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

Florian Thibord

4 septembre 2013

Table des matières

Introduction

Ce stage s'est déroulé au sein du projet ANR-10-SEGI-017 CERCLES[2] [3]. L'objectif du projet est de certifier formellement des composants réutilisables, afin 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 [6] : 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 un couple de machines B.
Un composant Scade est constitué d'un *noeud* correspondant à un programme, et d'un ensemble de conditions sur les entrées et sorties du programme qui vont former le contrat. Le traducteur suit une ligne de compilation classique, prenant en entrée le programme et son contrat, et produit en sortie une machine abstraite correspondant au contrat, ainsi qu'une machine raffinée qui implante le programme.

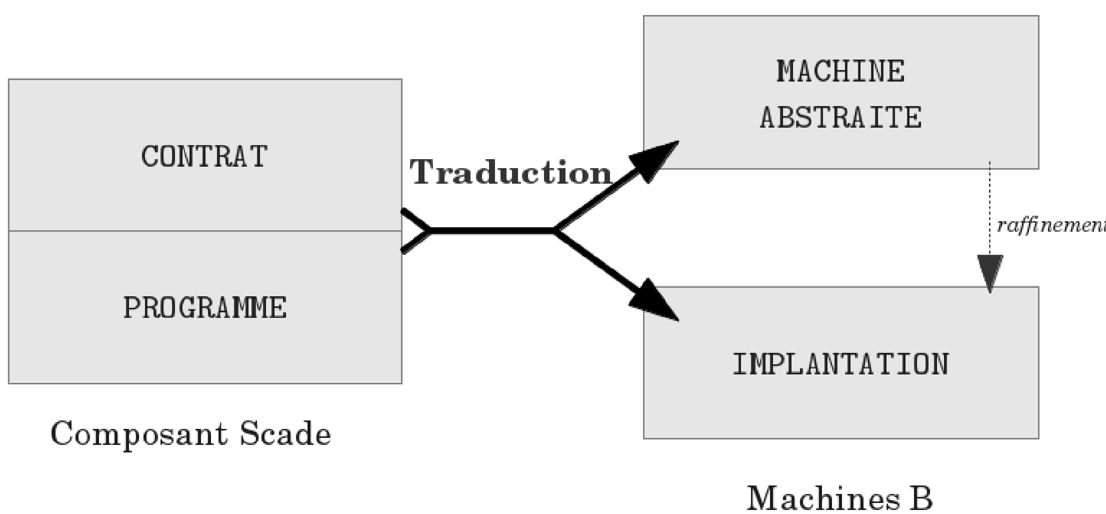

Figure 1 – Schéma du principe de traduction

Chapitre 1

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 [8]. 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.

1.1 Architecture d'un composant Scade

Scade étant un environnement de programmation par schémas-blocs, on développe avec des "boîtes". Par exemple, un programme prenant en entrées 3 entiers `b_sup`, `b_inf` et `z`, qui retourne un entier v égal à :

- `z`, si `z` est compris entre `b_inf` et `b_sup`
- `b_inf`, si `z` est inférieur à `b_inf`
- `b_sup`, si `z` est supérieur à `b_sup`

Ce programme s'écrira de la façon suivante dans Scade.

