Adaptabilité et validation de la traduction de B vers C

Points de vue et résultats du projet BOM

Frédéric Badeau[*] — Didier Bert[**] — Sylvain Boulmé[**]
Christophe Métayer[*] — Marie-Laure Potet[**] — Nicolas Stouls[**]
Laurent Voisin[*]

[*] *ClearSy, Europarc de Pichaury - Bât. C2*
1330, avenue G. de la Lauzière
F-13856 Aix-en-Provence cedex 3
{frederic.badeau, christophe.metayer, laurent.voisin}@clearsy.com

[**] *Laboratoire LSR, Institut IMAG*
681, rue de la Passerelle
F-38400 Saint Martin d'Hères
{didier.bert, sylvain.boulme, marie-laure.potet, nicolas.stouls}@imag.fr

RÉSUMÉ. *Cet article présente quelques résultats du projet RNTL BOM (B Optimisant la Mémoire). Le but était d'obtenir un traducteur du langage* B *vers le langage* C*, facilement adaptable à des contraintes mémoire (logiciels embarqués sur cartes à puce). Nous proposons une méthode d'extension des types de base permettant d'inclure certains types du langage cible. Nous présentons comment le traducteur est décrit sous forme de règles de traduction pour permettre d'adapter la traduction aux plates-formes cibles. Nous indiquons une démarche pour assurer la correction du code produit. Nous terminons par quelques mesures de taille de code généré pour la machine virtuelle Java Card.*

ABSTRACT. *This paper presents some results of the RNTL BOM project (B Optimizing Memory). The goal was to design and implement a translator from* B *to* C *easily adaptable to memory constraints (embedded software on smart cards). We propose an extension method of the basic data types able to include some types of the target language. We present how the translator is described by the way of translation rules and how the translation can be adapted according to the target platforms. Then, we show how to ensure that the generated code is correct. Eventually, measures of the size of the generated code for the Java Card Virtual Machine are given.*

MOTS-CLÉS : *méthode B, génération de code, traducteur adaptable, cartes à puce.*

KEYWORDS: *B method, code generation, adaptable translator, smart cards.*

1. Introduction

La méthode B [ABR 96] a été conçue pour écrire des programmes qui satisfont des propriétés de sûreté et préservent l'effet des opérations décrites au plus haut niveau d'abstraction (dans les *machines*). Cette garantie est obtenue par le fait que la théorie sous-jacente définit les *obligations de preuves* (lemmes) qui doivent être démontrées pour assurer cette cohérence. Les outils, tel l'AtelierB de ClearSy [CLE 02a], permettent de générer puis de démontrer ces lemmes, à l'aide du démonstrateur automatique ou interactif. Cependant, l'assurance obtenue par la preuve mathématique s'arrête au dernier niveau de développement prévu par la méthode B. Ce niveau est celui des *implémentations*. A ce stade, les instructions des opérations sont déterministes et ne comprennent qu'un sous-ensemble simple des instructions usuelles des langages de programmation. Les valeurs des variables ne peuvent être que des entiers bornés, des littéraux de types énumérés, des booléens ou des tableaux. Ce niveau de langage est appelé le B_0. Il existe donc des traducteurs intégrés dans l'atelierB, qui fournissent une traduction dans les langages de programmation comme C, C++ ou Ada.

Afin de ne pas rompre le développement formel, le processus de traduction doit être le plus sûr possible. De plus, ce processus doit être adaptable. En effet, chaque domaine d'application, ou chaque constructeur, peut avoir ses propres contraintes en termes de code produit. Dans certaines applications pour le ferroviaire par exemple, la technologie utilisée pour assurer la correction à l'exécution est la technique du processeur codé, qui nécessite en entrée une forme particulière du langage Ada [FOR 89, BEH 99]. De la même manière, dans le domaine des applications embarquées s'exécutant sur des matériels de petite surface (carte à puces, Digital Signal Processors, etc.) le code développé doit être adapté au matériel visé. Le processus de traduction doit pouvoir être ajusté en fonction de la cible. Le projet RNTL BOM (B Optimisant la Mémoire)[1] a pour but de développer une approche flexible de mise en œuvre de traducteurs vers le langage C, pour des applications carte à puce qui nécessitent de gérer avec minutie la mémoire. De plus, cette approche doit être validable, en accord avec le processus formel de la méthode B. Ce projet a réuni les partenaires GemPlus, porteur du projet, ClearSy développeur de l'AtelierB, et les laboratoires universitaires LSR-IMAG (Grenoble) et LIFC (Besançon).

L'adaptabilité du processus de traduction peut s'envisager suivant différents angles. La première approche consiste à offrir des possibilités d'étendre le langage B_0 afin de permettre au développeur d'écrire le code B_0 le plus adapté. Cette solution est la plus confortable. En particulier, elle permet de rester dans le cadre d'un développement formel et donc de produire du code validé. Néanmoins les possibilités d'extension doivent rester compatibles avec le développement formel. La seconde approche consiste à avoir un langage B_0 figé (donc parfaitement sûr) et à développer des traducteurs capables de produire du code adapté aux compilateurs des langages cibles. Cette

1. URL : http ://lifc.univ-fcomte.fr/~tatibouet/WEBBOM/. Ce projet a reçu un soutien fi nancier du ministère de l'Industrie d'août 2001 à mars 2003, dans le cadre du RNTL (Réseau National des Technologies du Logiciel).

solution offre une grande souplesse, mais pose des problèmes du point de vue validation. En effet il faut pouvoir valider chaque traducteur et, plus les traductions sont sophistiquées, plus elles sont difficiles à valider. La solution retenue ici est un compromis entre ces deux solutions. Certaines particularités du langage cible sont intégrées dans le niveau B_0. Par exemple, il est possible d'adapter la représentation des entiers pour pouvoir manipuler des entiers courts, non signés, etc. Par contre d'autres particularités sont traitées au niveau de la traduction, par exemple tout ce qui est relatif à la manipulation d'adresses, notion non supportée par la méthode B et par les méthodes formelles en général. La validation du processus de traduction va concerner différents aspects : la cohérence entre la représentation des données en B_0 et celle du langage cible, la correction des règles de traduction et la validation de certaines optimisations, comme le passage de paramètres par référence.

La section 2 décrit les principales caractéristiques du langage B_0 vu comme un langage de programmation sur lequel des preuves formelles peuvent être faites. Dans la section 3, nous discutons des possibilités d'adaptation des types du niveau B_0, en illustrant l'approche choisie pour différents types entiers du langage C. Nous mettons aussi en évidence les conditions à respecter pour préserver la validité de la démarche que nous proposons. Dans la section 4, nous présentons la phase de traduction proprement dite et le principe de validation de cette étape. Enfin, la section 5 compare les tailles de code généré avec le traducteur de l'AtelierB et les traducteurs réalisés dans le projet BOM, à partir d'un cas d'étude commun, la machine virtuelle Java Card, embarquée sur deux plates-formes cartes à puces aux architectures différentes (8 bits et 32 bits).

2. Quelques caractéristiques du B_0

Le langage B_0 peut être vu à la fois comme un sous-ensemble du langage B et comme un langage de programmation. Dans cette section, nous mettons principalement en évidence les particularités liées à l'aspect langage de programmation.

