

Rapport de Stage

-

Traduction de composants Scade/Lustre vers des machines B

FLORIAN THIBORD

14 août 2013

# Table des matières

<b>1</b>	<b>Introduction</b>	<b>2</b>
<b>2</b>	<b>Scade</b>	<b>4</b>
2.1	Architecture d'un composant Scade . . . . .	4
2.2	Restrictions . . . . .	5
2.2.1	Le temps avec Scade . . . . .	5
2.2.2	Contrats . . . . .	6
<b>3</b>	<b>Machines B</b>	<b>7</b>
3.1	Machine B . . . . .	7
3.1.1	Structure d'une machine . . . . .	7
3.1.2	Clause . . . . .	8
3.1.3	Prédicats . . . . .	8
3.2	Expressions . . . . .	9
3.3	Substitutions . . . . .	9
3.4	Raffinements . . . . .	11
3.4.1	Principes du raffinement . . . . .	11
3.4.2	Obligations de preuves . . . . .	11
<b>4</b>	<b>Schémas de traduction</b>	<b>12</b>
4.1	Specification . . . . .	12
4.2	Implémentation . . . . .	12
4.2.1	Traduction des expressions . . . . .	12
4.3	Le traducteur . . . . .	14
<b>5</b>	<b>Exemples tests</b>	<b>15</b>
<b>6</b>	<b>Preuve de correction de la traduction</b>	<b>16</b>
<b>7</b>	<b>Conclusion</b>	<b>17</b>

# Chapitre 1

## Introduction

Ce stage s'est déroulé au sein du projet *CERCLES*<sup>2 1</sup>, financé par l'Agence Nationale de la Recherche. L'objectif du projet est de certifier formellement des composants réutilisables, l'intérêt étant de réduire les tests en prouvant grâce à des méthodes formelles la sûreté des différentes briques formant un logiciel. L'intérêt est à la fois pratique par la réutilisabilité des composants certifiés, et économique en réduisant le temps et le coût des tests.

Un acteur majeur du développement de systèmes embarqués critiques est Scade, un acronyme pour Safety Critical Application Development Environment. Cet environnement de développement est basé sur la programmation graphique, par schémas-blocs, permettant de définir des programmes faciles à lire et permettant d'engendrer directement du code compilable (C ou ADA). Il est notamment utilisé en aéronautique (grande partie du logiciel embarqué de l'A380), dans le domaine spatial ou dans le nucléaire. Dans le cadre du projet, c'est avec Scade que sont écrits les composants.

Pour assurer que ces composants sont sûrs et réutilisables, on utilise une méthode formelle, qui permet d'exprimer la signification d'un composant dans un formalisme mathématique, afin de démontrer leur validité par rapport à une spécification.

Il faut alors introduire le concept des contrats : un contrat est associé à un composant et impose des conditions sur ses entrées (pré-conditions) et sur ses sorties (post-conditions). Ils formeront ainsi une spécification du composant. Les contrats ont été introduits par C.A.R Hoare, qui donne la définition suivante : Soit  $P$  et  $R$  les pré-conditions et post-conditions associées au programme  $Q$ ,

$$P\{Q\}R$$

*"If the assertion  $P$  is true before initiation of a program  $Q$ , then the assertion  $R$  will be true on its completion"*

A partir d'un programme et de son contrat, il faut alors vérifier formellement que :

- (i) Le programme est cohérent vis-à-vis de sa spécification.
- (ii) l'initialisation du programme satisfait les pré-condition, et en conséquence de (i) le résultat satisfait les post-conditions.

La validation est alors faite par une démonstration formelle.

Il existe différentes approches de démonstrations formelles, comme celle basée sur des règles de typage des programmes, introduites par la correspondance de Curry-Howard dans à la fin des années 50. L'avantage de la méthode choisie, la méthode B, est qu'elle a déjà fait ses preuves industriellement,

---

1. CERTification Compositionnelle des Logiciels Embarqués critiqueS et Sûrs

elle a notamment été utilisée pour développer la ligne METEOR (ligne 14) du métro parisien, qui est entièrement automatisée.

Elle a été introduite par J.R. Abrial dans les années 80. Elle est basée sur le raffinement de spécifications formelles vers une spécification exécutable. La spécification formelle est rédigée dans un formalisme mathématique de haut niveau appelé machine abstraite, dont le principe de calcul est basé sur le calcul des prédicats du premier ordre étendu avec une théorie des ensembles. Le raffinement de cette machine abstraite consiste à la reformuler de façon plus concrète et à l'enrichir avec des substitutions correspondantes aux instructions du composant. Le raffinement de plus bas niveau, exécutable, est appelé implantation. Il peut y avoir des raffinements intermédiaires, mais nous n'aurons besoin que d'une étape de raffinement de la machine abstraite vers l'implantation. Chaque étape de raffinement passe par une étape d'obligations de preuves, une validation par démonstration formelle, garantissant la fidélité de la machine raffinée par rapport à la machine abstraite.

Mon travail fut de développer un traducteur permettant de transposer un composant écrit en SCADE vers B. Le traducteur suit une ligne de compilation classique, prenant en entrée un code issu de SCADE et produisant en sortie une machine abstraite correspondant aux spécifications du contrat, ainsi que la machine raffinée qui implante le programme.

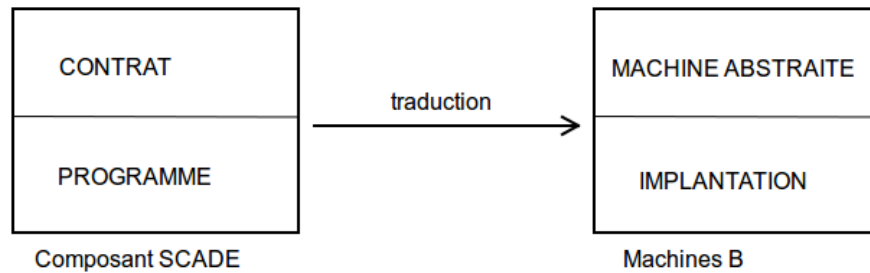


FIGURE 1.1 – Schéma du principe de traduction

## Chapitre 2

# Scade

Scade a été développé par le laboratoire Verimag à partir des travaux sur le langage synchrone Lustre, puis repris par Esterel-technologie. On retrouve ainsi les notions de Lustre dans le langage de Scade, un programme est découpé en noeuds dont les entrées et sorties sont des flux de données. Ces noeuds sont les composants que nous voulons traduire. Les noeuds Scade considérés dans le cadre du projet *Cercles*<sup>2</sup> sont soumis à quelques restrictions. En effet, il faut limiter le langage utilisé, car certains éléments du langage sont spécifiques aux langages synchrones et ne sont donc pas traduisibles en B.

### 2.1 Architecture d'un composant Scade

Scade étant un environnement de programmation par schémas-blocs, on développe avec des boîtes. Par exemple, un programme d'addition sur deux flux d'entiers A et B ayant pour sortie un flux C s'écrit :

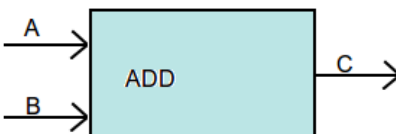


FIGURE 2.1 – Schéma-block de l'addition

Ce langage en boîtes est basé sur le langage synchrone Lustre, et c'est le programme en Lustre que nous allons parser avec le traducteur. La représentation graphique du programme est ainsi écrite en Lustre avant d'être traduite. C'est donc du code Lustre que l'on traduit.

```
node ADD (A:int, B:int) returns (C:int);  
let  
    C = A+B;  
tel
```

Les composants que nous voulons traduire sont formés d'un unique noeud, dont la définition contient autant d'entrée et de sorties que nécessaire. Il n'y a pas non plus de restriction sur le nombre de va-

riables locales utilisables.

Au niveau des types de données utilisées, on pourra manipuler des entiers, réels et booléens. Et on pourra également manipuler des tableaux de ces types. En revanche, les types définis par l'utilisateur tels que les types enregistrement ne seront pas gérés par le traducteur.

Le comportement du noeud est ensuite défini par une liste d'équations, dont l'ordre n'a pas d'importance. Ces équations sont de la forme :

```
left_part = expr;
```

Où `left_part` désigne une variable locale ou une sortie du composant, et `expr` est une expression portant sur une ou plusieurs variables locales ou entrées.

Les expressions disponibles sont toutes les expressions arithmétiques (+, -, /, \*, mod), les expressions relationnelles (<, >, <=, >=, =, <>) et logiques (and, or, xor, not). Les expressions conditionnelles sont également possibles (if .. then .. else ..). On peut également faire des appels à d'autres noeuds, sous condition qu'ils aient été traduits auparavant.

Sont également disponibles les opérations sur les tableaux, telles que la définition, l'index, et la concaténation.

## 2.2 Restrictions

### 2.2.1 Le temps avec Scade

Le temps est un élément primordial dans ces systèmes dits réactifs, où on manipule des flux de données. Le temps est discrétisé en instants, et chaque instant correspond à 1 tic de l'horloge de base. A chaque instant  $i$ , les équations du noeud sont résolues à partir du flux reçu en entrée à cet instant, et produit le flux de sortie correspondant au résultat.

**Une horloge unique** Ainsi, la première restriction à noter est qu'il n'y a qu'une seule horloge, celle de base. Avec les langages synchrones, on peut synchroniser des instructions sur des horloges différentes avec, mais dans les programmes que nous manipulerons, il n'y aura donc pas de définitions d'horloges, d'utilisation de `when`, ou `current`. Les instructions d'un noeud sont toutes calculées sur l'horloge de base uniquement.

**Des registres** La seconde restriction concerne l'utilisation des opérateurs `pre` et `->`. Ce sont des opérateurs temporels :

- `pre X` donne la valeur de l'expression  $X$  à l'instant précédent. A l'instant  $0^1$ , la valeur de `pre X` n'est pas définie.
- `A -> B` donne au premier instant la valeur de l'expression  $A$ , puis la valeur de l'expression  $B$  pour les instants allant de 1 à  $n$ .

On ne pourra utiliser que la construction suivante utilisant ces deux opérateurs :

$$A \rightarrow (pre\ X)$$

---

1. On suppose que le premier instant est l'instant 0

Cette construction correspond au bloc SIMULINK  $1/Z^2$ , où A représente un flux constant qui donnera la valeur de sortie à l'instant 0 du bloc. Puis pour les instants 1 à n, on aura la valeur de l'expression X à l'instant (1 à n)-1.

On appellera cette construction un registre, qui est initialisé avec la valeur A, et qui permet d'accéder à la valeur précédente de X à tout instant. Cette construction permet de donner un état à un composant.

### 2.2.2 Contrats

**Assertions** On peut définir des assertions dans un noeud afin de poser des restrictions sur les valeurs d'entrée ou de sortie du composant. Avec Scade, ces assertions sont possibles avec :

- *assume*  $x : expr$ , où x est une des entrées du noeud, et expr un prédicat portant sur cette entrée.
- *guarantee*  $x : expr$ , où x est une des sorties du noeud, et expr un prédicat portant sur cette sortie.

Ces assertions forment le contrat du composant, et seront obligatoires sauf pour la restriction sur les booléen qui est triviale (la valeur sera vraie ou fausse).

Par exemple, en reprenant le noeud ADD précédent, on impose comme condition sur les entrées qu'elles doivent être comprises entre 0 et 100 inclus. Si les préconditions sont respectées, alors la sortie sera comprise entre 0 et 200 inclus :

```
node ADD (A:int, B:int) returns (C:int);
let
  assume A : A <= 100 & A >= 0;
  assume B : B <= 100 & B >= 0;
  guarantee C : C <= 200 & C >= 0;
  C = A+B;
tel
```

## Chapitre 3

# Machines B

Concernant le langage B, langage de sortie du traducteur, nous n'aurons besoin d'utiliser que les éléments nécessaires pour exprimer les éléments de Scade en B, et pour certifier formellement le composant ainsi traduit.

La méthode B s'appuie sur un raisonnement mathématique rigoureux, basé sur des étapes de raffinements. Nous n'aurons besoin que d'une étape de raffinement pour notre traducteur. Il faudra ainsi produire deux machines en sortie du traducteur :

- une signature : elle correspond à la machine abstraite qui reprend les éléments de spécification du composant traduit.
- une implantation : elle raffine la machine abstraite et contient les substitutions correspondant à l'opération définie dans le composant.

La méthode B est utilisée avec l'environnement de développement AtelierB, développé par Clearsy. Nous utiliserons cet environnement pour vérifier que le code traduit est correctement traduit en B à l'aide d'un analyseur syntaxique intégré ainsi qu'un type checkeur. On utilise ensuite l'environnement pour générer les obligations de preuves liées au couple de machines.

### 3.1 Machine B

#### 3.1.1 Structure d'une machine

Une machine B est divisée en *clauses*, que l'on peut assimiler à des services permettant l'initialisation puis l'évolution des données manipulées. Une clause ne peut-être utilisée plus d'une fois dans une machine, mais l'ordre n'est pas imposé. Il en existe une vingtaine, mais nous n'en utiliserons que sept, que nous détaillerons dans la partie suivante.

Ces données sont exprimées dans le même type qui est utilisé avec Scade, c'est à dire soit des entiers, soit des réels, soit des booléens, soit des tableaux de ces types, qui sont des types primitifs du langage B.

La machine abstraite reprenant la spécification du composant contient des prédicats portant sur ces données, et la transformation de ces prédicats se fait grâce à un mécanisme de substitutions. Les différentes substitutions requises pour ce projet seront détaillées dans la partie correspondante.

Une machine est précédée d'un en-tête qui diffère selon la machine abstraite et l'implantation :

- pour la signature, l'en-tête sera composé du mot MACHINE suivi du nom du composant.
- pour l'implantation, ce sera IMPLEMENTATION suivi du nom du composant auquel on ajoute le suffixe "\_i".

Voir annexe A pour un exemple de couple de machines abstraite/implantation.



### 3.1.2 Clause

Les différentes clauses requises pour assurer la traduction sont décrites dans cette partie. La machine abstraite ne requière pas les clauses `IMPORTS`, `CONCRETE_VARIABLES`, `INVARIANT` et `INITIALISATION` car elle ne manipule que la spécification. En revanche, l'implémentation ne manipule que des données et substitutions ayant un équivalent informatique, similaire à un langage impératif, et on aura besoin de ces clauses pour exprimer l'opération définie dans le composant Scade.

**Refines** La clause `refines` est présente dans la machine implémentation afin d'indiquer la machine qui est raffinée. Nous ne faisons qu'une étape de raffinement, donc la machine raffinée sera toujours la machine abstraite.

**Imports** Ici, nous indiquons quelles machines B seront nécessaires pour manipuler les données. Pour la programmation par composant, nous avons besoin de faire appel à d'autres composants, et cette clause permet d'importer une instance de ces composants.

**Sees** Nous avons besoin de faire aussi appel à des machines contenant des définitions de constantes. Nous mettrons la liste des machines nécessaires dans cette clause.

**Concrete\_Variables** Cette clause indique quelles sont les variables d'état de la machine. C'est dans cette clause que nous déclarons les registres définis dans le composant Scade. Les autres variables locales sont définies dans une substitution `VAR`.

**Invariant** Nous pouvons alors établir des invariants sur les registres utilisés dans cette clause. Les invariants seront des restrictions sur les intervalles sur lesquels les registres seront manipulés, et ils seront écrits sous forme de prédicats.

**Initialisation** L'initialisation permet d'indiquer la valeur donnée aux registres lors de l'initialisation du composant. L'initialisation doit être en accord avec l'invariant.

**Opérations** La clause principale d'une machine B est la clause `Operations`. On y définit la spécification du composant dans la machine abstraite, et cette spécification est concrétisée dans l'implantation où on écrira les expressions du composant sous forme de substitutions et de prédicats. Bien qu'on puisse définir autant d'opération qu'on le souhaite dans cette clause, nous ne définirons qu'une seule opération, celle correspondant au composant Scade.

### 3.1.3 Prédicats

Un prédicat est une formule mathématique qui peut être prouvée ou réfutée. Elle peut être présente pour exprimer des propriétés sur une donnée, comme dans la clause *Invariant*, ou dans la substitution *Precondition*. Elle peut-être aussi utilisée pour exprimer une condition, comme dans la substitution *Condition*.

Les prédicats de base sont exprimables à l'aide des opérateurs de comparaison habituels :  $<$ ,  $>$ ,  $\leq$ , et  $\geq$ . Les expressions doivent être de type entier ou réel.

Pour exprimer un prédicat plus complexe à partir de prédicats basique, on utilise des opérateurs de proposition : conjonction  $\vee$ , disjonction  $\wedge$ , négation  $\neg$ , parenthèses  $()$ , implication  $\Rightarrow$  et équivalence  $\Leftrightarrow$ .

Par exemple, soit  $P$  et  $Q$  des prédicats :  $P \wedge \neg(Q)$

On utilisera aussi le quantificateur  $\forall$ , notamment lorsqu'on définira des tableaux, pour établir une condition pour tous les éléments du tableau. Et on aura besoin de l'opérateur d'appartenance à un ensemble  $\in$ .

Par exemple, soit  $tab$  un tableau :  $\forall iii.(iii \in (1..5) \Rightarrow tab(iii) < 5)$

Cet exemple indique que tout élément du tableau ayant un indice compris entre 1 et 5 doit être strictement inférieur à 5. Nous reviendrons en détail sur le fonctionnement des tableaux en B dans la section sur les expressions.

## 3.2 Expressions

Les expressions permettent de désigner les données utilisées. Les expressions de base désignent une variable ou une valeur primitive. On retrouve toutes les expressions arithmétiques classiques, addition, soustraction, multiplication, division et modulo.

**Des fonctions** Nous utiliserons également les fonctions, pour appeler des opérations définies dans d'autres composants, mais aussi pour modéliser les tableaux en B. Il n'y a pas de type primitif tableau en B, il faut les modéliser à l'aide de fonctions.

**Les tableaux en B** Un tableau est une fonction dont l'ensemble de départ est le produit cartésien de  $n$  ensembles (où  $n$  correspond au nombre de dimensions du tableau), et dont l'ensemble d'arrivée est un ensemble concret.

Par exemple, soit  $tab$  un tableau,

$tab \in (0..4) * (0..5) \rightarrow INT$

est une matrice de 5 lignes et 6 colonnes contenant des entiers.

## 3.3 Substitutions

Les substitutions permettent de transformer les prédicats, il en existe 18 mais nous ne nous intéresseront qu'à 7 d'entre elles. Soit une substitution  $S$  et un prédicat  $P$ ,  $[S/P]$  se lit "la substitution  $S$  établit le prédicat  $P$ ". Les substitutions ne sont présentes que dans les clauses Initialisation et Operations. Dans la partie suivante, on détaillera comment passer des équations de Scade à ces substitutions.

**Substitution Bloc** La substitution bloc est la substitution de base, elle permet de grouper des substitutions. Elles contiennent une substitution ou une séquence de substitution. Elle se note : Soit  $S$  une substitution,

*BEGIN S END*

**Substitution Sequence** Une séquence permet d'appliquer en séquence deux substitutions à un prédicat. Les deux substitutions sont séparées par un ;.

**Substitution Parallèle** A la différence de la substitution séquence, la substitution parallèle permet d'effectuer deux substitutions de façon simultanée et indépendamment l'une de l'autre. Les deux substitutions sont séparées par  $||$ .

**Substitution Devient égal** Cette substitution réalise l'affectation, elle remplace une variable par une expression. Elle se note : Soit  $e$  une expression,  $x$  une variable et  $P$  un prédicat,

$[x := e]P$

Le prédicat obtenu a alors toute les occurrences libre de  $x$  dans  $P$  par  $e$ .

**Substitution Condition** C'est cette substitution que l'on utilise pour exprimer le choix entre deux substitutions. Elle se note : Soit  $P$  et  $R$  des prédicats, et  $S$  et  $T$  des substitutions,

$[IF P THEN S ELSE T]R$

Si le prédicat  $P$  est évalué à vrai, alors c'est la substitution  $S$  que l'on applique au prédicat  $R$ . Si  $P$  est faux, alors c'est la substitution  $T$  qui s'applique à  $R$ .

**Substitution variable locale** Cette substitution n'est pas utilisée dans la machine abstraite. Elle permet d'introduire une liste de variables locales. Elle se note : Soit  $S$  une séquence de substitutions et  $X$  une liste de variables,

$VAR X IN S END$

La liste de variable sera accessible dans les substitutions  $S$  contenues dans le bloc  $IN \dots END$ , correspondant à une substitution bloc.

**Substitution Precondition** Cette substitution n'est utilisée que dans la machine abstraite. Elle fixe les préconditions sous lesquelles une substitution. Elle se note : Soit  $P$  et  $R$  des prédicats et  $S$  une substitution,

$[PRE P THEN S END]R$

L'application de cette substitution correspond à la preuve de la précondition  $P$  et à l'application de la substitution  $S$ . Si la précondition  $P$  est fausse, le résultat de la substitution n'est alors plus garanti.

**Substitution Appel operation** cette substitution se note : Soit  $R$  un identificateur correspondant à la sortie de l'opération  $op$  appliquée aux expressions  $E$ , et un prédicat  $P$ ,

$[R \leftarrow op(E)]P$

La substitution appel d'opération permet d'appliquer la substitution d'une opération (non locale ou locale), en remplaçant les paramètres formels par des paramètres effectifs.

On retrouve ainsi les constructions d'un langage de programmation impératif, avec des affectations, appels d'opération, alternative, et la définition de variables locales. De plus, les substitutions sont réordonnées via un tri topologique par rapport aux équations du composant Scade, dont l'ordre n'avait pas d'importance. Cette distinction entre les deux langages, synchrone contre impératif, sera développée dans la section concernant la preuve de correction du traducteur.

## 3.4 Raffinements

### 3.4.1 Principes du raffinement

Le raffinement de la spécification est une reformulation en une expression plus concrète et enrichie. L'activité de preuve d'une machine B consiste à la réalisation d'un certain nombre de démonstrations, afin de prouver l'établissement et la conservation des propriétés invariantes de la spécification. La génération des hypothèses à démontrer est automatique, grâce à la transformation des prédicats par les substitutions.

La relation de raffinement est transitive : les valeurs calculées par l'implémentation sont conformes à celles attendues par la machine abstraite.

[ Raffinement jusqu'à un code exécutable après une compilation vers du code C ou Ada. Donc vers un programme équivalent à celui écrit avec Scade, qui est également compilé vers du C en fin de chaîne. Cependant nous ne nous intéresserons pas au programme produit par l'atelierB, car le compilateur de Scade produisant le C est certifié pour la norme DO178.

-

L'atelier B produit automatiquement les obligations de preuves nécessaires pour valider le raffinement. Ces obligations sont construites ...

Cf intro obligation proof paper

|

### 3.4.2 Obligations de preuves

Cf Proof obligation paper

A chaque raffinement, on passe par une étape de type checking et d'obligation de preuves. Démonstrations basée sur les principales substitutions. Montrer qu'on utilise ce qu'il faut pour assurer que les obligations de preuves sont possibles.

Principe :

Hypothèses (liste de prédicats)  $\Rightarrow$  But (doit être prouvé sous ces hypothèses)

Pour cela on applique des substitutions aux prédicats. Développer pour les différentes substitutions, exemples...

## Chapitre 4

# Schémas de traduction

Dans les deux parties précédentes, nous avons identifié les différents éléments de Scade et de la méthode B dont nous avons besoin pour établir la traduction. Cette partie définit les schémas de traduction utilisés pour réaliser le traducteur.

... Reprendre les schémas de traduction du rapport scade to B.

### 4.1 Specification

Signature des noeuds  $\rightarrow$  spécification en B Reprise des informations de typage Reprise du contrat et traduction en PRE .. THEN .. END

### 4.2 Implémentation

Ajout des variables d'états + initialisation (pour les registres) Ajout des variables locales tri topologique pour la séquence de substitution

#### 4.2.1 Traduction des expressions

Les équations sont traduites différemment selon le "type" d'expression qu'elles contiennent. Concernant la partie droite, il y a 4 types d'expressions de Scade à traduire en B :

- Les expressions à manipulant les variables, constantes et opérateurs de base (arithmétiques, relationnels, booléens,...).
- Les appels de noeuds, sous réserve que le noeud appelé a déjà été traduit.
- L'alternative.
- Le registre.

**Opérateurs de base** Les opérateurs de base sont traduits par une substitution *devient égal*, on effectue une simple affectation. Les opérateurs de base de Scade sont identiques à ceux du langage B. Le membre gauche de l'équation correspond à une unique variable, les opérations étant atomiques dans Scade.

$a = op(b_1, \dots, b_n)$  est traduit en B par  $a := op(b_1, \dots, b_n)$ .

**Appel de noeud** Un appel de noeud est traduit par une substitution *appel d'opération*. le membre gauche de l'équation contient autant de variables qu'il y a de sorties pour le noeud appelé. Le noeud appelé doit avoir été traduit auparavant, et la machine B correspondante doit être représentée dans la clause *IMPORT*.  $(a_1, \dots, a_n) = n(b_1, \dots, b_m)$  est traduit par  $(a_1, \dots, a_n) \leftarrow n(b_1, \dots, b_m)$

**Alternative** On traduit l'alternative par la substitution *conditionnelle*. On utilise également la substitution *devient égal* pour chaque branche de l'alternative.  $a = \text{if } \text{cond} \text{ then } b1 \text{ else } b2$  est traduit par *IF cond THEN a := b1 ELSE a := b2*

**Registre** Le registre est également traduit en substitution *devient égal*, cependant les substitutions correspondantes doivent être placées après les autres. Ces équations correspondent à la mise à jour de l'état d'une variable, la mise à jour est donc faite à la fin de l'opération. La valeur initiale du registre doit être indiquée dans la clause *INITIALISATION* de la machine et la variable d'état correspondant au registre doit être déclarée dans *CONCRETE\_VARIABLE*. De plus il faut indiquer dans la clause *INVARIANT* les contraintes de typage de la variable d'état.

$a = \text{ini} \rightarrow (\text{pre } b)$  est donc traduit par  $a := b$  et est placée après les autres substitutions.

**Tri topologique** Dans Scade, l'ordre des équations n'a pas d'importance, mais en B elles doivent s'exécuter en séquence. Les équations correspondants aux registres sont automatiquement placées à la fin, car elles mettent à jour l'état de la machine après son exécution. Il faut donc effectuer un tri topologique des 3 autres types d'équations.

On utilise alors une fonction de tri prenant en entrées :

- la liste des équations du programme (sans les équations de registre)
- la liste des variables d'entrée du programme.

La fonction retourne une liste d'équations triées selon l'ordre topologique.

Fonction Tri (eqs: liste d'équations, vars\_in: liste de variables)

```

eq_non_triees : liste d'équations
eq_admis : liste d'équations
v_admis : liste de variables
eq : equation

BEGIN
  eq_non_triees <- eqs;
  v_admis <- vars_in;
  TANT QUE (eq_non_triees ≠ ∅ )
    eq <- tete(eq_non_triees);
    SI vars_droite(eq) ⊂ v_admis ALORS
      ajout_fin(eq_admis, eq);
      ajout_fin(v_admis, vars_gauche(eq))
    SINON
      ajout_fin(eq_non_triees, eq)
    FIN SI
  FIN TANT QUE
  RETOURNE eq_admis;
END

```

On utilise 4 procédures externe nécessaire à cet algorithme :

- `tete(l)` : retourne le premier élément de la liste en le supprimant
- `ajout_fin(l,e)` : ajoute `e` à la fin de la liste `l`
- `vars_droite(e)` : liste des variables contenues dans la partie droite de l'équation `e`
- `vars_gauche(e)` : liste des variables contenues dans la partie gauche de l'équation `e`

la préocédure `tete` retourne le premier élément de la liste donnée en argument, et le supprime. La procédure Pour commencer, les variables d'entrées sont considérées comme admises. Les premières équations sont celles dont la partie droite ne dépend que des variables admises. La partie gauche des premières équations est ajoutée à la liste des variables admises, et on ajoute les équations dont la partie droite dépend du nouveau set de variables admises. La fonction retourne la liste d' équations triée.

### 4.3 Le traducteur

Développement du traducteur en Ocaml (pourquoi, avantages inconvénients) Comment a-t-il été développé Parties sensibles...

## Chapitre 5

# Exemples tests

Faire 2/3 exemples utilisant tous les traits utilisables de scade.  
Montrer le tri topologique,  
Utiliser les tableaux  
Cas critiques ?  
A TROUVER



## Chapitre 6

# Preuve de correction de la traduction

## Chapitre 7

# Conclusion

Difficultés, Apport projet, Après projet, ...