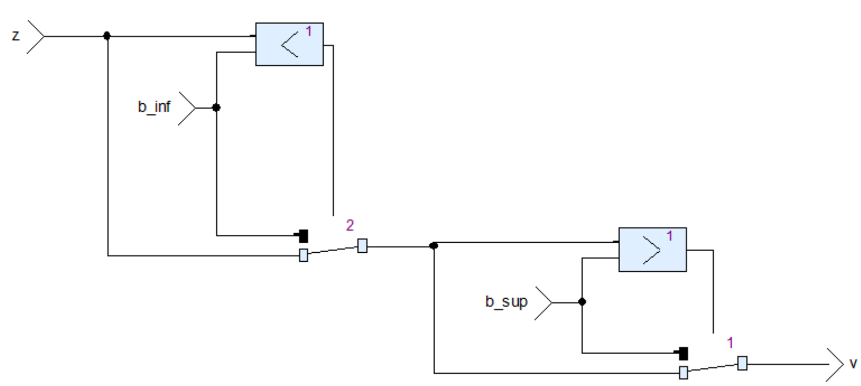

FIGURE 1.1 – Version graphique du noeud bound

La version textuelle de ce programme correspond au noeud `bound` suivant :

```
node bound (z: int; b_inf: int; b_sup: int) returns (v: int);
var
  a: int;
  c1: int;
  c2: int;
let
  a = if c1 then b_inf else z;
  v = if c2 then b_sup else a;
  c1 = z < b_inf;
  c2 = a > b_sup;
tel
```

FIGURE 1.2 – 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`). Sont également disponibles les opérations sur les tableaux, telles que la définition, l'index, et la concaténation. Lors de la génération textuelle du noeud, Scade *atomise* l'ensemble des opérations et introduit des variables locales qui correspondent aux fils du noeud. Pour ces opérations, la forme des équations sera donc toujours semblable, on aura une seule variable à gauche de l'équation, et à droite, on aura un opérateur appliqué à un ensemble de variables :

$\texttt{v = op}_{base}\texttt{(x}_1\texttt{, ..., x}_n\texttt{);}$

Les expressions conditionelles sont également possibles :

$\texttt{v = if c then x}_1\texttt{ else x}_2\texttt{;}$

avec `c`, $\texttt{x}_1$ et $\texttt{x}_2$ des variables.

Enfin, on peut évidemment faire des appels à d'autres noeuds, pour mettre en pratique la notion de composant réutilisable. Les équations correspondantes peuvent avoir plusieurs variables dans la partie gauche de l'équation, car un noeud appelé peut avoir plusieurs sorties.

$\texttt{v}_1\texttt{, ..., v}_p\texttt{ = op}_{appel}\texttt{(x}_1\texttt{, ..., x}_n\texttt{);}$

1.2 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 une étape de calcul des horloges [2]. 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.

L'opérateur fby Il y a un seul opérateur temporel qui reste utilisable, c'est l'opérateur `fby`. Cet opérateur prend 3 arguments :
- une variable v
- un délai
- une initialisation

A l'instant i[1], `fby` retourne la valeur de la variable v à l'instant (i - délai). Dans le cas ou (i - délai) est négatif, l'opérateur retourne la valeur initialisation. Par exemple, on représente dans le tableau suivant la valeur de sortie de l'opérateur `fby` en fonction de la valeur d'une variable d'entrée `v`, avec une initilisation à 0, et un délai fixé à 1.

`instant`	0	1	2	...
`v`	10	20	30	...
`z`	0	10	20	...

FIGURE 1.3 – `z = fby(v, 1, 0)`

Cet opérateur permet de donner un *état* à un composant, les calculs des équations étant alors dépendants de l'instant où ils sont effectués. Par la suite, on appellera cette construction un *registre*.

1.3 Contrats

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 : une précondition.
- `guarrantee G: expr`, où `G` est l'identifiant de la condition, et `expr` un prédicat portant sur une sortie du noeud : une postcondition.

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). Pour les entiers et les réels, on indiquera des intervals de valeurs.

Par exemple, en reprenant le noeud `bound` précédent, on donne comme condition sur les entrées qu'elles doivent être comprises entre -2000 et 2000 inclus. Si les préconditions sont respectées, alors la sortie sera comprise entre -2000 et 2000 inclus :

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

```
node bound (z: int; b_inf: int; b_sup: int) returns (v: int);
var
  a: int;
  c1: int;
  c2: int;
let
  assume A_1 : b_inf <= 2000 and b_inf >= -2000;
  assume A_2 : b_sup <= 2000 and b_sup >= -2000;
  assume A_3 : z <= 2000 and z >= -2000;
  guarantee G_1 : v <= 2000 and v >= -2000;
  a = if c1 then b_inf else z;
  v = if c2 then b_sup else a;
  c1 = z < b_inf;
  c2 = a > b_sup;
tel
```

Figure 1.4 – Noeud bound avec contrat

Chapitre 2

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.