2.1. *Les informations abstraites et concrètes*

Dans le langage B, il y a deux sortes de constantes et de variables (ou entités) : ce sont les constantes et variables *abstraites* et les constantes et variables *concrètes*. Les entités abstraites permettent de décrire les données (ensembles, relations, etc.) à un haut niveau d'abstraction et donc sont principalement utilisées pour la spécification, c'est-à-dire les machines et les premiers raffinements. Les entités concrètes sont celles qui sont utilisées pour la programmation. Elles sont fixées et non raffinables. Elles sont typées par des « types concrets » ou « types B_0». Notons que si des entités abstraites ne peuvent pas se trouver dans les implémentations, des entités concrètes peuvent très bien se trouver dans des machines abstraites, soit parce qu'on déclare d'emblée des variables ou des constantes concrètes, soit parce que certaines parties des machines ne peuvent pas être raffinées et sont donc nécessairement concrètes. C'est le cas des

paramètres et des résultats d'opérations, qui restent inchangés tout au long d'une suite de raffinements et doivent donc posséder des types concrets. Les types concrets appartiennent à l'une des catégories suivantes [CLE 01] :

– types énumérés (qui contiennent le type `BOOL`),

– type entier : `INT`,

– tableaux à plusieurs dimensions, dont le type des éléments est concret,

– types différés, c'est-à-dire types définissant un ensemble donné de valeurs.

2.2. *Les appels d'opérations*

En B, les opérations sont déclarées au niveau des machines par un texte de la forme :

$$r \longleftarrow op\,(p) \mathrel{\widehat{=}} \text{PRE } P \text{ THEN } S \text{ END}$$

où p est une liste de paramètres formels d'entrée et r est une liste de paramètres formels de sortie. Le prédicat P est la précondition et S la substitution corps de l'opération. Un appel de l'opération op s'écrit : $v \longleftarrow op\,(e)$ où v est la liste des paramètres effectifs de sortie (une liste de variables) et e la liste des paramètres effectifs d'entrée (une liste d'expressions). La sémantique de l'appel d'opération dans le B-Book [ABR 96] est définie par la règle de substitution suivante :

$$v \longleftarrow op\,(e) \;\equiv\; [r, p := v, e] \text{ PRE } P \text{ THEN } S \text{ END} \qquad (1)$$

Les restrictions sur l'appel sont que v est une liste sans répétition de variables simples (et non des références à des éléments de tableau, par exemple) et que ces variables n'apparaissent pas dans P et S (non aliasing). La définition (1) est adaptée à la preuve, mais n'est pas opérationnelle pour la traduction. En effet, un appel d'opération en B doit pouvoir se traduire par un appel au code de cette opération. La règle ci-dessus n'est directement applicable que sur les substitutions du niveau machine, qui spécifient un changement d'état de manière atomique. Dans le cas des implémentations, qui peuvent contenir des séquencements ou des itérations, il faut utiliser la règle suivante :

$$v \longleftarrow op\,(e) \;\equiv\; \text{PRE } [p := e]\, P \text{ THEN VAR } p, r \text{ IN } p := e; S; v := r \text{ END END} \qquad (2)$$

qui correspond explicitement au passage par copie. Les restrictions sur la structuration des espaces de variables en B font que la règle (2) est équivalente à la forme :

$$v \longleftarrow op\,(e) \;\equiv\; \text{PRE } [p := e]\, P \text{ THEN VAR } p \text{ IN } p := e; [r := v]\, S \text{ END END} \qquad (3)$$

La raison essentielle de l'équivalence entre (2) et (3) est que les variables de la liste v n'apparaissent pas dans S. Il s'ensuit que les paramètres résultats peuvent être substitués, ce qui correspond à un passage par référence. C'est cette règle qui est utilisée par les traducteurs de l'AtelierB. Un résultat important des études théoriques du projet BOM est que les paramètres d'entrée peuvent aussi être passés par référence sous la condition suivante :

La liste des paramètres effectifs d'entrée ne contient aucune occurrence des paramètres effectifs résultats, ni aucune occurrence des variables référencées dans l'opération.

Sous ces hypothèses, il peut être montré que la règle (1) est applicable sur les substitutions admises dans les implémentations. La preuve est détaillée dans [POT 02]. Cette optimisation permet de traduire les appels d'opérations B par des passages par référence, ce qui est intéressant dans le cas de paramètres de type tableau par exemple, car la copie est évitée.

2.3. *Le modèle standard de calcul*

Le fait de déclarer en B une variable x vérifiant la propriété $x \in \texttt{INT}$ signifie simplement que sa valeur est un entier dans l'ensemble mathématique $\mathbb{Z}$ qui devra respecter l'invariant $x \in \texttt{MININT} \mathinner{..} \texttt{MAXINT}$. Une expression comme « $x + 1$ » ne nécessite aucune vérification dans les niveaux abstraits de B, puisque x est dans $\mathbb{Z}$, 1 est la dénotation d'un entier de $\mathbb{Z}$, et donc le $+$ mathématique est bien défini avec un résultat dans $\mathbb{Z}$. Il en va différemment en B_0 où les types, comme dans tout langage de programmation, sont finis. L'expression « $x + 1$ » fait alors implicitement référence à une opération arithmétique dans le type d'implémentation INT. En B, une fonction est dénotée par la formule : $f \mathrel{\widehat{=}} \lambda x \cdot (P \mid E)$ qui signifie que si le prédicat P est vrai alors la valeur de $f(x)$ est E. Si l'on note $+_{\texttt{INT}}$ l'opération d'addition dans ce type INT, elle peut être décrite par :

$$+_{\texttt{INT}} \mathrel{\widehat{=}} \lambda(x, y) \cdot (x \in \texttt{INT} \land y \in \texttt{INT} \land x + y \in \texttt{INT} \mid x + y)$$

Cette définition fait apparaître des restrictions sur les arguments x et y. Ces restrictions assurent qu'il n'y a pas de débordement dans les calculs. De manière générale, l'application d'une fonction f à un argument e va produire une condition à vérifier qui est $e \in \mathsf{dom}(f)$. Dans notre exemple, une expression comme $e1 + e2$ produira les conditions $e1 \in \texttt{INT} \land e2 \in \texttt{INT} \land e1 + e2 \in \texttt{INT}$. Ces conditions sont qualifiées de « conditions de bonne définition ». Elles consistent à vérifier que les opérateurs partiels sont appliqués dans leur domaine de définition. Elles peuvent être vues, pour les fonctions, comme l'analogue des préconditions des opérations.

En B_0, les types de données sont bornés, il n'y a pas d'appel récursif et les obligations de preuve assurent que les boucles terminent. Il est donc théoriquement possible de connaître statiquement une borne supérieure de la taille de mémoire requise pour l'exécution d'un programme B_0.

Dans une implémentation, les obligations de preuve assurent que les calculs donnent les résultats attendus par la spécification des opérations (comme aux autres niveaux de développement) et que l'invariant est préservé. De plus, elles permettent de garantir que les valeurs manipulées sont toujours des valeurs du type B_0 spécifié. Les preuves sur le langage B_0 vont donc assurer l'absence d'erreur à l'exécution : la méthode B garantit la correction totale des programmes (sous réserve qu'il y ait suffi-

samment de place en mémoire). Cette caractéristique essentielle de B est fournie par les arguments suivants :

1) Préservation du typage fort. Les règles de typage statique imposent que dans une affectation la partie droite et la partie gauche aient le même type. Il en est de même pour la substitution des paramètres effectifs. L'égalité choisie est l'égalité de noms des types concrets.

