# Rapport de Stage
-
# Traduction de composants Scade/Lustre vers des machines B

Florian Thibord

30 août 2013

# Table des matières

# Chapitre 1

# Introduction

Ce stage s'est déroulé au sein du projet ANR-10-SEGI-017 CERCLES$^2$ [2]. 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 embarquées critiques est Scade, un acronyme pour Safety Critical Application Developpement Environment. Cet environnement de développement est basé sur la programmation graphique, par schémas-blocs, permettant de définir des programmes faciles à lire et d'engendrer 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, les composants sont écrits soit avec Scade, soit avec un environnement de développement similaire, Simulink. Cependant, il est possible d'importer des composants Simulink dans l'environnement Scade.

Pour assurer que ces composants et leur réutilisation sont sûrs, 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 indique des conditions sur ses entrées (pré-conditions) et sur ses sorties (post-conditions). Ils formeront ainsi une spécification du composant. A la fin des années 60, C.A.R Hoare donne la définition suivante [5] : 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"*

Cette définition donnera une première intuition qui sera reprise par Bertrand Meyer lorsqu'il introduira la programmation par contrat avec le language Eiffel en 1985.

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 associées aux programmes, comme celle basée sur des règles de typage, 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, 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 [1]. 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* correspondants aux instructions du composant. Le raffinement de plus bas niveau, exécutable, est appelé *implantation*. Il peut y avoir des raffinements intermédiaires, mais dans le cadre du projet 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.

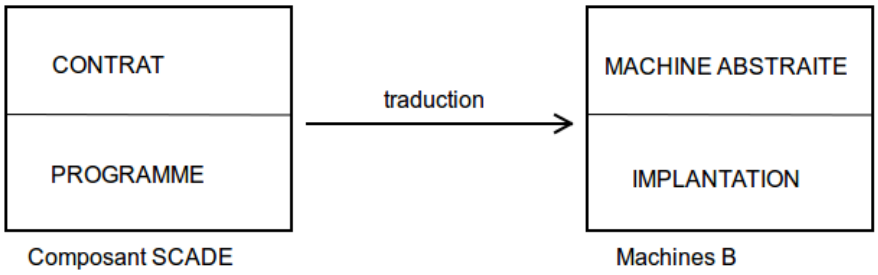


FIGURE 1.1 – Schéma du principe de traduction

# Chapitre 2

# Scade

Scade a été développé par le laboratoire Verimag à partir des travaux sur le langage synchrone Lustre, puis repris par la société Esterel Technologies [7]. 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$^2$ 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

La version de ce programme correspond au noeud `add` suivant :

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

FIGURE 2.2 – Version tectuelle

Au niveau des types de données utilisées, on pourra manipuler des entiers, réels et booléens. 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 conditionelles sont également possibles (`if .. then .. else ..`). Sont également disponibles les opérations sur les tableaux, telles que la définition, l'index, et la concaténation.
Enfin, on peut évidemment faire des appels à d'autres noeuds, pour mettre en pratique la notion de composant réutilisable.

## 2.2 Le temps avec Scade

Le temps est un élément primordial dans ces systèmes dits "réactifs", où l'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 au même instant.

**Une horloge unique** Avec les langages synchrones, on peut synchroniser des instructions sur des horloges différentes. On utilise des opérateurs spécifiques au temps pour synchroniser une instruction sur une horloge spécifique. Pour assurer la bonne définition des noeuds dont les instructions sont calculées sur des horloges différentes, il existe

A COMPLETER AVEC ARTICLE POUZET

Cependant, dans le cadre de ce projet, nous n'utiliserons qu'une seule horloge, celle de base. Toutes les équations sont résolues au même instant. Le seul opérateur temporel utilisable est l'opérateur `fby`.

**L'opérateur fby**

- 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.

Cette construction correspond au bloc Simulink 1/Z, 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.

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

## 2.3 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 A: expr`, où `A` correspond à l'identifiant de la condition, et `expr` un prédicat portant sur une entrée du noeud.
- `guarrantee G: expr`, où `G` est l'identifiant de la condition, et `expr` un prédicat portant sur une sortie du noeud.

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_1 : A <= 100 & A >= 0;
    assume A_2 : B <= 100 & B >= 0;
    guarantee G_1 : 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 :

- un contrat : 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 aux équations du 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 vérificateur de types. 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.
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 généralisées*.
Une machine est précédée d'un en-tête qui diffère selon la machine abstraite et l'implantation :

- pour la machine abstraite, 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"`.

### 3.1.2 Clauses

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. Lors d'un appel d'opération d'une machine importée, l'opération est instanciée.

**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. Ce sont des machines vues, car il n'y a aucune instanciation d'opération, on a seulement besoin de voir les constantes et leurs valeurs.

**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 *Variable Locale*.

**Invariant** Nous pouvons alors établir des invariants sur les registres déclarés dans la clause précédente dans cette clause. Les invariants indiquent le type et la restriction sur l' intervalle sur lequel les registres seront manipulés. Ils seront écrits sous forme de prédicats.

**Initialisation** L'initialisation permet d'indiquer la valeur donnée aux registres lors de l'initialisation du composant, ce sont des substitutions. 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 connecteurs propositionnels : conjonction $\vee$, disjonction $\wedge$, négation $\neg$, 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.

## 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 la moitié 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 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 || . Cette substitution n'est disponible que pour la machine abstraite.

**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 Devient Element De** Les conditions sur les entrées et sorties du programmes sont souvent des restrictions sur des ensembles de valeurs. Cette substitution permet d'attribuer à une variable, une valeur tirée dans un ensemble. Elle se note : Soit E un ensemble et X une variable,

[X :∈ E]

Cette substitution n'est pas une substitution d'implantation.

**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 ... 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 sera valide. Elle se note : Soit P un prédicat et S une substitution,

[PRE P THEN S END]

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 garantit.

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

[R← 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 d'une machine est une reformulation en une expression plus concrète et enrichie. La relation de raffinement est transitive : les valeurs calculées par l'implantation sont conformes à celles attendues par la machine abstraite.

L'implantation correspond 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 (KCG 6) est qualifié pour produire du code certifié pour la norme DO178b.

Ainsi, le raffinement permet de concrétiser un programme jusqu'à obtenir un code exécutable, mais il permet surtout de générer un certain nombre de preuves à démontrer pour prouver que la reformulation de la spécification est valide. La génération des hypothèses à démontrer est automatique dans l'Atelier B, grâce à la transformation des prédicats par les substitutions.

La machine abstraite reprendra uniquement les éléments du contrat du composant, c'est à dire les conditions indiquées sur les entrées et sorties du noeud Scade. Ce sont ces conditions qui devront être vérifiée par les différents raffinements de la machine abstraite. Nous n'utiliserons qu'une étape de raffinement : l'implantation. Il faut donc prouver que cette machine raffinée conserve les propriétés invariantes de la machine abstraite.

La forme générale d'une machine abstraite et de son raffinement est le suivant :

```
MACHINE M

OPERATION

outs ← op(ins) =
  PRE
    P
  THEN
    S
  END

END
```

```
IMPLEMENTATION M_i
REFINES M

IMPORTS M_imp
SEES M_see

CONCRETE_VARIABLES regs
INVARIANT
  Inv
INITIALISATION
  Ini

OPERATIONS

outs ← op(ins) =
  S'

END
```

### 3.4.2 Obligations de preuves

Pour chaque machine, depuis la spécification à l'implémentation, il faut passer une étape de type checking et d'obligation de preuve. L'étape de type checking vérifie la cohérence des types des données manipulées ainsi que la syntaxe du programme. Une fois cette étape validée, on peut générer les obligations de preuves. Le principe des obligations de preuves consiste à prouver une formule à partir d'une liste d'hypothèses :
Hypothèses (liste de prédicats)
⇒
But (doit être prouvé sous ces hypothèses)
On ne montrera pas la construction des différentes obligations de preuves, ces dernières ont été définies dans le B-Book.

Pour l'implantation, les obligations seront générées pour la clause opération, mais aussi pour la clause initialisation.

**Initialisation de l'implantation** Pour l'initialisation, la preuve dépend également des machines présentes dans les clauses IMPORTS et SEES. L'obligation de preuve générée est la suivante :

$\text{Inv}_{imp} \wedge \text{Inv}_{see} \Rightarrow [\text{Ini}_{imp} ; \text{Ini}]\ \text{Inv}$

Avec $\text{Inv}_{imp}$ et $\text{Inv}_{see}$ les invariants des machines importées et vues, et $\text{Ini}_{imp}$ les initialisations des machines importées.
...EXPLICATION NECESSAIRE

**Opération de l'implantation** L'opération de l'implantation dépend également des machines importées et vues, mais aussi et surtout de la machine qu'elle raffine. L'obligation générée est la suivante :

$\text{Inv}_{imp} \wedge \text{Inv}_{see} \wedge \text{Inv} \wedge \text{P} \Rightarrow [\text{S'}]\neg[\text{S}]\neg(\text{Inv})$

...EXPLICATION NECESSAIRE

# 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 avions besoin pour établir la traduction. Cette partie définit les schémas de traduction utilisés pour réaliser le traducteur.

## 4.1 Specification

La machine abstraite est engendrée à partir du contrat du composant. Ce sont donc les conditions posées par les instructions *assume* et *guarantee* qui nous intéressent. Le nom de l'opération sera le même que le nom de la définition Scade. Le nom de la machine reprend également le nom de la définition Scade, cependant le nom de l'opération doit être différent du nom de la machine, donc la première lettre sera une majuscule pour marquer la différence de nom. Ainsi, les défintions Scade doivent être écrites en minuscules.

### 4.1.1 Traduction de la déclaration du noeud

La déclaration d'un noeud Scade comporte le nom du composant, ses entrées/sorties, et le types de ses entrées/sorties. En B, on reprend ces informations sur le nom du noeud et le nom des entrées sorties pour déclarer une opération. Ainsi la déclaration Scade :

```
node mon_noeud (in_1: type_in_1, ..., in_n: type_in_n)
                returns (out_1: type_out_1, ..., out_m: type_out_m);
```

devient l'operation B :

```
in_1, ..., in_n ← mon_noeud(out_1, ..., out_m) =
```

Les informations de types sur les entrées et sorties sont reprises pour les conditions.

### 4.1.2 Traduction des conditions

La machine abstraite de B forme une spécification de la machine implanté, l'opération est ainsi ordinairement consitué d'une substitution précondition PRE P THEN S END. P étant le prédicat correspondant aux conditions des *assumes* et aux informations de typage sur les entrées, tandis que S est la substitution qui reprend les conditions sur les *guarantees* et les informations de typage sur les sorties.

**traduction des préconditions** Les instructions *assumes* sont des formules logiques, généralement des restrictions sur des intervalles. Les opérateurs logiques utilisés sont les mêmes pour Scade que pour le langage B, la traduction est donc directe. Les conditions sur les différentes entrées sont combinées par un opérateur ET logique (&). La condition est précédée par le type de la variable, reprit depuis la déclaration Scade. Dans le cas où une variable d'entrée ou de sortie n'est pas conditionnée, comme c'est souvent le cas pour les variables booléennes, alors on indique seulement le type de la variable. Pour chaque variable le shéma de traduction est le suivant :

`assume in : formule_logique_sur_in`

devient le prédicat B :

`in ∈ type_in & formule_logique_sur_in`

**traduction des postconditions** Les instructions *guarantee* sont également des formules logiques. Cependant, on utilise des substitution *devient element de* pour les post-conditions. Les variables en sortie prennent une valeur comprise dans un certain ensemble donné dans la formule logique. Les substitutions sont regroupée dans une substitution parallèle, elles sont séparées par un ||. Le schéma de traduction est le suivant :

`guarantee out : formule_logique_sur_out`

devient la substitution B :

`out ∈: { iii | iii ∈ type_out & formule_logique_sur_out}`

Cette substitution se lit *out* devient l'élément *iii* de l'ensemble de *type_ out* respectant *formule_ logique_ sur_ out.*

**Traduction des types** La traduction des types de base est directe :
- int est traduit par INT
- real est traduit par REAL
- bool est traduit par BOOL

### 4.1.3 Le cas des tableaux

La traduction des conditions pour les tableaux est moins directe, car il n'y a pas de type primitifs pour les tableaux en B. On utilise des fonctions à la place.

**Traduction des types** Les tableaux peuvent être multi-dimensionnels, mais ne peuvent contenir qu'un seul type de donnée. On peut voir les tableaux comme des fonctions prennant comme argument l'indice de la donnée stockée, et retournant la valeur de cette donnée. Les tableaux sont indexés par des entiers, selectionnés dans les intervalles allant de 1 à la taille du tableau. Par exemple, pour une matrice *Mat* de n lignes et m colonnes, les valeurs des données sont accessibles ainsi : $Mat(p, q)$, avec $1 \leq p \leq n$ et $1 \leq q \leq m$. Ainsi, le schéma correspondant à la traduction de la déclaration d'un tableau est :

`nom_tableau : type_tableau ^ dim_1 ^ ... ^ dim_n`

devient la substitution B :

`nom_tableau : (1..dim_1, ..., 1..dim_n) → type_tableau`

Dans la traduction B, la notation $1..dim_1$ correspond à un intervalle allant de 1 à la valeur de $\dim_1$

**Traduction des formules logiques** Les conditions sur les tableaux en B ont été évoquées dans la section sur les quantificateurs en B. Les conditions portent sur l'ensemble des données contenues dans le tableau. La condition est alors de la forme : pour toute valeur iii correspondant à un index du tableau, la donnée référencée par cet index respecte la condition donnée. Ainsi, une formule logique f_l portant sur un tableau T de n dimensions correspond à la formule B :

$\forall$ iii. (iii : (1..dim_1, ..., 1..dim_n) $\rightarrow$ f_l)

### 4.1.4 schéma général

Le shéma de traduction d'un composant Scade foo en une machine abstraite B est le suivant :

```
node foo
  (in_1: in_1_type, ..., in_p: in_p_type)
  returns
  (out_1: out_1_type, ..., out_q: out_q_type);
var
  v1 : v1_type;
  ...
  vn : vn_type;
  r1 : r1_type;
  ...
  rn : rn_type;
let
  assume in_1 : pred_in_1;
  ...
  assume in_p : pred_in_p;

   liste d'equations

  guarantee out_1 : pred_out_1;
  ...
  guarantee out_q : pred_out_q;
tel;
```

```
MACHINE Foo

OPERATION

out_1, ..., out_q ← foo(in_1, ..., in_p) =
  PRE
    in_1 ∈ in_1_type & pred_in_1 &
    ... &
    in_p ∈ in_p_type & pred_in_p &
  THEN
    out_1 ∈: { iii | iii ∈ out_1_type & pred_out_1}
    ...
    out_q ∈: { iii | iii ∈ out_q_type & pred_out_q}
  END
END
```

*note : les formules booléennes sont notées pred_ nom où nom correspond au nom de la variable concernée par cette formule, qui à la même syntaxe en Scade et en B*

## 4.2 Implémentation

### 4.2.1 Traduction des équations

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.

$$a = op(b_1, ..., b_n) \xrightarrow{traduction\ equations} a := op(b_1, ..., b_n).$$

**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 êtreprésente dans la clause *IMPORT*.

$$(a_1, ...a_n) = n(b_1, ..., b_m) \xrightarrow{traduction\ equations} (a_1, ...a_n) \leftarrow n(b_1, ..., b_n)$$

**Alternative** On traduit l'alternative par la substitution *conditionnelle*. On utilise également la substitution *devient égal* pour chaque branche de l'alternative.

$$a = if\ cond\ then\ b1\ else\ b2 \xrightarrow{traduction\ equations} IF\ cond\ THEN\ a := b1\ ELSE\ a := b2$$

**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.

$$a = ini \rightarrow (pre\ b) \xrightarrow{traduction\ equations} a := b$$

**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 automatiquements 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

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

```
Fonction Tri (eqs: liste d'equations, vars_in: liste de variables)

eq_non_triees : liste d'equations
eq_admis : liste d'equations
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 l et supprime l'élément en question de l
- 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.

### 4.2.2 Retour sur la traduction des registres

Dans Scade, les équations correspondant aux registres sont initialisées à une certaine valeur, puis ils prennent la valeur d'une autre variable pour les instants suivants. Avec B, les registres sont des variables d'état, qui auront lors de l'initialisation du programme la valeur définie pour le premier instant avec Scade. Cette initialisation de registre doit se faire dans la clause INITIALISATION, et le nom du registre doit auparavant être déclaré dans la clause CONCRETE_VARIABLES. L'information de type du registre doit alors être déclarée comme prédicat dans la clause INVARIANT. De plus, si le registre porte sur une variable d'entrée ou de sortie, on peut alors récupérer la condition (si elle existe) sur l'entrée ou la sortie en question pour compléter le prédicat de la clause INVARIANT.

Ainsi, soit un registre reg de type t, ayant l'équation suivante :
$reg = ini \rightarrow (pre\ a)$
avec a une variable d'entrée ou de sortie du composant, possédant une précondition ou postcondition P, et ini une valeur d'initialisation de reg. Il faudra alors écrire dans l'implantation correspondante :

```
IMPLEMENTATION ...
...

CONCRETE_VARIABLES ..., reg
INVARIANT
```

```
  ...& reg : t & P
INITIALISATION
  ... ; rn := ini;

OPERATION
... =
VAR ... IN
  ...;
  reg := a
END
```

### 4.2.3 Gestion des clauses SEES et IMPORTS

Pour inclure des opérations ou des constantes définies dans des machines externes, il faut les ajouter respectivement dans les clauses IMPORTS et SEES de la machine courante. Cependant, il n'y a pas de processus automatique pour ajouter les machines nécessaires dans ces clauses. Il faut donc les ajouter manuellement une fois la traduction est réalisée.

### 4.2.4 schéma général

Le shéma de traduction d'un composant Scade foo en une implémentation B est le suivant :

```
node foo
  (in_1: in_1_type, ..., in_p: in_p_type)
  returns
  (out_1: out_1_type, ..., out_q: out_q_type);
var
  v1 : v1_type;
  ...
  vn : vn_type;
  r1 : r1_type;
  ...
  rn : rn_type;
let
  assume in_1 : pred_in_1;
  ...
  assume in_p : pred_in_p;

  liste d'equations

  guarantee out_1 : pred_out_1;
  ...
  guarantee out_q : pred_out_q;
tel;
```

```
IMPLEMENTATION Foo_i
REFINES Foo

IMPORTS M_imp
SEES M_see

CONCRETE_VARIABLES r1, ..., rn
INVARIANT
  r1 : r1_type &
  ... &
  rn : rn_type
INITIALISATION
  r1 := ;
  ... ;
  rn := ;

OPERATION

out_1, ..., out_q ← foo(in_1, ..., in_p) =

VAR v1, ..., vn IN

  sequence de substitutions

END
```

Les clauses invariant, initialisation et la séquence de substitutions sont obtenues en appliquant la traduction des équations sur la liste d'équations du composant Scade.

## 4.3 Le traducteur

Le traducteur a été écrit en OCaml, qui est un langage très efficace pour développer des compilateurs, et donc des traducteurs.
Le parseur/lexeur a été écrit à partir de la grammaire de Scade, définie dans le manuel Textual Scade.

Les programmes parsés sont alors représentés sous forme d'arbre de syntaxe abstraite, donné en annexe A. Cette représentation permet une manipulation sur les différents éléments du programme, telle que la liste d'équation sur laquelle est effectuée l'algorithme du tri topologique.
On identifie également les différentes équations que l'on répertorie en opération de base, registres, appel de noeud, et alternative.

Cet arbre est ensuite transformé en un arbre donné en annexe B, qui peut être imprimé dans deux fichiers de sortie, correspondant à la machine abstraite et à l'implantation correspondant au composant donné en entrée. L'impression respecte la grammaire donnée dans le Manuel de référence de B, et le couple de fichier peut être importé dans un projet de l'Atelier B afin de vérifier le typage et la syntaxe de chaque machine, et de passer les étapes d'obligation de preuve de façon automatique.

# Bibliographie


[1] Jean-Raymond Abrial. *The B-Book : Assigning Programs to Meanings.* Cambridge University Press, Cambridge, 1996.

[2] Cercles$^2$. La certification compositionnelle des logiciels embarqués critiques et sûrs.

[3] Clearsy. Atelier b. www.atelierb.eu.

[4] Clearsy. *B Language Reference Manual*, 2012.

[5] C. A. R. Hoare. An axiomatic basis for computer programming (reprint). *Commun. ACM*, 26(1) :53–56, 1983.

[6] Marie-Laure Potet. *Spécifications et développements formels : Etude des aspects compositionnels dans la méthode B.* Habilitation à diriger les recherches, INPG, Grenoble, France, 2002.

[7] Esterel Technologies. www.esterel-technologies.com.

[8] Esterel Technologies. *Scade Language Reference Manual*, 2012.

[9] J.B Wordsworth. *Software Engineering with B.* Addison-Wesley, England, 1996.