2.1 Machine B

2.1.1 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".

2.1.2 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.

2.1.3 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 qui composent ces prédicats de base 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 : $P \wedge \neg(Q)$ est également un prédicat.

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

2.2 Expressions

Expressions de base 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 MACHINES, 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 donc 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.

Les tableaux sont donc habituellement modélisé par des fonctions en B, mais on aurait également pu utiliser le type *séquence*. Ce dernier correspond au type liste que l'on retrouve dans des langages comme OCaml ou Scheme. Cependant, reproduire les opérations des tableaux avec les séquence n'est pas aussi intuitive qu'avec les fonctions.

Par exemple, la sélection d'un élément présent à l'indice i d'une séquence T se fait à l'aide de deux opérateurs :
- $T \downarrow i$: retourne la séquence T dont les i premiers éléments on été supprimés.
- $first(T)$: retourne le premier élément de la séquence T

La sélection s'écrit alors : $first(T \downarrow i)$. Tandis qu'avec les fonctions, la sélection s'écrit T(i) pour une fonction T modélisant un tableau dans lequel on veux obtenir l'élément présent à l'indice i.

Ensembles en compréhension La définition d'ensemble en compréhension fait également partie des expressions de B. On l'utilise notamment dans l'écriture des postconditions. A partir de la condition sur la sortie, on obtient un ensemble qui respecte cette propriété. A partir d'un ensemble E, et d'une condition C portant sur les éléments de l'ensemble, la définition d'ensemble en compréhension s'écrit $\{iii \in E|C\}$.

2.3 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 Devient égal Cette substitution réalise l'affectation, elle remplace une variable par une expression. Notation : 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 Appel operation Notation : 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.

Substitution Condition C'est cette substitution que l'on utilise pour exprimer le choix entre deux substitutions. Notation : 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. Notation : 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, correpondant à une substitution bloc.

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

On retrouve ainsi les constructions d'un langage de programmation impératif, avec des affectations, appels d'opération, l'alternative, la définition de variables locales et la séquence d'instructions.
Les substitutions suivantes ne pas utilisables dans une implantation. Elles permettent d'exprimer des notions abstraites, et n'ont pas de correspondance dans les langages de programmations.

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 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. C'est une substitution inderterminée. Notation : Soit E un ensemble et X une variable,

[X :∈ E]

Substitution Precondition Cette substitution fixe les préconditions sous lesquelles une substitution sera valide. Notation : 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.

2.4 Raffinements

2.4.1 Principes du raffinement

Le raffinement d'une machine est une reformulation en une expression plus concrète et enrichie. La relation de raffinement est transitive : si le raffinement est correct, 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 semblable 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 propriétés 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.

Dans le cadre du projet, nous n'avons qu'une étape de raffinement, et 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
```

2.4.2 Obligations de preuves

Pour chaque machine, de la spécification à l'implantation, il faut passer trois étapes de vérification : au niveau syntaxique, le typage, et les obligations de preuves. Les obligations de preuves ont été définies dans le B-Book [1].

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

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

Avec Inv_{imp} et Inv_{see} les invariants des machines importées et vues, et Ini_{imp} les initialisations des machines importées.

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 :

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

Prenons un exemple. Soit la machine Integr, qui utilise la machine Bound, générée à partir du noeud bound. Le couple de machine importées est le suivant :

```
MACHINE Integr

OPERATION

yy ← integr(xx) =
  PRE
    xx ∈ INT & -256 <= xx & xx <= 255
  THEN
    yy ∈:  yy | yy ∈ INT &
          -1024 <= yy & yy <= 1023
  END
```

```
IMPLEMENTATION Integr_i
REFINES Integr

IMPORTS Bound

CONCRETE_VARIABLES reg1
INVARIANT
  reg1 ∈  INT & -1024 <= reg1 & reg1 <= 1023
INITIALISATION
  reg1 := 0

OPERATIONS

yy ← integr(xx) =
VAR zz IN
   zz := xx + reg1;
   yy ← bound(-1024, zz, 1023);
   reg1 := yy
END
```

La machine `Bound` n'a pas d'invariant ni d'initialisation, on ne s'intéresse donc qu'à l'invariant, l'initialisation et les substitutions de la machine `Integr`.
Pour l'initialisation, on obtient l'obligation suivante :

$\Rightarrow$ [reg1 :=0] reg1 $\in$ INT & -1024 <= reg1 & reg1 <= 1023

En appliquant la substitution à l'invariant on obtient :

$\Rightarrow$ 0 $\in$ INT & -1024 <= 0 & 0 <= 1023

ce qui est correct.

Pour l'opération on a :

(reg1 $\in$ INT & -1024 <= reg1 & reg1 <= 1023) $\wedge$
(xx $\in$ INT & -256 <= xx & xx <= 255)
$\Rightarrow$
[zz := xx + reg1 ;
  yy $\leftarrow$ bound(-1024, zz, 1023) ;
  reg1 := yy]
$\neg$ [yy $\in$: yy | yy $\in$ INT & -1024 <= yy & yy <= 1023]
$\neg$ (reg1 $\in$ INT & -1024 <= reg1 & reg1 <= 1023)

Chapitre 3

Schémas de traduction

Dans les deux parties précédentes, nous avons posé 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.

3.1 Spécification

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.

3.1.1 Traduction de la déclaration du noeud

La déclaration d'un noeud Scade comporte le nom du composant, ses entrées/sorties, et le type 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) =
```

En reprenant le noeud `bound`, la traduction donne :
`vv ← bound(zz, b_inf, b_sup) =`
On peut noter que le nom de la variable de sortie a été modifié par rapport à la version de Scade. Dans B, les noms de variables n'ayant qu'une lettre sont réservés, donc on effectue un rennomage sur l'ensemble des variables du programmes : les lettres simples sont doublées, et si le nouveau nom est déjà utilisé par une autre variable, on redouble le nom jusqu'à ce qu'il n'y ai aucun conflit dans les noms de variables.

Traduction des types de base Les informations de types sur les entrées et sorties sont reprises pour les préconditions et postconditions. La traduction des types de base est directe :

- int est traduit par INT

- real est traduit par REAL
- bool est traduit par BOOL

3.1.2 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.
En reprenant l'exemple bound, les prédicats générés pour les préconditions sont :

```
zz ∈ INT & zz <= 2000 & zz >= -2000 &
b_sup ∈ INT & b_sup <= 2000 & b_sup >= -2000 &
b_inf ∈ INT & b_inf <= 2000 & b_inf >= -2000
```

Traduction des postconditions Les instructions `guarantee` sont également des formules logiques. Cependant, on utilise des substitution *Devient Element De* pour les post-conditions, que l'on combine avec une définition d'ensemble en compréhension. Les variables en sortie auront une valeur respectant la postcondition qui va être utilisée pour définir l'ensemble en compréhension. Les substitutions sont regroupée dans une substitution parallèle, elles sont séparées par un ||. Les substitutions seront de la forme :

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

avec `out` la variable de sortie, `type_out` son type, et `C` la postcondition associée.
En reprenant l'exemple de `bound` on obtient pour v :
`vv ∈: { iii | iii ∈ INT & iii <= 200 & iii >= -2000 }`

3.1.3 Le cas des tableaux

La traduction des conditions pour les tableaux est moins directe, car il n'y a pas de type primitif 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, sélectionnés 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 :

```
∀ iii. (iii : (1..dim_1, ..., 1..dim_n) → f_l)
```

Prenons par exemple un tableau T de taille 2 comprenant des entiers. La condition associée à T est que ses éléments doivent être compris entre 0 et 10 exclus.
Le type du tableau sera alors :
`T : (1 .. 2) → INT`
Et la formule associée au tableau sera :
`∀iii. (iii ∈ (1 .. 2) => 0 < T(iii) & T(iii) < 10)`

3.1.4 schéma général

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

```
node foo
  (in1: in1_type, ..., inp: inp_type)
  returns
  (out1: out1_type, ..., outq: outq_type);
var

  ...

let
  assume A1 : pred_in1;
  ...
  assume Ap : pred_inp;

   liste d'equations

  guarantee G1 : pred_out1;
  ...
  guarantee Gq : pred_outq;
tel;
```

```
MACHINE Foo

OPERATION

out1, ..., outq ← foo(in1, ..., inp) =
  PRE
    in1 ∈ in1_type & pred_in1 &
    ... &
    inp ∈ inp_type & pred_inp &
  THEN
    out1 ∈: { iii | iii ∈ out1_type & pred_out1}
    ...
    outq ∈: { iii | iii ∈ outq_type & pred_outq}
  END
END
```

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. De plus, les variables locales ne sont pas considérées dans la machine abstraite.

3.2 Implémentation

3.2.1 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 Egal*, 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.

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

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 être présente dans la clause `IMPORT`.

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

Alternative On traduit l'alternative par la substitution *Condition*. On utilise également la substitution *Devient Egal* pour chaque branche de l'alternative.

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

Registres

Le registre est également traduit en substitution *Devient Egal*, 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.

$$a = fby(ini, delai, b) \xrightarrow{traduction\ \ equations} a := b$$

Dans Scade, les équations correspondant aux registres sont initialisées à une certaine valeur, puis ils prennent la valeur d'une autre variable après un certain délai, supérieur à 1. Avec B, on a un langage impératif sans notion de temps, mais dont les machines peuvent avoir un état grâce à des variables d'état déclarées dans la clause `CONCRETE_VARIABLES`. Pour traduire un registre avec un délai égal à 1, il faut donc une variable d'état qui sera initialisée avec la valeur d'initialisation déclarée dans l'opérateur `fby`, et il faudra mettre à jour ce registre à la fin de l'opération.

L'initialisation doit se faire dans la clause `INITIALISATION`, et l'information de type du registre doit être indiquée 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 avec un délai de 1, ayant l'équation suivante :
`reg = fby(ini, 1, 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.
Cependant, la condition P porte sur une variable d'entrée ou de sortie que l'on connait, et on veux qu'elle porte sur le registre, il faut donc effectuer un renommage de la condition P en remplaçant les occurence du nom de la variable concernée par le nom du registre. On obtient une condition P_{reg}.
Pour résumer, on obtiendra dans l'implantation :

```
IMPLEMENTATION ...
...

CONCRETE_VARIABLES ..., reg
INVARIANT
  ...& reg : t & P_reg
INITIALISATION
  ... ; reg := ini

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

Dans le cas ou on fixe le délai de l'opérateur `fby` à 2 ou plus, il faut introduire de nouvelle variables d'état intermédiaires. On ne peut simuler qu'un registre ayant un délai égal à 1, donc pour tout registre ayant un délai supérieur, il faudra introduire (délai - 1) nouvelles variables. Toute les variables intermédiaires auront la même initialisation et le même invariant que la variable d'état "principale". A la fin de l'opération, elle devront cependant être ordonnée correctement, la variable principale doit être affectée en dernière.
Prenons par exemple l'équation suivante avec un délai fixé à 3 : `reg = fby(ini, 3, a)`
La machine générée doit être :

```
IMPLEMENTATION ...
...

CONCRETE_VARIABLES ..., reg, reg_i1, reg_i2
INVARIANT
  ...& reg : t & P_reg &
  reg_i1 : t & P_reg_i1 &
  reg_i2 : t & P_reg_i2
INITIALISATION
  ... ; reg := ini;
  reg_i1 := ini;
  reg_i2 := ini

OPERATION
... =
VAR ... IN
  ...;
  reg_i1 := a;
  reg_i2 := reg_i1;
  reg := reg_i2
END
```

Séquencement des équations

Dans Scade, l'ordre des équations n'a pas d'importance, une analyse de causalité est effectuée lors de la validation du noeud. Comme toutes les équations sont résolues au même instant dans un noeud, cette analyse vérifie qu'un flot ne dépend jamais de lui même au même instant. Par exemple, on ne peut accepter une équation du type : `X = not X`, car au même instant la variable X est vraie et fausse. Ainsi, une variable ne peut être dans la partie droite et gauche d'un équation, sauf si la partie droite est composée d'un opérateur `fby`. Dans ce cas, le flot retourné par l'opérateur correspond à celui d'un instant précédent, donc il n'y a pas de dépendance directe et l'équation est valide.

Cependant, en B les substitutions doivent s'exécuter en séquence. Il faut donc effectuer un séquencement des équations avant de les traduire en substitutions. 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)
- un liste de variables comprenant les variables d'entrée du programme et les registres

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

```
Fonction Tri (eqs: liste d'equations, vars: 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;
 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 externes nécessaires 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`

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 nouvel ensemble de variables admises. La fonction retourne la liste d' équations triée.

3.2.2 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.

3.2.3 Schéma général

Le nom de la machine reprend 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. 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.

3.3 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.

Conclusion

L'utilisation de méthodes formelles pour la validation de programmes n'est pas une activité récente. Cependant, les industriels ayant toujours eu recours aux tests pour valider un programme, il existe une inertie dans ce domaine qui bloque la propagation d'autres approches. Il y a néanmoins un regain d'intérêt pour les méthodes formelles depuis quelques années, la fiabilité des composants formellement validés étant supérieure à celle des composants testés. C'est surtout le cas dans les domaines critiques tel que la santé, les systèmes de transport, ou la production d'énergie, qui recquièrent un niveau d'exigence élevé pour la validation des programmes utilisés.

Scade est un outil très utilisé pour le développement de composants pour les logiciels embarqués. Mais la validation de ces composants passe encore par des tests, car il n'y a aucune validation par méthode formelle integrée à l'outil de développement. L'intérêt de ce projet est de proposer une validation par méthode formelle en utilisant B, ce qui est rendu possible par une traduction automatique des composants Scade vers des machines B. Ainsi, dans l'état actuel, le traducteur permet un gain de temps et de sécurité dans la validation de composants développés avec Scade.

Il reste cependant à formaliser une preuve de correction de la traduction elle même. On pourra réfléchir à une démonstration basée sur la structure des différentes équations traduites.
De plus, ce travail ne s'appuie que sur un fragment de Scade, les possibilités offertes par le langage étant très vastes. Ainsi, à partir de ce projet, il est envisageable de poursuivre différents axes pour élargir le fragment de Scade traduit. On pourrait notamment s'intéresser aux machines à état, qui permettent de décrire des automates. On peut également réfléchir à la possibilité de traduire d'autres opérateurs synchrones en B.

Bibliographie

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

[2] Paul Caspi and Marc Pouzet. Synchronous Kahn Networks. In *ACM SIGPLAN International Conference on Functional Programming*, Philadelphia, Pensylvania, 1996.

[3] Cercles^2. la CERtification Compositionnelle des Logiciels Embarqués critiqueS et Sûrs. www.algo-prog.info/cercles.

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

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

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

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

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

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

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