2) Satisfaction des conditions d'utilisation. Les opérateurs partiels introduisent des conditions à vérifier pour traiter les cas de division par zéro, les conditions d'accès aux éléments de tableau et, comme montré ci-dessus, les débordements arithmétiques.

3) Initialisation des variables. L'initialisation des variables est garantie par construction après l'exécution de la clause INITIALISATION pour les variables globales ; elle est garantie par la preuve avant leur lecture, dans le cas des paramètres de sortie et des variables locales.

Les obligations de preuve liées aux opérateurs partiels ont été développées dans [BEH 98, BUR 00, ABR 02] et sont maintenant introduites dans la version 3.6 de l'AtelierB. Dans les versions précédentes, elles étaient produites de manière *ad hoc*. L'approche proposée dans la nouvelle version offre un mécanisme plus souple, que nous allons exploiter dans la suite. Remarquons aussi que le système de type de B_0 est pauvre. Il n'y a pas de notion de sous-type au sens de Ada, par exemple. En particulier, la déclaration $x \in 0..10$ produit `INT` comme type B_0 de x, sachant que la contrainte que la variable reste dans les bornes a été vérifiée statiquement par la démonstration des obligations de preuves.

3. Adaptabilité du modèle de calcul pour le langage C

Comme nous l'avons évoqué dans l'introduction, l'objectif est de pouvoir enrichir le langage B_0 afin que les développeurs maîtrisent plus finement le code qui va être produit. Dans le cas des cartes à puce, domaine d'application de BOM, les optimisations attendues concernent l'usage de la mémoire. Le développeur doit donc pouvoir manipuler la représentation des variables afin d'adapter la taille mémoire nécessaire, comme cela est offert en C. Une autre optimisation importante est le passage par référence (offert en C par la possibilité de manipuler les adresses des objets) qui a été analysé dans la section 2.2. D'autres optimisations sont liées aux plates-formes d'exécution. Par exemple, suivant le compilateur utilisé, il est plus efficace de programmer des boucles à l'aide de l'instruction « `for` » ou de l'instruction « `while` ». Nous nous intéressons ici aux extensions du langage B_0 qui permettent de manipuler les types entiers C. Nous adoptons une approche mixte dans laquelle certaines informations sur les transformations à faire sont attendues du développeur, des outils faisant automatiquement les autres transformations :

– les types concrets du langage B_0 sont étendus par les types C,

– un raffineur automatique génère les opérateurs adéquats dans les expressions, en fonction du type des opérandes,

– le mécanisme d'obligations de preuve permet de vérifier la correction de la traduction des opérateurs.

3.1. *Le modèle des types entiers en* C

Pour la spécification des opérations sur les entiers, la norme ISO [STA 99] a été respectée. Lorsque certains traits sont laissés dépendants de l'implémentation, nous avons choisi la sémantique habituellement adoptée par les compilateurs C. Dans ces cas bien identifiés, l'utilisateur du traducteur BOM doit s'assurer que son propre compilateur satisfait la spécification donnée, pour que la traduction soit valide dans son cas.

Suivant l'usage commun, il existe plusieurs tailles de représentation des entiers, chacune d'elles pouvant représenter des entiers signés ou non signés. Plus formellement, pour des architectures 16-bits, la correspondance entre les types C, leur définition mathématique et les types qui sont manipulés dans le langage B_0 est donnée dans le tableau ci-après.

Type B_0	Intervalle mathématique	Type C
t_int16	$-2^{15}..2^{15}-1$	`int`
t_int32	$-2^{31}..2^{31}-1$	`long int`
t_uint16	$0..2^{16}-1$	`unsigned int`
t_uint32	$0..2^{32}-1$	`unsigned long int`

La spécification mathématique des opérations sur les types C est relativement complexe. Il faut tenir compte des arrondis (troncature des résultats) en cas de débordement. Un petit exemple permet d'illustrer le problème. Supposons que l'on code les entiers signés sur 3 bits en complément à 2. L'intervalle des valeurs est donc $100..011$, c'est-à-dire $-4..3$. Une addition $3+2$ ($011+010$) donne 101. Cette représentation est interprétée comme la valeur -3 en C, d'où $3+2=-3$. Pour chaque opération arithmétique C, deux opérations sont définies en B_0, une dont le résultat est contenu dans le type et l'autre qui autorise le débordement. Nous donnons l'exemple de la spécification mathématique de l'addition C en représentation signée et non signée, en fonction des opérandes de précision n. La troncature est exprimée à l'aide de l'opération `mod` en B qui calcule le modulo sur des entiers positifs. Les autres opérateurs arithmétiques peuvent être décrits formellement de la même façon [VOI 02].

$$
\begin{array}{lcl}
add\_int_n & \widehat{=} & \lambda(x,y)\cdot(\ x\in t\_int_n \wedge y\in t\_int_n \wedge x+y\in t\_int_n \mid x+y\ ) \\
add\_uint_n & \widehat{=} & \lambda(x,y)\cdot(\ x\in t\_uint_n \wedge y\in t\_uint_n \wedge x+y\in t\_uint_n \mid x+y\ ) \\
\\
add\_int\_trunc_n & \widehat{=} & \lambda(x,y)\cdot(\ x\in t\_int_n \wedge y\in t\_int_n \\
 & & \qquad \mid ((x+y+2^{n-1}+2^n) \bmod 2^n)-2^{n-1}\ ) \\
add\_uint\_trunc_n & \widehat{=} & \lambda(x,y)\cdot(\ x\in t\_uint_n \wedge y\in t\_uint_n \mid (x+y) \bmod 2^n\ )
\end{array}
$$

Dans ces définitions, add_int et add_uint_trunc sont les opérations $+$ définies dans la norme ISO-C, respectivement sur les entiers signés et non signés. La fonction

add_int_trunc est celle qui est habituellement implantée dans les compilateurs en cas de débordement du $+$ signé. La fonction add_uint est fournie en complément, pour restreindre la fonction add_uint_trunc et faciliter les preuves en cas de non-débordement assuré. On peut vérifier que si $n = 3$, on a $add\_int\_trunc_3(3, 2) = ((3 + 2 + 2^2 + 2^3) \bmod 2^3) - 2^2 = -3$.

Nous avons aussi défini la valeur mathématique des conversions C entre entiers signés et non signés de même précision n, ainsi que les opérateurs d'extension ou de troncature pour les changements de précision.

3.2. *Les machines de base*

Une machine de base est une machine dont la seule particularité est de ne pas être développée en B. Initialement le langage B_0 n'offrait aucun moyen prédéfini pour manipuler les entités concrètes. Par exemple, l'introduction d'une variable entière se faisait en important une instance de la machine *BASIC_SCALAR_VAR* qui offrait des opérations permettant de lire et de mettre à jour cette variable ainsi qu'une instance de la machine *BASIC_ARITHMETIC* pour réaliser les opérations arithmétiques sur les `INT`. L'affectation $x := x + 3$ (où x : `INT`) devait s'écrire, avec $v1$ et $v2$ des variables locales :

VAR $v1, v2$ IN
  $v1 \longleftarrow x$.*VAL*;
  $v2 \longleftarrow$ *ADD*$(v1, 3)$;
  x.*SET*$(v2)$
END

L'opération x.*VAL* a pour effet de retourner la valeur de x. L'opération *ADD*(a, b), qui effectue l'addition entre deux entiers, a pour précondition $a \in$ `INT` $\wedge\ b \in$ `INT` $\wedge\ a + b \in$ `INT`. Finalement, l'opération x.*SET*(a) affecte à x la valeur a si la précondition $a \in$ `INT` est vérifiée. Le mécanisme des machines de base offre à l'utilisateur la possibilité de définir ses propres « instructions » et, par les préconditions, permet de spécifier les contraintes d'application. Par la suite, ce mécanisme a été allégé pour simplifier l'écriture des implémentations, avec l'introduction d'opérateurs arithmétiques B_0. Il est encore utilisé pour les entrées-sorties ou pour spécifier par exemple des opérations de niveau système.

Nous allons donc utiliser les machines de base pour étendre les types concrets. Un type concret est déclaré sous la forme d'une constante ensemble et les opérateurs sur ce type sont définis par des fonctions. La définition du type va donc décrire les contraintes relatives à ce type et les spécifications des fonctions introduisent les conditions pour lesquelles elles sont définies. La solution adoptée ici est plus souple que celle choisie initialement en B, qui nécessitait de décrire tous les calculs par des suites d'appels d'opérations. Elle permet de garder une notation fonctionnelle, tout en offrant à peu près la même possibilité d'extensibilité. Néanmoins, elle n'est pas aussi complète puisque l'affectation n'est pas une opération dont la précondition peut

être ajustée. Pour obtenir des vérifications de sous-typage, les fonctions de conversion devront être systématiquement introduites.

La machine de base *BASIC_INT16*, qui spécifie les types signés sur 16 bits en C, est partiellement décrite ci-après. Par rapport au B standard, la version du B_0 adopté dans le projet BOM [CLE 02b] demande à ce que le typage soit explicite lors de la déclaration des constantes concrètes (clause CONSTANTS). Une constante est donc déclarée par $x : T$ où T est le type de x. Un type peut être simple (entier, etc.) ou fonction. Dans la notation B ci dessous, « $\rightarrow$ » est le symbole de fonction totale alors que « $\nrightarrow$ » est le symbole d'une fonction partielle.

MACHINE
BASIC_INT16
DEFINITIONS
$MIN\_INT16\_VAL == -32768;$ /* $32768 = 2^{15}$ */
$MAX\_INT16\_VAL == 32767$ /* $32767 = 2^{15} - 1$ */
CONSTANTS
$t\_int16 = MIN\_INT16\_VAL .. MAX\_INT16\_VAL \wedge$
$MIN\_INT16 \in t\_int16 \wedge$
$MAX\_INT16 \in t\_int16 \wedge$
$add\_int16 \in (t\_int16 \times t\_int16) \nrightarrow t\_int16 \wedge$
$sub\_int16 \in (t\_int16 \times t\_int16) \nrightarrow t\_int16 \wedge$
$mult\_int16 \in (t\_int16 \times t\_int16) \nrightarrow t\_int16 \wedge$
$div\_int16 \in (t\_int16 \times t\_int16) \nrightarrow t\_int16 \wedge$
$add\_int16\_trunc \in (t\_int16 \times t\_int16) \rightarrow t\_int16 \wedge$
...
PROPERTIES
$MIN\_INT16 = MIN\_INT16\_VAL \wedge$
$MAX\_INT16 = MAX\_INT16\_VAL \wedge$
$add\_int16 = \lambda(x, y).(x \in t\_int16 \ \wedge \ y \in t\_int16 \ \wedge \ x + y \in t\_int16 \mid x + y) \wedge$
...
END

Figure 1. *Machine de base des entiers sur 16 bits*

La définition $t\_int16 = MIN\_INT16\_VAL .. MAX\_INT16\_VAL$ a pour effet de déclarer un nouveau type concret t_int16 qui sera utilisé pour typer les constantes, les variables et les paramètres d'opérations. Les différentes fonctions applicables sur ce type sont déclarées dans la clause CONSTANTS et spécifiées dans la clause PROPERTIES.

Le jeu de machines de base, pour les types arithmétiques C, contient la définition des types décrits dans le tableau de la section précédente et, pour chacun, la liste des fonctions arithmétiques utilisables. Ces spécifications ne correspondent pas exactement au langage C puisque nous adoptons ici un typage fort qui différencie les types

arithmétiques C. Ce typage oblige à donner des noms différents aux différentes versions des opérateurs $+, -$, etc. En explicitant les types et les conversions, le niveau B_0 offre les moyens de décrire finement les calculs effectués. Ceci se fait au détriment de la lisibilité et de la facilité d'écriture des implémentations. L'outil présenté ci-après permet d'automatiser le passage d'une expression mathématique au calcul C.

3.3. *Raffineur d'expressions de* B_0 *vers* C

Dans les niveaux abstraits, l'utilisateur dispose des opérateurs mathématiques sur les entiers $+, -, \ldots$, (cf. section 2.3). Au niveau de l'implémentation, les expressions arithmétiques doivent être spécifiées en utilisant uniquement les fonctions arithmétiques C offertes par les machines de base.

Le principe du raffineur automatique d'expressions est de produire automatiquement une implémentation d'un calcul arithmétique à partir d'une expression mathématique et du type C des variables. Les étapes du raffineur automatique consistent à suivre l'algorithme de conversion des opérandes de la norme C. Cet algorithme explicite les conversions qui sont générées automatiquement par un compilateur C et fait le choix des fonctions d'après la représentation des opérandes[2]. L'expression C obtenue doit avoir la même forme et la même valeur que l'expression mathématique. Or, cela n'est pas garanti par les opérateurs avec troncature. Prenons un exemple : soit x un entier non signé égal à 0 : « $x \in t\_uint16 \ \wedge \ x = 0$ ». En B, le résultat de la comparaison « $x > -1$ » est « vrai ». Dans la traduction en C, x est déclaré par :

```
unsigned int x = 0 ;
```

et l'évaluation de l'expression C « `x>-1` » rend le résultat « faux », en raison de la conversion de `-1` en un entier non signé. Pour éviter ce problème, le raffineur utilise les fonctions sans troncature, qui renforcent les conditions d'utilisation, afin d'assurer que le résultat calculé par les opérateurs C est identique au résultat de l'opération mathématique associée. Les obligations de preuve de l'implémentation produites par le raffineur introduisent les lemmes de bonne définition associés aux fonctions C, ainsi que les preuves de raffinement. Ces dernières sont trivialement vraies puisque garanties par la traduction.

Avec cette nouvelle approche, le raffinement de la comparaison « $x > -1$ » produit l'expression sans troncature « $x > \ int16\_to\_uint16(-1)$ » dont une des obligations de preuve de bonne définition introduites par la conversion : « $-1 \in 0..2^{16} - 1$ » ne peut pas être prouvée, montrant par là qu'il n'est pas possible de préserver la sémantique de B par une traduction littérale en C. Une solution consiste ici, dans le code source B_0 de l'implémentation, à forcer la conversion de x en entier signé (qui est possible et prouvable puisque $-2^{15} \leq 0 \leq 2^{15} - 1$). On obtient alors une comparaison entre entiers signés en C avec le même résultat qu'en B.

2. Cet outil a été spécifi é dans le projet BOM, mais il n'a pas été implémenté.

4. Traducteur adaptable de B_0 vers C

4.1. *Principes de la traduction et de la validation*

Le langage d'entrée est décrit par une grammaire. Pour donner un aperçu du traducteur BOM, nous donnons ci-dessous une partie de la syntaxe concrète de quelques instructions simplifiées du langage B_0 et la syntaxe abstraite correspondante. Il s'agit de l'affectation, de l'instruction vide, de l'appel d'opération et de l'instruction conditionnelle. Dans la syntaxe concrète, la catégorie *Terme* est celle des expressions, la catégorie *Condition* représente les prédicats et *Instruction* sont les instructions. Les crochets indiquent les parties optionnelles et les accolades indiquent les répétitions.

Instruction	::=	... | *Idf* `:=` *Terme* | `skip`
		| *Appel_opération* | *Conditionnelle* | ...
Appel_opération	::=	[*Idf* `<-`] *Idf* [`(` *Terme* { `,` *Terme* } `)`]
Conditionnelle	::=	`IF` *Condition* `THEN` *Instruction*
		[`ELSE` *Instruction*] `END`

Après la phase d'analyse syntaxique, le texte du programme se présente sous forme d'un arbre abstrait. Dans la syntaxe abstraite, les notions *Terme*, *Condition* et *Instruction* sont décrites respectivement par $EXPR$, $COND$ et $INST$. Les nœuds de l'arbre abstrait sont des « constructeurs » et sont spécifiés à l'aide d'un profil de fonction. Les fils d'un nœud (paramètres des constructeurs de l'arbre abstrait) apparaissent dans le même ordre que les catégories qu'ils représentent dans la syntaxe concrète pour faciliter la correspondance abstrait-concret.

Un *Appel_opération* donne deux nœuds différents dans l'arbre abstrait suivant qu'il y a un résultat ($InstAppelOpF$) ou non ($InstAppelOpP$). La suite des paramètres de l'opération est représentée par une séquence (éventuellement vide) d'expressions. Pour l'instruction conditionnelle, si la partie `ELSE` est absente, elle est représentée par le nœud de l'instruction vide $InstVide$. Les nœuds des quatre instructions considérées sont :

$$
\begin{array}{lcl}
InstDevEgal & \in & (IDF \times EXPR) \rightarrow INST \\
InstVide & \in & INST \\
InstAppelOpF & \in & (IDF \times IDF \times \mathsf{seq}(EXPR)) \rightarrow INST \\
InstAppelOpP & \in & (IDF \times \mathsf{seq}(EXPR)) \rightarrow INST \\
InstCond & \in & (COND \times INST \times INST) \rightarrow INST
\end{array}
$$

Le point de départ du traducteur est la syntaxe abstraite. Un traducteur prend donc en entrée un arbre abstrait du langage source et produit en sortie un arbre abstrait du langage cible. Il est spécifié inductivement par rapport à la syntaxe abstraite du langage à l'aide de schémas de traduction, appelés aussi règles de traduction. Les règles de traduction du B_0 seront détaillées dans la section 4.2. La mise en œuvre de ce traducteur est réalisée par un interpréteur des règles de traduction. Cette technique offre un moyen d'adapter les traducteurs, ce qui est, rappelons-le, un des objectifs

du projet BOM : il suffit simplement de changer le fichier des règles de traduction. L'architecture du traducteur est présentée à la figure 2.

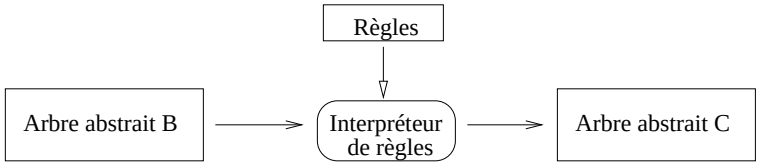

Figure 2. *Traducteur paramétré par des règles*

D'une manière générale, la validation d'un traducteur peut se décomposer en deux étapes [CUR 92, BUT 92] :

1) la correction de la spécification du traducteur,

2) la correction de l'implémentation du traducteur vis-à-vis de sa spécification.

Un traducteur met en jeu trois formalismes : le langage source, le langage cible et le langage d'implémentation du traducteur. La spécification d'un traducteur décrit le lien entre les programmes source et les programmes cible. Elle est correcte si les programmes cible *reflètent* la sémantique des programmes source qui leur sont associés. Si la correspondance entre les langages source et cible est décrite dans un langage suffisamment abstrait, cette validation est indépendante du langage d'implémentation du traducteur. C'est le cas dans notre approche, avec la description du traducteur par des schémas de traduction. Nous reviendrons en détail sur cette validation dans les sections 4.3 et 4.4.

La correction de l'implémentation du traducteur revient à vérifier la correction du code du traducteur vis-à-vis de sa spécification. Dans notre cas, elle est indépendante du langage source et du langage cible : il suffit de montrer que l'interpréteur de règles est correct pour ses fonctionnalités propres : filtrage et génération d'arbres abstraits.

Nous nous intéressons dans la suite à la validation de la spécification du traducteur. Elle consiste à montrer :

1) la correction des règles de traduction, vis-à-vis de la sémantique des langages source et cible ;

2) des propriétés intrinsèques au jeu de règles, essentiellement la complétude et la confluence qui vont garantir que tout programme du langage source peut être traduit. La cohérence est assurée par la correction des règles.

Avant de montrer comment ces validations peuvent être effectuées, nous présentons le formalisme utilisé pour les règles de traduction.

4.2. *Mise en œuvre du traducteur*

La traduction est spécifiée par un ensemble de règles. Ces règles sont interprétées par un outil développé spécifiquement dans le projet BOM. Il s'agit d'un interpréteur construit sur le *Logic Solver* fourni avec l'AtelierB. Le Logic Solver est un outil général de manipulation de formules qui est utilisé dans le cœur de l'AtelierB. Les règles de traduction sont groupées dans des *procédures* (ensembles de règles) qui expriment comment traduire chaque nœud d'un arbre abstrait B_0. Une règle de traduction se présente sous la forme :

nom d'opérateur (modèle)	en-tête de la règle
$\wedge$ *liste de gardes*	conditions d'application de la règle
$\Longrightarrow$	séparateur entre l'antécédent et le conséquent
traduction	effet de la règle si les conditions sont remplies

Un *modèle* est un schéma d'arbre de syntaxe abstraite de B_0 contenant des métavariables, appelées *jokers*. La *traduction* est un nœud d'arbre de syntaxe abstraite de C. La notion de *garde* va permettre d'exprimer des contraintes qui ne peuvent être prises en compte par le filtrage. Ces contraintes sont essentiellement des conditions sur la nature des objets manipulés (variables, constantes) ou sur le type de ces objets. L'interpréteur de règles prend l'arbre de syntaxe abstraite du programme B_0 en cours de traduction, choisit un nœud à traduire et le compare par filtrage avec les modèles des règles de traduction. Lorsqu'une règle s'applique (les conditions syntaxiques et les gardes sont satisfaites), alors l'arbre abstrait membre droit de la règle est généré avec substitution des métavariables de la règle par les valeurs obtenues lors du filtrage.

Pour bien comprendre la traduction, nous donnons un fragment de la syntaxe abstraite du langage C, d'après la formalisation de [GUR 93]. Les types des nœuds sont indicés par C. Le constructeur $IdExpr$ construit une expression qui est un identificateur ; $AssignExpr$ est l'affectation et $FunCall$ l'appel de fonction. Dans les instructions, $ExprStmt$ construit une instruction à partir d'une expression ; $IfStmt$ est l'instruction conditionnelle `if` et $VoidStmt$ est l'instruction vide. Comme pour la syntaxe abstraite de B_0, nous considérons que l'absence de partie `else` est codée par une instruction vide.

$$
\begin{array}{lcl}
IdExpr & \in & IDF_C \rightarrow EXPR_C \\
AssignExpr & \in & (EXPR_C \times EXPR_C) \rightarrow EXPR_C \\
FunCall & \in & (EXPR_C \times \mathsf{seq}(EXPR_C)) \rightarrow EXPR_C \\
 & & \\
ExprStmt & \in & EXPR_C \rightarrow INST_C \\
IfStmt & \in & (EXPR_C \times INST_C \times INST_C) \rightarrow INST_C \\
VoidStmt & \in & INST_C
\end{array}
$$

Nous donnons ci-après quelques règles pour illustrer le formalisme et les traitements. La fonction de traduction des expressions est appelée tr^E, celle qui traduit les conditions est tr^C et celle des instructions est tr^I. Les fonctions de traduction sont partielles parce que l'arbre abstrait doit avoir passé avec succès le contrôle des types, la vérification des obligations de preuve et des conditions propres au B_0 BOM. Sa-

chant que les conditions B_0 sont traduites par des expressions en C, le typage des fonctions de traduction est :

$tr^E \in EXPR \mapsto EXPR_C$	traduction des expressions
$tr^C \in COND \mapsto EXPR_C$	traduction des conditions
$tr^I \in INST \mapsto INST_C$	traduction des instructions

Les traductions d'une instruction conditionnelle, de l'instruction vide et de l'affectation simple sont directes. Notons qu'il y a une passe de renommage des identificateurs avant la traduction proprement dite [BER 03a], qui fait que les identificateurs de B_0 sont traduits par les mêmes identificateurs en C et que chaque identificateur nomme une entité unique et distincte des autres. Cette condition sur l'espace des noms simplifie la traduction et la validation et facilite la traçabilité entre les textes source et cible. Par exemple, l'instruction B d'affectation « $x := add\_int16(x, 1)$ » est traduite en C par « `x = x+1 ;` ».

$tr^I(\ InstCond(c, s_1, s_2)\ )$	instruction conditionnelle
$\Longrightarrow$	
$IfStmt(tr^C(c), tr^I(s_1), tr^I(s_2))$	
$tr^I(\ InstVide\ )$	instruction vide
$\Longrightarrow$	
$VoidStmt$	
$tr^I(\ InstDevEgal(v, e)\ )$	instruction d'affectation simple
$\Longrightarrow$	
$ExprStmt(AssignExpr(IdExpr(v), tr^E(e)))$	

Les règles de traduction ci-dessus n'ont pas de garde. Par contre, la traduction d'un appel d'opération avec paramètre de retour nécessite plusieurs cas. Le texte B_0 se présente, par exemple, sous la forme « $r \longleftarrow op(0, add\_int16(x, 1))$ » où r est une variable qui reçoit la valeur de retour de l'opération op. A ce stade, les types des paramètres effectifs sont corrects par rapport à la déclaration de l'opération op.

1) Le paramètre de retour r est scalaire : l'appel de l'opération est traduit par un appel de fonction C dont le résultat est affecté à la variable de retour r par l'instruction d'affectation : « `r = op(0,x+1) ;` ».

2) Le paramètre de retour r est un tableau : l'appel de l'opération est traduit par un appel de fonction C sans résultat et la variable de retour r est mise en dernier dans la liste des paramètres d'entrée. En effet, un tableau ne peut pas être affecté directement en C ; en revanche, il peut être passé en paramètre (appel par référence) (voir la section 2.2). Le code produit est donc : « `op(0,x+1,r) ;` ».

Evidemment, la déclaration de l'opération op a été traduite avec la même stratégie en une déclaration de fonction C. La différence entre les deux cas de traduction est réalisée à l'aide d'une garde qui teste la nature de la variable de retour. Notons l'uti-

lisation de l'expression $map(f)(s)$ qui applique la fonction f à chaque élément de la séquence s et retourne la séquence résultat. Enfin, l'expression $s \leftarrow e$ signifie ajouter l'élément e à la fin de la séquence s.

$tr^I(\ InstAppelOpF(r, o, p)\ )$ appel d'opération à résultat scalaire
$\wedge\ nature(r) = Scalaire$
$\Longrightarrow$
$ExprStmt(AssignExpr(IdExpr(r)$
$, FunCall(IdExpr(o), map(tr^E)(p))))$

$tr^I(\ InstAppelOpF(r, o, p)\ )$ appel d'opération à résultat tableau
$\wedge\ nature(r) = Tableau$
$\Longrightarrow$
$ExprStmt(FunCall(IdExpr(o), map(tr^E)(p) \leftarrow IdExpr(r)))$

4.3. *Validation du traducteur*

La validation de la spécification d'un traducteur reste, dans toute sa généralité, un problème complexe. Néanmoins nous sommes ici dans un cas particulier : le langage B_0 est intrinsèquement simple et la distance entre le langage source et le langage cible est relativement petite. En effet, le langage B_0 contient peu de constructions et de plus ne permet pas l'introduction d'alias. Il s'ensuit que la sémantique du langage B_0 est compositionnelle. Il suffit de décrire la sémantique de chaque construction pour obtenir la sémantique d'un programme.

La validation des règles se fait par rapport à une définition sémantique des langages. A proprement parler, le langage B n'a pas de sémantique opérationnelle. En revanche, il possède une définition axiomatique basée sur les transformateurs de prédicats. Pour le niveau B_0, il est possible de définir une sémantique opérationnelle qui prend en compte les pas de calcul et qui est compatible avec la sémantique des transformateurs de prédicats [HOA 98]. En ce qui concerne les langages cibles, la sémantique est fournie soit d'une manière formelle, soit de manière informelle à l'aide d'une norme. Le langage C est défini par une norme ISO [STA 99]. Pour chaque règle de traduction, la validation peut se faire en comparant la sémantique du texte source et la sémantique du texte cible de chaque règle [BUT 92]. Le schéma de la validation est représenté à la figure 3. La sémantique des programmes dans les deux langages repose sur l'observation des états de la mémoire.

La relation de comparaison doit garantir que les deux sémantiques sont observationnellement équivalentes. Si les divers éléments qui entrent en jeu sont décrits formellement, il est possible d'avoir un mécanisme formel de validation des règles, en précisant l'aspect observationnel. Néanmoins, cette approche nécessite, dans toute sa généralité, une description complète de la sémantique du langage C. Pour maîtriser cette difficulté l'approche choisie dans le projet est la suivante :

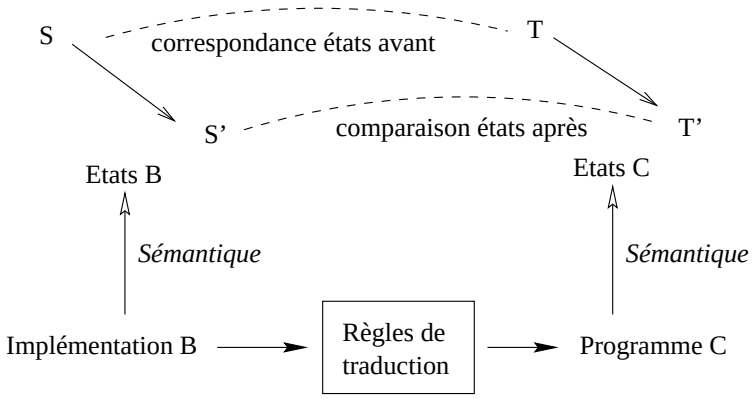

Figure 3. *Schéma de validation du traducteur*

– validation formelle d'un traducteur simple qui correspond à une traduction directe de B_0 vers C (le traducteur standard),

– validation (formelle ou non) d'un traducteur plus spécifique, et peut-être plus complexe, en référence au traducteur standard.

Comme précédemment cette approche va permettre de couper la complexité. Pour la première étape, une sémantique d'un sous-ensemble du langage C pourra être adoptée. Elle sera très proche de la sémantique du langage B_0, vu la simplicité de ce langage. Pour avoir un résultat convaincant, il est préférable de définir une sémantique formelle de ce sous-ensemble de C pour modéliser la sémantique informelle (issue de la norme ISO-C ou de celle des compilateurs cibles). Comme d'habitude, cette sémantique ne sera toutefois qu'une approximation de la réalité (par exemple, on supposera que la mémoire est infinie). Pour les travaux au sujet de la sémantique de C, voir [GUR 93, NOR 97] et les références citées.

Dans la pratique, les traducteurs spécifiques sont obtenus en remplaçant certaines règles (par exemple, codage d'un `while` par un `for`) ou en donnant des règles qui décrivent des cas particuliers de traduction standard (par exemple remplacement du codage d'une condition de la forme $e =$ `TRUE` par le codage direct de l'expression e). Le jeu de règles ainsi obtenu peut être validé par rapport au jeu de règles standard, en exprimant les nouvelles règles comme des combinaisons et des instanciations des règles standards et en vérifiant la préservation de la sémantique au niveau du C produit. Cette approche n'est bien sûr possible que pour certaines formes d'adaptabilité, qui peuvent être exprimées par des modifications locales qui ne nécessitent pas d'informations globales sur le programme (comme dans le cas de l'optimisation du passage de paramètres par exemple). Ce niveau d'adaptabilité est celui requis dans le projet BOM, dans lequel on veut pouvoir configurer la traduction en fonction des caractéristiques d'une plate-forme, les optimisations plus globales étant traitées au niveau du modèle B.

4.4. *Vers une vérification formelle*

Les premiers travaux sur la vérification formelle de compilateurs, avec un souci de mécanisation du processus de vérification, sont dus à W. Polak [POL 81], qui a développé une preuve de correction partielle d'un compilateur d'un sous-ensemble conséquent de Pascal, à l'aide de l'outil Stanford Pascal Verifier [GRO 79]. On peut aussi citer les travaux de J. S. Moore et son équipe [MOO 89, YOU 89] qui ont vérifié formellement la compilation d'un langage *à la Algol* à l'aide du démonstrateur de théorèmes NQTHM [BOY 90]. Des travaux plus récents ont porté sur d'autres langages, comme Scheme [GUT 95], OCCAM (projet Esprit ProCoS[3]) et Java [STR 02] par exemple. Le projet VERIFIX[4] a pour objectif de développer des outils permettant d'établir mathématiquement la construction de compilateurs, principalement pour des langages impératifs [ZIM 97].

Dans cette partie, on explicite plus formellement le diagramme de validation du traducteur (figure 3). Il s'agit de montrer que pour toute implémentation p en B_0, dont la correction totale a été prouvée dans un certain contexte c_B, le programme C correspondant p' dans un contexte c_C compatible, a un « effet observationnel » équivalent à celui réalisé par p. En particulier, p' ne produit pas d'erreur à l'exécution. Pour cela, il est nécessaire de disposer de :

1) la sémantique du langage source B_0,

2) la sémantique du sous-langage cible C utilisé,

3) une relation définissant l'équivalence observationnelle entre les valeurs B_0 et les valeurs C,

4) une relation définissant l'équivalence observationnelle entre les états des programmes B_0 et les états des programmes C.

Nous développons brièvement chacun de ces points.

4.4.1. *Sémantique de* B_0

L'ensemble des valeurs B_0 comprend les entiers, les booléens, les littéraux d'énumération et les tableaux. Il est noté $\mathcal{V}_B$. Ces valeurs sont fortement typées et à chaque type B_0 est associé un sous-ensemble de ces valeurs. La relation de typage entre une valeur et un nom de type est notée $v : \tau$. A l'exécution, il existe un *état mémoire*, composé de variables, que l'on note $VARS_B$[5]. En B_0, toutes les variables ont un type par une fonction de déclaration et une valeur après initialisation, par une fonction partielle de spécification de la mémoire. Ces deux sortes de fonctions ont respectivement les profils $\mathcal{D}_B$ et $\mathcal{M}_B$ suivants :

---

3. URL : http ://www.afm.sbu.ac.uk/procos/
4. VERIFIX - Provably Correct Compilers, URL : http ://www.info.uni-karlsruhe.de/~verifi x/
5. Pour faciliter la lecture, nous ne rentrons pas dans les détails de la structuration de la mémoire (variables globales, locales, etc.).

$\mathcal{D}_B = \mathit{VARS}_B \rightarrow TypeB_0$	déclaration des variables B
$\mathcal{M}_B = \mathit{VARS}_B \mapsto \mathcal{V}_B$	valeur des variables B

Un contexte en B est donc une paire de fonctions qui a pour type $\mathcal{C}_B = \mathcal{D}_B \times \mathcal{M}_B$. La sémantique opérationnelle du langage B_0 peut être construite par induction sur la syntaxe (section 4.1). Elle est définie par des fonctions d'interprétation des expressions[6] et des instructions [BER 03b]. La fonction d'interprétation des expressions prend en paramètre un arbre abstrait d'une expression et un contexte et retourne la valeur de l'expression pour ce contexte. La fonction d'interprétation des instructions prend en paramètre un arbre abstrait d'une instruction et un contexte, et retourne le contexte produit par l'exécution de l'instruction.

$Intrp^E_B \in \mathit{EXPR} \rightarrow \mathcal{C}_B \mapsto \mathcal{V}_B$	interprétation des expressions
$Intrp^I_B \in \mathit{INST} \rightarrow \mathcal{C}_B \mapsto \mathcal{C}_B$	interprétation des instructions

La partialité des fonctions d'interprétation exprime que les programmes B_0 auxquels on s'intéresse doivent terminer normalement. Les domaines des fonctions décrivent les cas pour lesquels l'évaluation termine et ne produit pas d'erreur. Ceci est garanti par la méthode B comme décrit dans la section 2.3.

4.4.2. *Sémantique de* C

Les valeurs C, notées $\mathcal{V}_C$, comprennent les chaînes de bits sur une longueur donnée et les adresses. En général, les chaînes de bits sont groupées en octets (ou bytes), demi-mots ou mots qui correspondent à des unités d'adressage. Nous ne rentrons pas dans ces détails ici.

Suivant [GUR 93], les variables sont typées par une fonction de déclaration de profil $\mathcal{D}_C$. La fonction environnement (profil $\mathcal{E}_C$) associe à chaque variable une adresse d'allocation et la fonction mémoire ($\mathcal{M}_C$) fournit la valeur C associée à une adresse et un type. Dans cette modélisation, nous supposons que les adresses utilisées sont des adresses valides (la fonction mémoire est totale) :

$\mathcal{D}_C = \mathit{VARS}_C \rightarrow TypeC$	déclaration des variables C
$\mathcal{E}_C = \mathit{VARS}_C \rightarrow Adr$	adresse d'allocation des variables C
$\mathcal{M}_C = (TypeC \times Adr) \rightarrow \mathcal{V}_C$	lecture de la mémoire C

Un contexte en C est un triplet de fonctions qui a pour type $\mathcal{C}_C = \mathcal{D}_C \times \mathcal{E}_C \times \mathcal{M}_C$. La sémantique de C repose sur l'interprétation des expressions et des instructions pour un certain contexte. Contrairement au langage B, les expressions peuvent modifier la mémoire. L'interprétation $Intrp^E_C$ retourne donc un couple constitué d'une valeur et d'un nouveau contexte.

$Intrp^E_C \in \mathit{EXPR}_C \rightarrow \mathcal{C}_C \mapsto (\mathcal{V}_C \times \mathcal{C}_C)$	interprétation des expressions
$Intrp^I_C \in \mathit{INST}_C \rightarrow \mathcal{C}_C \mapsto \mathcal{C}_C$	interprétation des instructions

6. Les conditions peuvent être traitées comme des expressions, ou par des fonctions d'interprétations spécifiques.

Comme pour le langage B_0, les domaines des fonctions d'interprétation décrivent les données pour lesquelles l'évaluation termine et ne produit pas d'erreur.

4.4.3. *Observabilité et compatibilité des valeurs et des états*

Le langage B_0 étant fortement typé, la correspondance des types permet de déterminer la compatibilité des valeurs. Dans la spécification de la traduction, chaque type B_0 a un correspondant en C par une fonction de représentation des types :

$$RepType \in TypeB_0 \rightarrow TypeC$$

Pour chaque type τ de B_0, il existe une fonction d'abstraction qui, pour des valeurs C du type de la représentation de τ, donne la valeur B abstraite qui est représentée :

$$AbsVal \in TypeB_0 \rightarrow \mathcal{V}_C \nrightarrow \mathcal{V}_B$$

Pour un type τ de B_0, le domaine de $AbsVal(\tau)$ est contenu dans les valeurs de C qui sont de type $RepType(\tau)$ et si $v' \in \mathsf{dom}(AbsVal(\tau))$ alors la valeur abstraite associée est de type τ, c'est-à-dire $AbsVal(\tau)(v') : \tau$. Le prédicat de correspondance observationnelle entre les valeurs typées B_0 et les valeurs typées C est défini par :

$$Obs_V \ \in \ (\mathcal{V}_B \times TypeB_0) \leftrightarrow (\mathcal{V}_C \times TypeC)$$

avec

$$\begin{array}{l}((v,\tau) \mapsto (v',\tau')) \in Obs_V \ \Leftrightarrow \\ \qquad v : \tau \ \wedge \ v' : \tau' \ \wedge \ \tau' = RepType(\tau) \ \wedge \ v = AbsVal(\tau)(v')\end{array}$$

La notation $x \mapsto y$ est équivalente à (x, y). Nous l'utilisons lorsque les éléments x et y sont eux-mêmes des n-uplets de valeurs, par souci de lisibilité.

Dans la traduction, les noms des variables de B_0 sont inclus dans les noms des variables de C, grâce au renommage préalable qui permet de conserver le même espace de noms. Lorsque les variables sont initialisées, à un point donné des programmes B_0 et C en correspondance dans la traduction, $d_B \in \mathcal{D}_B$ et $m_B \in \mathcal{M}_B$ étant respectivement des déclarations et une mémoire en B_0, les conditions suivantes sont satisfaites par construction :

$VARS_B \subseteq VARS_C$	toutes les variables B_0 sont représentées en C
$\mathsf{dom}(m_B) = \mathsf{dom}(d_B)$	toutes les variables déclarées sont initialisées

Soit $d_C \in \mathcal{D}_C$, $e_C \in \mathcal{E}_C$ et $m_C \in \mathcal{M}_C$ respectivement des déclarations, des allocations et une mémoire en C, alors le contexte d'exécution (d_B, m_B) d'un programme en B_0 est *compatible* avec le contexte d'exécution (d_C, e_C, m_C) d'un programme en C si les valeurs des mêmes variables dans les deux contextes sont observationnellement équivalentes. Formellement, l'équivalence observationnelle entre états est définie par :

$$Obs_M \ \in \ \mathcal{C}_B \leftrightarrow \mathcal{C}_C$$

avec

$$\begin{array}{l}((d_B, m_B) \mapsto (d_C, e_C, m_C)) \in Obs_M \ \Leftrightarrow \\ \qquad \forall x \cdot (x \in \mathsf{dom}(m_B) \ \Rightarrow \\ \qquad\qquad (m_B(x), d_B(x)) \mapsto (m_C(d_C(x), e_C(x)), d_C(x)) \in Obs_V)\end{array}$$

4.4.4. *Correction des règles du traducteur*

La spécification de la traduction (section 4.2) est correcte si les conditions suivantes sont satisfaites :

1) les contextes initiaux $ci_B \in \mathcal{C}_B$, et $ci_C \in \mathcal{C}_C$ vérifient $(ci_B \mapsto ci_C) \in Obs_M$,

2) les règles de traduction transforment les programmes B qui terminent normalement en programmes C qui terminent normalement,

3) les règles de traduction préservent la correspondance observationnelle.

Pour chacune des règles de traduction, la propriété d'équivalence observationnelle du contexte et des résultats doit être démontrée. Donc, pour toute instruction i, pour toute expression e du langage B_0 et pour tout couple de contexte (c_B, c_C), les conditions de validité du point 2 sont :

2.1 $e \in \mathsf{dom}(tr^E) \ \wedge \ c_B \in dom(Intrp^E_B(e)) \ \wedge \ (c_B \mapsto c_C) \in Obs_M$
$\Rightarrow \ c_C \in dom(Intrp^E_C(tr^E(e)))$

2.2 $i \in \mathsf{dom}(tr^I) \ \wedge \ c_B \in dom(Intrp^I_B(i)) \ \wedge \ (c_B \mapsto c_C) \in Obs_M$
$\Rightarrow \ c_C \in dom(Intrp^I_C(tr^I(i)))$

Pour toute instruction i, pour toute expression e du langage B_0 et pour tout couple de contexte (c_B, c_C), les conditions de validité du point 3 sont :

3.1 $e \in \mathsf{dom}(tr^E) \ \wedge \ c_B \in dom(Intrp^E_B(e)) \ \wedge \ (c_B \mapsto c_C) \in Obs_M \ \wedge$
$Intrp^E_C(tr^E(e)(c_C)) = (v', c'_C)$
$\Rightarrow \ ((Intrp^E_B(e)(c_B), \tau) \mapsto (v', \tau')) \in Obs_V \ \wedge$
$(c_B \mapsto c'_C) \in Obs_M$

3.2 $i \in \mathsf{dom}(tr^I) \ \wedge \ c_B \in dom(Intrp^I_B(i)) \ \wedge \ (c_B \mapsto c_C) \in Obs_M$
$\Rightarrow \ (Intrp^I_B(i)(c_B) \mapsto Intrp^I_C(tr^I(i))(c_C)) \in Obs_M$

avec τ le type de l'expression e et τ' le type de $tr^E(e)$.

4.5. *Discussion au sujet de la validation formelle du traducteur*

La démarche décrite ci-dessus donne les principes de la validation formelle du processus de traduction, règles par règles. