusionner deux flots complémentaires : « `merge m h (x when not m)` » spécifie un flot identique à celui d'`x`.

La construction des expressions et équations est régie par un système de typage basé sur les horloges statiques. Une *horloge* est définie par la grammaire suivante que nous encodons comme le type inductif *clock* dans notre formalisation Coq.

$$ck ::= \mathsf{base} \mid ck \ \mathsf{on} \ (x = b)$$
$$b ::= \mathsf{true} \mid \mathsf{false}$$

Le constructeur `on` est associatif de gauche à droite. Dans l'exemple, l'horloge des variables `m` et `x` est base — elles sont toujours présentes dans `f` — et celle de `h` est base on $(m = \mathsf{true})$. Dans la définition de `e0`, l'horloge de « `x when m` » est base on $(m = \mathsf{true})$, tout comme celle de « `not (x when m)` » ; l'horloge de « `(not (x when m)) when h` » et donc aussi celle d'`e0` est base on $(m = \mathsf{true})$ on $(h = \mathsf{true})$. Les horloges servent à rejeter les expressions incorrectes. Par exemple, « `(x when m) + (x when not m)` » ne peut pas être calculée sans une mémoire tampon potentiellement sans borne puisque l'opérateur d'addition exige une valeur de chacun de ses flots d'entrée pour en produire une sur son flot de sortie. En Lustre, les arguments d'un opérateur binaire doivent avoir la même horloge statique. Les horloges jouent aussi un rôle important dans la génération du code impératif tel que décrit dans la section suivante.

Il y a trois formes d'équation en Lustre-N : (*a*) les équations simples, dans l'exemple, celles qui définissent `e0`, `e1`, `e2`, `e3` et `o2`, (*b*) les équations définies par un seul `fby`, comme celles de `r1` et `r2`, et (*c*) les équations définies par l'instanciation d'un autre nœud, comme celles de `w`/`y` et `o1`. Une équation `fby` spécifie un délai initialisé. Pour les entrées dans l'exemple, `r1` et `r2` prennent les valeurs suivantes.

`r1`			false		⋯		true		false	true	⋯
`r2`	true	false		true	⋯	false		true			⋯

La première valeur est définie par la constante à la gauche du `fby` et la suite égale au flot à sa droite (avec un délai unitaire).

L'exemple de la figure 2 n'est pas accepté par les versions précédentes de Vélus [6, 5], car les interfaces des nœuds contiennent une ou plusieurs variables sur des horloges différentes. En `n1`, par exemple, `c` et `v` ont l'horloge base alors que `x` et `o` ont l'horloge base on $(c = \mathsf{true})$.

Plusieurs modifications aux parties flot de données du compilateur — les langages et algorithmes au-dessus de la ligne en pointillé de la figure 1 — sont nécessaires pour permettre les interfaces plus riches. Les modifications de l'analyseur syntaxique et de l'arbre de syntaxe abstrait étaient mineures. L'élaboration a été adaptée pour déduire les horloges des constantes — dans l'exemple, à la ligne 26, le « `when not h` » autour de la constante `0` est ajouté automatiquement — et pour contrôler les horloges dans les équations où un nœud est instancié. L'étape d'ordonnancement et le système de typage ne changent pas. La plupart des modifications s'appliquent au système d'horloges et aux modèles sémantiques.

Dans notre compilateur, un nœud est représenté par un enregistrement qui rassemble son nom, les listes des déclarations de variables d'entrée (nins), des variables locales (nvars) et des

1. L'utilisation de `merge` est restreinte dans un programme normalisé, mais les règles précises ne sont pas importantes ici.

$$\textsc{SameVar-None}\quad \mathsf{SameVar\ None}\ e \qquad \textsc{SameVar-Some}\quad \mathsf{SameVar\ (Some}\ z)\ z \qquad \textsc{inst-base}\quad inst_{sub}\ nck\ \mathsf{base} = \mathsf{Some}\ nck$$

$$\textsc{inst-on}\ \frac{inst_{sub}\ nck\ ck = \mathsf{Some}\ ck' \qquad sub\ x = \mathsf{Some}\ x'}{inst_{sub}\ nck\ (ck\ \mathsf{on}\ (x = b)) = \mathsf{Some}\ \big(ck'\ \mathsf{on}\ (x' = b)\big)}$$

$$\textsc{Clk-EqApp}\ \frac{\mathsf{find\text{-}node}\ f\ G = \mathsf{Some}\ n \qquad (\forall x,\ \neg \mathsf{In}\ x\ (\mathsf{map\ fst}\ n.(\mathsf{nouts})) \implies osub\ x = \mathsf{None}) \qquad \mathsf{Forall2}\left(\lambda(x,(ty,ck))\ e,\ \mathsf{SameVar}\ (isub\ x)\ e \wedge \begin{array}{c}\exists ck',\ inst_{isub}\ nck\ ck = \mathsf{Some}\ ck' \\ \wedge \qquad \mathsf{wc\_exp}\ e\ ck'\end{array}\right) n.(\mathsf{nins})\ es \qquad \mathsf{Forall2}\left(\begin{array}{ll}\lambda(x,(ty,ck))\ y, & (isub \lhd\!\!+ osub)\ x = \mathsf{Some}\ y \\ & \wedge \exists ck', \quad \mathsf{In}\ (y,ck')\ vars \\ & \wedge\ inst_{isub \lhd\!\!+ osub}\ nck\ ck = \mathsf{Some}\ ck'\end{array}\right) n.(\mathsf{nouts})\ xs}{\mathsf{wc\_equation}(xs =^{nck} f(es))}$$

FIGURE 4 – La règle d'horloges statiques d'instanciation de nœud dans Lustre-N.

variables de sortie (nouts), la liste d'équations et quelques prédicats de bonne formation. Les déclarations de variables sont du type list (ident × (type × clock)).

Nous introduisons le prédicat wc_env pour affirmer la bonne formation des horloges dans une déclaration. Ce prédicat est utilisé trois fois : wc_env nins, wc_env (nins ++ nouts) et wc_env (nins ++ nvars ++ nouts). De plus, elle exige que chaque équation satisfasse un prédicat wc_equationqui affirme la bonne formation des horloges dans les équations.

Les cas de wc_equation pour les équations simples et `fby`s suivent simplement la structure des termes. Les définitions associées aux instanciations des nœuds sont décrites dans la figure 4. Étant donné un programme G et une liste de déclarations de variables $vars$, la règle Clk-EqApp définit le prédicat pour une équation « $xs =^{nck} f(es)$ » où xs est une liste de noms de variable, nck est l'horloge de base de l'instanciation, f est le nom du nœud et es est une liste d'expressions. La règle exige l'existence d'un nœud n associé à f dans G et deux fonctions partielles $isub$ et $osub$ qui associent respectivement les variables d'entrée et les variables de sortie dans la déclaration du nœud aux variables dans le contexte de son instanciation. Il faut que $osub$ n'associe que les variables de sorties déclarées pour le nœud.

La première proposition Forall2 [2] dans Clk-EqApp fait le lien entre chaque déclaration d'entrée dans $n.(\mathsf{nins})$ et l'expression correspondante dans es. Pour une variable d'entrée x d'horloge ck et l'expression correspondante e, il y a trois conditions : (a) si e est une variable z alors $isub\ x = \mathsf{Some}\ z$, c'est-à-dire, les variables qui peuvent apparaître dans les horloges déclarées dans l'interface d'un nœud sont cohérentes avec celles qui peuvent apparaître dans les horloges de es et de xs, (b) l'instanciation de la horloge ck en substituant ses variables selon $isub$ et son horloge de base avec nck est une horloge ck', pour laquelle (c) l'expression e est bien cadencée.

La deuxième proposition Forall2 dans Clk-EqApp relie chacune des variables de sortie dans $n.(\mathsf{nouts})$ à la variable correspondante dans xs. Pour une variable de sortie x d'horloge ck et la variable correspondante y, il y a trois conditions : (a) x est remplacé, soit par $isub$ soit par

2. Forall2 P xs ys est satisfait seulement si $xs = ys = [\,]$ ou si P (hd xs) (hd ys) et Forall2 P (tl xs) (tl ys).

osub, par y[3], (*b*) y est déclarée dans *vars* avec horloge ck', (*c*) l'instanciation de ck en utilisant *isub*, *osub* et *nck* est ck'.

À titre d'exemple de la règle Clk-EqApp, considérons l'interface de `n1` dans la figure 2.

$$n_1.(\mathsf{nins}) = [(c, (\mathsf{bool}, \mathsf{base})), (x, (\mathsf{bool}, \mathsf{base\ on}\ (c = \mathsf{true})))]$$
$$n_1.(\mathsf{nouts}) = [(v, (\mathsf{int}, \mathsf{base})), (o, (\mathsf{int}, \mathsf{base\ on}\ (c = \mathsf{true})))]$$

Ce nœud est instancié dans `f` sur l'horloge $\mathsf{base\ on}\ (m = \mathsf{true})$ avec deux arguments $es = [h, e_3]$ pour définir $xs = [w, y]$. Pour les substitutions $isub = [c \mapsto h, x \mapsto e_3]$ et $osub = [v \mapsto w, o \mapsto y]$, les horloges des entrées sont instanciées à $\mathsf{base\ on}\ (m = \mathsf{true})$ et $\mathsf{base\ on}\ (m = \mathsf{true})\ \mathsf{on}\ (h = \mathsf{true})$, en concordance avec celles déclarées pour h et e_3, et les horloges des sorties sont instanciées à $\mathsf{base\ on}\ (m = \mathsf{true})$ et $\mathsf{base\ on}\ (m = \mathsf{true})\ \mathsf{on}\ (h = \mathsf{true})$, en concordance avec celles déclarées pour w et y.

La généralisation des horloges des instanciations entraîne deux modifications peu invasives aux modèles sémantiques. La première a été de remplacer une contrainte forçant les valeurs des flots d'entrée et de sortie d'être tous présentes ou absentes ensemble aux mêmes instants par une contrainte forçant l'horloge de base du nœud à être vrai exactement quand au moins une des entrées est présente. La seconde a été de contraindre les présences et absences de variables à correspondre avec leurs horloges déclarées. L'utilisation des annotations d'horloge non seulement pour contrôler les comportements dynamiques, mais aussi pour les contraindre peut sembler peu satisfaisante. En Lustre-N cependant, on a voulu donner une signification déterministe aux composants « isolés » comme « `x = false fby (not x)` » qui sont autrement libres d'avoir une valeur présente ou absente à n'importe instant. Ce problème ne se pose pas dans Lustre non normalisé où la valeur à gauche d'un `fby` est une expression et non une constante. Une constante s'exécute par définition sur l'horloge de base alors que les `when`s peuvent être ajoutés à une expression pour adapter son rythme. La solution se trouvera dans le nouveau modèle sémantique de Lustre. Nous réservons à l'avenir également les preuves pour lier les règles d'horloges aux propriétés du modèle sémantique

2.1 Travaux connexes

Les travaux précédents [10] sur un langage flot de données d'ordre supérieur démontrent que les horloges statiques peuvent être formulées comme des types dépendants et traités par les techniques utilisés dans les systèmes de typage à la ML. Ces horloges sont stratifiées en schèmes d'horloges, horloges, horloges de flot et variables porteuses [10, §4]. Tout comme pour les types polymorphes de ML, un schème d'horloge est créé à la déclaration d'un nœud par une quantification sur les variables libres dans les horloges de ses entrées et sorties. Un schème est instancié par la substitution de ces variables libres chaque fois qu'un nœud est utilisé dans une équation. Dans notre cadre normalisé, les nœuds sont instanciés dans des équations distinctes, alors les noms des variables dans les horloges sont connus directement et les variables porteuses ne sont pas nécessaires.

Des travaux précédents [3, 4] décrivent le plongement superficiel d'un langage synchrone flot de données dans Coq où les horloges sont représentées comme des suites coinductives de booléens. Nous nous concentrons sur un plongement profond puisque notre but est d'extraire un compilateur qui contrôle et transforme les termes explicitement.

3. L'union de deux substitutions $S \lhd\!\!+ T$ avec priorité à S, c'est-à-dire, $(S \lhd\!\!+ T)(x)$, égal $S(x)$ si ce n'est pas None, sinon $T(x)$.

3 Compilation vers code impératif

Après ordonnancement, un programme Lustre-N est compilé en deux étapes. La première produit et optimise un code dans le langage impératif Obc. La deuxième génère un code Clight.

Le code Obc optimisé produit pour l'exemple est illustré à la figure 3 à l'exclusion des éléments en rouge. À ce stade, nous remarquons seulement que chaque équation devient une affectation et que des instructions conditionnelles sont introduites pour traduire les horloges du programme source. Par exemple, puisque l'horloge d'`e0` est base on $(m = \mathsf{true})$ on $(h = \mathsf{true})$, l'affectation correspondante est emboîtée dans les instructions conditionnelles suivantes.

```
if (m) { ··· if (h) { ··· } ··· }
```

En conséquence de cet encodage, dans une instanciation de nœud avec des arguments sur différentes horloges, les variables passées en argument n'ont pas nécessairement toujours été initialisées. Par exemple, lorsque `m = false`, les valeurs de `h`, `e1`, `e0` et `e2` sont indéterminées dans l'appel aux lignes 35 et 36.

Nous avons adapté Obc pour permettre l'utilisation de variables non initialisées dans les appels de méthode. Cependant la norme C99 précise que le passage de valeurs indéterminées dans un appel de fonction n'est pas bien défini : si un lvalue ne désigne pas un objet lors de son évaluation, le comportement n'est pas défini [4] [12, §6.3.2.1], pendant la préparation d'un appel de fonction, les arguments sont évalués, et à chaque paramètre est affecté la valeur de l'argument correspondant [5] [12, §6.5.2.2]. Cette interprétation est formalisée par les modèles sémantiques de Clight [2, figure 10]. Lors de l'évaluation d'un appel de fonction interne, les arguments sont d'abord évalués un à un [6]. La conversion de type appliquée à chacun doit rendre une valeur, mais une conversion à tout autre type que `void` de la valeur donnée aux variables locales non initialisées (`Vundef`) rend `None` [7].

Il y a au moins deux façons de contourner cette contrainte. On pourrait traduire Obc dans l'un des autres langages intermédiaires de CompCert, en l'occurrence Cminor où les arguments indéfinis sont permis. Cela nécessiterait toutefois de reproduire et de revérifier les fonctionnalités qui ne sont pas fournis par Cminor, comme l'empilement d'arguments. Une autre possibilité serait de modifier les modèles sémantiques de Clight et de corriger les preuves associées. Cette approche induirait du travail supplémentaire et le code généré ne serait plus conforme à la norme C. Nous avons donc décidé d'accepter les contraintes imposées par C99 et Clight et de chercher une autre solution.

L'approche adoptée dans Scade est de faire une extension inline des fonctions où une ou plusieurs entrées sont sous-échantillonnées [8]. Cela a trois avantages : les valeurs des arguments dans le code généré sont toujours définies, l'optimisation de fusion s'applique à travers les nœuds et les justifications de correction sont faites dans le langage flot de données. Nous n'avons pas suivi cette approche parce que l'extension inline n'a pas encore été implémentée dans Vélus et nous étions curieux d'évaluer une différente solution qui maintien la modularité du programme source dans le code généré.

Au moins deux autres solutions sont possibles lors de la traduction en Obc et la génération de Clight. La première est d'implémenter les appels de fonction avec un argument pointeur vers un enregistrement contenant toutes ou une partie des valeurs d'entrée. Le pointeur serait initialisé et on sait par construction que le code généré ne lit jamais les valeurs indéterminées.

4. « . . . if an lvalue does not designate an object when it is evaluated, the behavior is undefined. »
5. « In preparing for the call to a function, the arguments are evaluated, and each parameter is assigned the value of the corresponding argument. »
6. Voir `cfrontend/Clight.v` : `eval_exprlist`.
7. Voir `cfrontend/Cop.v` : `sem_cast`.

La seconde solution est de traiter certaines variables d'entrées comme des variables d'état qui sont mises dans la mémoire statique et sont donc toujours bien définies en C. L'une ou l'autre solution compliquerait les fonctions de compilation et leurs preuves de correction.

En fin de compte, la solution la plus simple est d'initialiser toutes les variables au début de chaque fonction. Dans bien des cas, CompCert sait éliminer les écritures qui ne sont pas nécessaires et le coût des autres initialisations est probablement insignifiant. Nous avons essentiellement adopté cette solution, mais avec deux améliorations. D'abord, nous exploitons la preuve de correction et quelques petites modifications à la sémantique d'Obc pour éviter d'introduire d'écritures inutiles, tel que décrit dans la section 4. Ensuite, nous ajoutons une étape de compilation pour réécrire les programmes Obc en ajoutant les initialisations nécessaires, telle que décrit dans la section 5. La nouvelle étape de compilation utilise une heuristique qui tente de minimiser à la fois les répétitions inutiles d'écritures et le nombre d'initialisations ajoutées sans trop compliquer les obligations de preuve concomitantes.

4 La génération du code intermédiaire

La traduction de Lustre-N en Obc est détaillée ailleurs [6, 5]. Ici nous nous concentrons sur la généralisation du passage des arguments dans les appels de méthodes.

Les expressions e et les commandes s du langage Obc sont définies par la grammaire suivante (avec l'omission des annotations de typage).

$$e ::= x \mid \texttt{st}(x) \mid c \mid \ \diamond e \mid e \oplus e \mid \langle e \rangle$$
$$s ::= x := e \mid \texttt{st}(x) := e \mid \texttt{if}\ e\ \{s\}\ \texttt{else}\ \{s\} \mid s\ ;\ s \mid xs := cls(i).f(es) \mid \texttt{skip}$$

Une expression est une variable (x), une variable d'état ($\texttt{st}(x)$), une constante (c), un opérateur unaire ($\diamond$), un opérateur binaire ($\oplus$) ou une *assertion de validité* ($\langle\cdot\rangle$). Une commande est une affectation à une variable, une affectation à une variable d'état, une instruction conditionnelle, une composition séquentielle, un appel de méthode ou une instruction nulle. Nous définissons une sémantique à grands pas pour les expressions et les commandes. Nous écrivons

$$menv, env \vdash e \Downarrow vo$$

pour affirmer que l'évaluation d'une expression e dans l'environnement de variables d'état $menv$ et l'environnement de variables env produit la valeur optionnelle vo. Nous écrivons

$$p, menv, env \vdash s \Downarrow (menv', env')$$

pour affirmer que l'évaluation d'une commande s dans le programme p et les environnements $menv$ et env produit les environnements $menv'$ et env'.

Un environnement de variables (env) est une application partielle entre variables et valeurs. Un environnement de variables d'état ($menv$) a deux composants : une application partielle entre variables d'état et valeurs ($menv_{\mathsf{v}}$) et une application entre noms de nœud et environnements de variables d'état ($menv_{\mathsf{o}}$)[8].

Les règles de sémantique d'Obc sont présentées dans les figures 5 et 6. La seule chose de remarquable est le traitement de valeurs partielles, autrement dit, la possibilité qu'une variable ne soit pas définie dans un environnement. Une expression x s'évalue à None si x n'est pas défini dans l'environnement de variables, c'est-à-dire, si une valeur n'a pas été affectée à lui. Toutes les autres expressions doivent s'évaluer à une valeur « Some ». L'évaluation d'une commande d'affectation n'est pas définie quand son expression s'évalue à None. Cependant, les appels de méthode ne sont pas ainsi contraints, il est donc possible de passer et de renvoyer une variable qui n'a pas forcément été écrite. La règle pour les appels de méthode utilise une fonction adds

8. Ces applications partielles sont réalisées dans Coq avec l'aide de la bibliothèque FMapPositive.

$$\textsc{Obc-Exp-Var}\ \frac{env\ x = vo}{menv, env \vdash x \Downarrow vo} \qquad \textsc{Obc-Exp-State}\ \frac{menv_{\mathsf{v}}\ x = \mathsf{Some}\ v}{menv, env \vdash \mathtt{st}(x) \Downarrow \mathsf{Some}\ v} \qquad \textsc{Obc-Exp-Const}\ menv, env \vdash c \Downarrow \mathsf{Some}\ [\![c]\!]$$

$$\textsc{Obc-Exp-Unop}\ \frac{menv, env \vdash e \Downarrow \mathsf{Some}\ v \qquad [\![\diamond\ v]\!] = \mathsf{Some}\ v'}{menv, env \vdash \diamond\ e \Downarrow \mathsf{Some}\ v'} \qquad \textsc{Obc-Exp-Assert}\ \frac{menv, env \vdash e \Downarrow \mathsf{Some}\ v}{menv, env \vdash \langle e \rangle \Downarrow \mathsf{Some}\ v}$$

$$\textsc{Obc-Exp-Binop}\ \frac{menv, env \vdash e_1 \Downarrow \mathsf{Some}\ v_1 \qquad menv, env \vdash e_2 \Downarrow \mathsf{Some}\ v_2 \qquad [\![v_1 \oplus v_2]\!] = \mathsf{Some}\ v'}{menv, env \vdash e_1 \oplus e_2 \Downarrow \mathsf{Some}\ v'}$$

FIGURE 5 – Les règles sémantiques des expressions Obc.

$$\textsc{Obc-Stmt-Assign}\ \frac{menv, env \vdash e \Downarrow \mathsf{Some}\ v}{p, menv, env \vdash x := e \Downarrow (menv, env[x \mapsto v])}$$

$$\textsc{Obc-Stmt-StAssign}\ \frac{menv, env \vdash e \Downarrow \mathsf{Some}\ v}{p, menv, env \vdash \mathtt{st}(x) := e \Downarrow (menv_{\mathsf{v}}[x \mapsto v], env)}$$

$$\textsc{Obc-Stmt-Comp}\ \frac{p, menv, env \vdash s_1 \Downarrow (menv_1, env_1) \qquad p, menv_1, env_1 \vdash s_2 \Downarrow (menv_2, env_2)}{p, menv, env \vdash s_1\ ;\ s_2 \Downarrow (menv_2, env_2)}$$

$$\textsc{Obc-Stmt-Ite-True}\ \frac{menv, env \vdash e \Downarrow \mathsf{Some\ true} \qquad p, menv, env \vdash s_1 \Downarrow (menv', env')}{p, menv, env \vdash \mathtt{if}\ e\ \{s_1\}\ \mathtt{else}\ \{s_2\} \Downarrow (menv', env')}$$

$$\textsc{Obc-Stmt-Ite-False}\ \frac{menv, env \vdash e \Downarrow \mathsf{Some\ false} \qquad p, menv, env \vdash s_2 \Downarrow (menv', env')}{p, menv, env \vdash \mathtt{if}\ e\ \{s_1\}\ \mathtt{else}\ \{s_2\} \Downarrow (menv', env')}$$

$$\textsc{Obc-Stmt-Skip}\ p, menv, env \vdash \mathtt{skip} \Downarrow (menv, env)$$

$$\textsc{Obc-Stmt-Call}\ \frac{\begin{array}{c}\mathsf{Forall2}\ (\lambda e\ vo,\ menv, env \vdash e \Downarrow vo)\ es\ vos \\ \mathsf{find\text{-}class}\ cls\ p = \mathsf{Some}\ (c, p') \qquad \mathsf{find\text{-}method}\ f\ c.(\mathsf{methods}) = \mathsf{Some}\ m \qquad |vos| = |m.(\mathsf{nins})| \\ p', menv_{\mathsf{o}}\ i, \mathsf{adds}(\mathsf{map\ fst}\ m.(\mathsf{nins}), vos, \mathsf{empty}) \vdash m.(\mathsf{mbody}) \Downarrow (omenv', oenv') \\ \mathsf{Forall2}\ (\lambda x\ vo,\ oenv'\ x = vo)\ (\mathsf{map\ fst}\ m.(\mathsf{nouts}))\ ros\end{array}}{p, menv, env \vdash ys := cls(i).f(es) \Downarrow (menv_{\mathsf{o}}[i \mapsto omenv'], \mathsf{updates}(env, ys, ros))}$$

FIGURE 6 – Les règles sémantiques des commandes Obc.

$$\textsc{NoOps-base}\ \frac{}{\mathsf{NoOps}\ \mathsf{base}\ e} \qquad \textsc{NoOps-const}\ \frac{}{\mathsf{NoOps}\ ck\ c} \qquad \textsc{NoOps-var}\ \frac{}{\mathsf{NoOps}\ ck\ x} \qquad \textsc{NoOps-when}\ \frac{\mathsf{NoOps}\ ck\ e}{\mathsf{NoOps}\ (ck\ \mathsf{on}\ (x = b))\ (e\ \texttt{when}\ (\texttt{not})\ x)}$$

FIGURE 7 – La condition de normalisation en Lustre-N entre une horloge d'une entrée d'un nœud et une expression passée comme argument.

pour créer un environnement initial pour les variables d'entrée et une fonction `updates` pour soit copier les valeurs qui sont définies, soit supprimer les valeurs non définies (qui sont donc égales à None).

Le traitement des valeurs partielles en Obc demande une attention particulière dans la preuve de correction de la traduction de Lustre-N en Obc. Cette preuve établit un rapport entre présence et absence dans le programme source et l'exécution conditionnelle des commandes dans le code généré. L'invariant sous-jacent est souvent suffisant pour garantir qu'une variable est bien définie dans les commandes où elle est lue. C'est le cas pour les variables dans les expressions générées d'équations simples et de `fby`s, et aussi pour celles dans les conditions générées à partir d'horloges statiques (p. ex., `h` à la ligne 16 de la figure 3). Par contre, un soin supplémentaire doit être apporté aux expressions dans les appels de méthode.

Considérons d'abord le problème posé dans un appel de méthode par une expression qui contient un opérateur unaire ou binaire. Par exemple, dans la figure 2 à la ligne 24 : le programme qui résulte du remplacement de `e3` par « `e3 + (r1 when h)` » est légitime, mais sa traduction en Obc, « `e3 + r1` », n'est pas définie lorsque `h = false` puisque le résultat d'une addition avec None n'est pas défini. Alors qu'une addition est toujours définie en Clight, ce n'est pas le cas pour tous les opérateurs : l'important est d'assurer qu'une propriété vérifiée sur un programme source, comme l'absence de division entière par zéro, l'est également vrai pour le code généré. Pour garantir l'existence d'un modèle sémantique pour le code généré, nous exigeons que le programme source satisfasse le prédicat NoOps dont la définition est présentée dans la figure 7. Ce prédicat doit être satisfait par tous les arguments de toutes les instanciations de nœud. Il définit des combinaisons admissibles entre une horloge statique de l'interface d'un nœud (celles dans nins) et une expression passée comme argument. Pour l'horloge base, toutes les expressions sont permises puisque les règles d'horloges et l'invariant de correction garantissent que les valeurs des variables dans l'expression sont toujours présentes lors de l'activation du nœud. Les constantes et les variables sont toujours permises. Autrement, un ou plusieurs niveaux d'échantillonnage peuvent être enlevés pourvu que l'expression échantillonnée soit présente quand l'horloge de base du nœud est vraie. On exploite cette condition syntaxique dans la preuve de correction de la traduction pour démontrer que le code généré a une sémantique. La condition doit être garantie par l'étape de normalisation ou contrôlée lors de l'élaboration.

Considérons maintenant les autres expressions d'arguments possibles. La sémantique d'une constante est toujours définie et ce cas ne pose donc aucun problème. Une variable x en Lustre-N est traduite en Obc soit comme un variable x soit comme une variable d'état `st`(x). Une variable d'état est toujours définie quelle que soit son horloge dans le programme source. Pour les autres variables, il y a assez d'information dans la preuve de correction pour garantir que celles sur l'horloge de base d'une instanciation de nœud sont toujours définies lors de l'exécution de l'appel de méthode correspondant dans le code généré. Pour marquer ce fait, nous enveloppons ces variables dans une assertion de validité : $\langle x \rangle$. L'évaluation d'une assertion ne rend une valeur que quand l'expression qu'elle enveloppe ne s'évalue pas à None. Certaines assertions de validité sont ajoutées par l'étape de traduction en Obc. La preuve de correction de la traduction garantit

qu'elles sont bien définies. On profite ensuite des assertions dans la preuve de correction de la génération de Clight.

Dans le code généré illustré dans la figure 3, les assertions (en noire) autour de `h` à la ligne 23 et de `m` à la ligne 35 sont ajoutées lors de la traduction vers Obc et donc justifiées par la preuve de correction correspondante. Nous pouvons savoir que ces variables sont toujours définies en amont des appels de méthode : `m`, car elle est toujours présente, et `h`, car elle est présente quand `m` est vraie et l'appel de méthode à la ligne 23 se trouve dans l'instruction conditionnelle `if (m) { ··· }`. Dans ce cas, les deux variables sont des entrées, mais le même raisonnement se tient pour les variables locales. Il n'est pas nécessaire d'ajouter une assertion de validité autour des expressions comme le `! x` à la ligne 35 : elles doivent satisfaire NoOps et leur sémantique est donc forcément définie.

Les variables dans un appel de méthode qui ne sont pas sur l'horloge de base du nœud instancié doivent être explicitement initialisées pour justifier l'addition des assertions de validité (celles illustrées en rouge) qui sont essentielles pour la preuve de correction de la génération de Clight. Cette transformation est réalisée par l'étape de compilation appelée, dans la figure 1, « initialisation d'arguments » et décrite dans la section suivante.

5 L'initialisation d'arguments

Le code Obc généré contient des assertions de validité pour toutes les variables d'argument qui sont garanties d'être définies lors d'un appel de méthode. L'idée maintenant est d'ajouter des assertions de validité à toutes les autres variables d'arguments et de les justifier en ajoutant des affectations supplémentaires auparavant. Les preuves de correction des nouveaux programmes produits ainsi et les codes Clight générés par la suite s'appuient sur trois invariants introduits dans cette section. En outre, bien que les variables qui sont définies dans l'ancien programme sont définies avec les mêmes valeurs dans le nouveau programme, les deux programmes ne sont pas strictement équivalents à cause des affectations supplémentaires. Nous devons donc introduire une notion de raffinement entre programmes Obc pour énoncer et démontrer la propriété de correction.

5.1 La nouvelle étape de compilation

Les fonctions pour ajouter les assertions de validité et affectations à une commande Obc sont illustrées dans la figure 8. La fonction *add_defaults_stmt* est appelée avec deux arguments : un ensemble[9] *required* de variables à initialiser et une commande *s* à transformer.

$$(s', required', sometimes, always) = add\_defaults\_stmt\ required\ s$$

Cette fonction rend s', une nouvelle commande, $required'$, l'ensemble des variables qui doivent être initialisées avant l'exécution de s', *sometimes*, l'ensemble des variables qui sont parfois mais pas toujours écrites par s' et *always*, l'ensemble des variables qui sont toujours écrites par s'[10].

Les cas pour `skip`, l'affectation à une variable et l'affectation à une variable d'état sont simples. Notons que le cas pour l'affectation à une variable enlève cette variable de l'ensemble *required* et l'ajoute à l'ensemble *always*. Pour les appels de méthode, les variables dans xs sont enlevés de l'ensemble *required* avant d'appliquer la fonction *add_valid* successivement sur les expressions d'argument. Des assertions de validité sont ajoutées pour toutes les variables qui n'en ont pas déjà et ces variables sont ajoutées à l'ensemble *required*. Autrement dit, nous

9. Nous représentons les ensembles dans Coq avec la bibliothèque MSetPositive.
10. Nous démontrons que sometimes ∩ always = ∅.

```
Variable type_of_var : ident ⟹ option type.

Definition add_write x s :=
  match type_of_var x with None ⇒ s | Some ty ⇒ (x := (init_type ty)) ; s end.

Definition add_writes W s := PS.fold add_write W s.

Definition add_valid (e : exp) (esreq : list exp * PS.t) :=
  match e, esreq with
  | Var x ty, (es, req) ⇒ (Valid e :: es, req ∪ {x})
  | _,        (es, req) ⇒ (      e :: es, req)
  end.

Fixpoint add_defaults_stmt (req: PS.t) (s: stmt) : stmt * PS.t * PS.t * PS.t :=
  match s with
  | skip ⇒ (s, req, ∅, ∅)
  | x := e ⇒ (s, req − {x}, ∅, {x})
  | st(x) := e ⇒ (s, req, ∅, ∅)

  | xs := f(o).m(es) ⇒
    let (es', req') := fold_right add_valid ([], ps_removes xs req) es
    in (xs := f(o).m(es'), req', ∅, of_list xs)

  | s1 ; s2 ⇒
    let (t2, req2, st2, al2) := add_defaults_stmt req s2 in
    let (t1, req1, st1, al1) := add_defaults_stmt req2 s1 in
    (t1 ; t2, req1, (st1 − al2) ∪ (st2 − al1), al1 ∪ al2)

  | if e { s1 } else { s2 } ⇒
    let (t1, req1, st1, al1) := add_defaults_stmt ∅ s1 in
    let (t2, req2, st2, al2) := add_defaults_stmt ∅ s2 in
    let (al1_req, al2_req) := (al1 ∩ req, al2 ∩ req) in
    let (w1, w2) := (al2_req − al1_req, al1_req − al2_req) in
    let w := ((st1 ∩ req) − w1) ∪ ((st2 ∩ req) − w2) in
    let (al1', al2') := (al1 ∪ w1, al2 ∪ w2) in
    let (st1', st2') := (st1 − w1, st2 − w2) in
    (add_writes w (if e { add_writes w1 t1 } else { add_writes w2 t2 }),
     (((req − al1_req) − al2_req) ∪ req1 ∪ req2) − w,
     (st1' ∪ st2' ∪ (al1' − al2') ∪ (al2' − al1')) − w,
     (al1' ∩ al2') ∪ w)
  end.
```

FIGURE 8 – Fonction principale de l'initialisation d'arguments dans un programme Obc.

affirmons qu'une variable est définie et l'ajoutons à l'ensemble de variables à initialiser. La liste de variables xs est transformée dans un ensemble *always*. Dans le cas de la composition séquentielle, on travail en arrière, en propageant les ensembles *required*, avant de recalculer les ensembles *sometimes* et *always*.

Le cas pour les instructions conditionnelles est le plus complexe. Il y a un appel récursif pour la commande dans chaque branche avec l'ensemble vide pour l'argument *required*. Sont calculés, l'ensemble *w1* des initialisations à faire avant la commande *s1*, l'ensemble *w2* des initialisations à faire avant la commande *s2* et l'ensemble *w* des initialisations à faire avant les deux branches. Dans la première branche, on ajoute des affectations aux variables dans *required* qui sont toujours écrites par *s2* mais qui ne sont pas toujours écrites par *s1*. La seconde branche est transformée de façon similaire. Les variables dans *required* qui sont parfois mais pas toujours écrites dans les deux branches sont initialisées avant la nouvelle instruction conditionnelle, en prenant en compte les autres nouvelles initialisations. Il y a une fonction auxiliaire, *type_of_var*,

qui associe une variable à son type, et une autre, *init_type*, qui associe un type à sa valeur par défaut, et encore d'autres pour calculer les ensembles *required*, *sometimes* et *always*.

La fonction *add_defaults_stmt* est utilisée dans la fonction qui transforme les méthodes.

```
add_defaults_method (m : method) : method
```

Elle construit l'argument *type_of_var* à partir des déclarations de *m*, transforme le corps de la méthode avec *add_defaults_stmt* en passant l'ensemble des variables de sortie comme *req* et ajoute les initialisations qui restent à faire après exclusion des variables d'entrée.

```
let (body', req, st, al) := add_defaults_stmt tyenv (ps_adds (map fst
outs) ∅) body in
add_writes tyenv (ps_removes (map fst ins) req) body')
```

La fonction *add_defaults_stmt* est conçue pour traiter la structure du code Obc généré de programmes Lustre-N et transformé par l'optimisation de fusion. Elle exprime un compromis entre l'ajout d'initialisations inutiles et le nombre d'affections supplémentaires à faire. D'un côté, l'ajout d'affectations aux feuilles d'un arbre d'instructions conditionnelles pourrait augmenter énormément la taille de code, de l'autre, leur ajout inconsidéré aux sommets peut augmenter inutilement le nombre d'affectations. La fonction présentée ci-dessus n'est pas toujours optimale, mais un meilleur résultat demanderait une analyse plus compliquée qui ne ferait que reconstruire en Obc l'information encodée par les horloges statiques dans le programme source.

Les assertions de validité et les initialisations ajoutées pour l'exemple sont indiquées en rouge dans la figure 3. Pour chaque nouvelle assertion de validité aux lignes 23, 35, et 36 une initialisation est ajoutée auparavant dans le programme. Puisque `h` est une entrée, nous supposons par récurrence que l'appelant l'a déjà initialisé. Une initialisation est ajoutée pour la variable de sortie `o2` qui n'est pas forcément écrite sinon. Ainsi, on peut supposer que les variables définies par un appel de méthode sont toujours initialisées, ce qui est nécessaire dans la preuve de correction d'*add_defaults_stmt*.

5.2 La relation de raffinement

La commande s' générée par *add_defaults_stmt* ne calcul pas forcément les mêmes résultats que la commande initiale s. Intuitivement, on s'attend que toutes les variables définies après exécution de s soient définies avec les mêmes valeurs après exécution de s'. Les valeurs des variables nouvellement définies ne sont pas importantes pourvue que les entrées et les sorties du nœud principal en Lustre-N (le nœud « main ») sont toutes sur l'horloge de base et donc toujours définies par le code généré. Nous exprimons ces intuitions formellement avec une relation de raffinement entre environnements.

$$env_2 \sqsubseteq env_1 \equiv \forall x\ v,\ env_2\ x = \mathsf{Some}\ v \implies env_1\ x = \mathsf{Some}\ v$$

Ensuite, nous entendrons la relation aux commandes exécutées dans deux programmes p_1 et p_2. L'idée est que l'exécution d'une commande « enrichie » s_1 dans un environnement env_1 qui est « plus défini » que env_2 produit un environnement identique à celui produit par la commande initiale s_2 exécuté dans env_2 sauf qu'il peut y avoir plus de variables définies. Malheureusement, une précondition P s'avère nécessaire pour démontrer le raffinement entre deux commandes et la relation est plus compliquée que l'on pourrait souhaiter.

$$\begin{aligned} s_2 \sqsubseteq_P^{p_2,p_1} s_1 \equiv\ & \forall menv\ menv'\ env_1\ env_2\ env_2', \\ & P\ env_1\ env_2 \implies \\ & env_2 \sqsubseteq env_1 \implies \\ & p_2, menv, env_2 \vdash s_2 \Downarrow (menv', env_2') \implies \\ & \exists env_1',\ p_1, menv, env_1 \vdash s_1 \Downarrow (menv', env_1')\ \wedge\ env_2' \sqsubseteq env_1' \end{aligned}$$

La relation s'étend directement aux méthodes et aux classes. On aboutit finalement à une définition récursive de raffinement entre programmes (défini de façon standard par un combinateur de point fixe et une relation bien fondée).

$$p_2 \sqsubseteq_P p_1 \equiv \forall n\ c_2\ p_2',\ \mathsf{find\text{-}class}\ n\ p_2 = \mathsf{Some}\ (c_2, p_2') \implies \exists c_1\ p_1',\ \mathsf{find\text{-}class}\ n\ p_1 = \mathsf{Some}\ (c_1, p_1') \ \wedge\ c_1 \sqsubseteq^{p_1,p_2}_{(P\ n)} c_2 \ \wedge\ p_2' \sqsubseteq_P p_1'$$

5.3 Les invariants et la preuve de correction

La relation de raffinement permet d'énoncer succinctement le cœur du lemme de correction principal.

$$\mathit{add\_defaults\_stmt}\ \mathit{tyenv}\ \mathit{req}\ s = (t,\ \mathit{req}',\ \mathit{st},\ \mathit{al}) \implies s \sqsubseteq^{p,p'}_{(\mathit{in1\_notin2}\ \mathit{req}'\ (\mathit{st} \cup \mathit{al}))} t$$

La précondition *in1_notin2* exige que les variables dans *req′* soient définies dans l'environnement initial de t — elles sont initialisées par *add_defaults_method* — et que les variables écrites dans t ne soient pas définies dans l'environnement initial de s — c'est le cas, car les variables d'entrée sont enlevées par *add_defaults_method* et ne sont donc jamais réécrites. Quelques hypothèses supplémentaires sont nécessaires pour démontrer le raffinement, à savoir qu'il y a un raffinement entre *p′* et *p*, que les méthodes des classes dans *p* définissent toutes leurs sorties si toutes leurs entrées sont définies et que *s* est bien typé et satisfait l'invariant NoOverwrites. Le prédicat NoOverwrites est présenté dans la figure 9. Il affirme, essentiellement, qu'un programme est en « static single assignment form (SSA) », c'est-à-dire, que les variables ne sont écrites qu'une fois au plus. La règle importante s'appelle NoO-Seq. Ce prédicat est indispensable pour démontrer que les initialisations ajoutées par *add_defaults_stmt* n'écrasent jamais les valeurs existantes. Il est vrai du code généré des programmes Lustre-N et préservé par l'optimisation de fusion. Par contre, ce prédicat n'est pas préservé par la fonction *add_defaults_stmt* elle-même.

Le code produit par *add_defaults_stmt* satisfait le prédicat NoNakedVars. Ce prédicat est aussi présenté dans la figure 9. Il affirme simplement que les variables ne sont jamais passées directement dans un argument de méthode. Il garantit l'existence d'un modèle sémantique pour le code Clight généré par la suite. Sinon les preuves en aval ne demandent que quelques modifications mineures et l'addition d'un contrôle pour assurer que les entrées et les sorties du nœud principal (le nœud « main ») sont toutes sur l'horloge de base.

6 Conclusion

Cet article décrit l'extension d'un compilateur Lustre vérifié pour qu'il traite les définitions de nœuds où certaines entrées ou sorties sont absentes lors d'une activation — c'est-à-dire, d'accepter l'utilisation de sous-horloges dans les interfaces de nœud. Nous décrivons une règle d'horloge pour l'instanciation d'un nœud en Lustre normalisé et une condition sur les arguments d'un nœud qui suffissent pour démontrer la correction du code généré. Nous modifions les règles de sémantique de notre langage intermédiaire pour permettre l'utilisation de variables non définies dans les appels de méthode. Cependant, de tels appels ne peut pas être convertis directement en Clight, car la sémantique de Clight respect la norme C99 qui ne permet pas l'utilisation d'arguments non initialisés dans un appel de fonction. Nous avons réglé ce problème par (a) l'ajout d'assertions dans le code généré pour affirmer que certaines variables sont sûrement bien définies au point de leur utilisation avec un raisonnement qui s'appuie sur la preuve de correction existante (*b*) la définition et la vérification d'une nouvelle étape de compilation pour ajouter les assertions qui manquent et les initialisations correspondantes.

$$\textsc{CWI-Assign}\ \ \mathsf{CanWrite}\ x\ (x := e) \qquad \textsc{CWI-AssignSt}\ \ \mathsf{CanWrite}\ x\ (\mathtt{st}(x) := e)$$

$$\textsc{CWI-Ifte-T}\ \ \frac{\mathsf{CanWrite}\ x\ s_1}{\mathsf{CanWrite}\ x\ (\mathtt{if}\ e\ \{s_1\}\ \mathtt{else}\ \{s_2\})}$$

$$\textsc{CWI-Ifte-F}\ \ \frac{\mathsf{CanWrite}\ x\ s_2}{\mathsf{CanWrite}\ x\ (\mathtt{if}\ e\ \{s_1\}\ \mathtt{else}\ \{s_2\})} \qquad \textsc{CWI-Seq-1}\ \ \frac{\mathsf{CanWrite}\ x\ s_1}{\mathsf{CanWrite}\ x\ (s_1\ ;\ s_2)} \qquad \textsc{CWI-Seq-2}\ \ \frac{\mathsf{CanWrite}\ x\ s_2}{\mathsf{CanWrite}\ x\ (s_1\ ;\ s_2)}$$

$$\textsc{CWI-Call}\ \ \frac{\mathsf{In}\ x\ xs}{\mathsf{CanWrite}\ x\ (xs := cls(i).f(es))} \qquad \textsc{NoO-Assign}\ \ \mathsf{NoOverwrites}\ (x := e) \qquad \textsc{NoO-Assignst}\ \ \mathsf{NoOverwrites}\ (\mathtt{st}(x) := e)$$

$$\textsc{NoO-Skip}\ \ \mathsf{NoOverwrites}\ \mathtt{skip} \qquad \textsc{NoO-Ifte}\ \ \frac{\mathsf{NoOverwrites}\ s_1 \qquad \mathsf{NoOverwrites}\ s_2}{\mathsf{NoOverwrites}\ (\mathtt{if}\ e\ \{s_1\}\ \mathtt{else}\ \{s_2\})}$$

$$\textsc{NoO-Seq}\ \ \frac{\begin{array}{c}(\forall x, \mathsf{CanWrite}\ x\ s_1 \implies \neg\mathsf{CanWrite}\ x\ s_2) \\ (\forall x, \mathsf{CanWrite}\ x\ s_2 \implies \neg\mathsf{CanWrite}\ x\ s_1) \qquad \mathsf{NoOverwrites}\ s_1 \qquad \mathsf{NoOverwrites}\ s_2\end{array}}{\mathsf{NoOverwrites}\ (s_1\ ;\ s_2)}$$

$$\textsc{NoO-Call}\ \ \mathsf{NoOverwrites}\ (xs := cls(i).f(es)) \qquad \textsc{NNV-Assign}\ \ \mathsf{NoNakedVars}\ (x := e) \qquad \textsc{NNV-Assignst}\ \ \mathsf{NoNakedVars}\ (\mathtt{st}(x) := e)$$

$$\textsc{NNV-Skip}\ \ \mathsf{NoNakedVars}\ \mathtt{skip} \qquad \textsc{NNV-Ifte}\ \ \frac{\mathsf{NoNakedVars}\ s_1 \qquad \mathsf{NoNakedVars}\ s_2}{\mathsf{NoNakedVars}\ (\mathtt{if}\ e\ \{s_1\}\ \mathtt{else}\ \{s_2\})}$$

$$\textsc{NNV-Seq}\ \ \frac{\mathsf{NoNakedVars}\ s_1 \qquad \mathsf{NoNakedVars}\ s_2}{\mathsf{NoNakedVars}\ (s_1\ ;\ s_2)} \qquad \textsc{NNV-Call}\ \ \frac{\mathsf{Forall}\ (\lambda e,\ e \neq x)\ es}{\mathsf{NoNakedVars}\ (xs := cls(i).f(es))}$$

FIGURE 9 – Invariants Obc : CanWrite, NoOverwrites et NoNakedVars.

Remerciements Nous remercions J.-L. Colaço pour ses explications sur Scade, X. Leroy pour ses explications et ses conseils sur CompCert et A. Guatto et L. Mandel pour leurs suggestions. Ce travail a été soutenu par le projet ITEA 3 14014 ASSUME.

Références

[1] D. Biernacki, J.-L. Colaço, G. Hamon et M. Pouzet : Clock-directed modular code generation for synchronous data-flow languages. *In Proc. 9th ACM SIGPLAN Conf. on Languages, Compilers, and Tools for Embedded Systems (LCTES 2008)*, p. 121–130, Tucson, AZ, USA, juin 2008. ACM Press.

[2] S. Blazy et X. Leroy : Mechanized semantics for the Clight subset of the C language. *J. Automated Reasoning*, 43(3):263–288, oct. 2009.

[3] S. Boulmé et G. Hamon : Certifying synchrony for free. *In* R. Nieuwenhuis et A. Voronkov, éds : *Proc. 8th Int. Conf. on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR 2001)*, vol. 2250 de *LNCS*, p. 495–506, Havana, Cuba, déc. 2001. Springer.

[4] S. Boulmé et G. Hamon : A clocked denotational semantics for Lucid-Synchrone in Coq. Rap. tech., LIP6, nov. 2001.

[5] T. Bourke, L. Brun, P.-É. Dagand, X. Leroy, M. Pouzet et L. Rieg : A formally verified compiler for Lustre. *In Proc. 2017 ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, p. 586–601, Barcelona, Spain, juin 2017. ACM Press.

[6] T. Bourke, P.-É. Dagand, M. Pouzet et L. Rieg : Vérification de la génération modulaire du code impératif pour Lustre. *In* J. Signoles et S. Boldo, éds : *28^{ièmes} Journées Francophones des Langages Applicatifs (JFLA 2017)*, p. 165–179, Gourette, Pyrénées, France, jan. 2017.

[7] P. Caspi : Clocks in dataflow languages. *Theor. Comp. Sci.*, 94(1):125–140, mars 1992.

[8] J.-L. Colaço : Private communication, nov. 2017.

[9] J.-L. Colaço, B. Pagano et M. Pouzet : Scade 6 : A formal language for embedded critical software development. *In Proc. 11th Int. Symp. Theoretical Aspects of Software Engineering (TASE 2017)*, Nice, France, sept. 2017. IEEE Computer Society.

[10] J.-L. Colaço et M. Pouzet : Clocks as first class abstract types. *In* R. Alur et I. Lee, éds : *Proc. 3rd Int. Conf. on Embedded Software (EMSOFT 2003)*, vol. 2855 de *LNCS*, p. 134–155, Philadelphia, PA, USA, oct. 2003. Springer.

[11] N. Halbwachs, P. Caspi, P. Raymond et D. Pilaud : The synchronous dataflow programming language LUSTRE. *Proc. IEEE*, 79(9):1305–1320, sept. 1991.

[12] Programming languages—C. Standard, ISO/IEC, Geneva, Switzerland, déc. 1999.

[13] R. Kumar, M. O. Myreen, M. Norrish et S. Owens : CakeML : A verified implementation of ML. *In Proc. 41st ACM SIGPLAN-SIGACT Symp. Principles of Programming Languages (POPL 2014)*, p. 179–191, San Diego, CA, USA, jan. 2014. ACM Press.

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

[15] The Coq Development Team : *The Coq proof assistant reference manual*. Inria, 2018. v. 8.8.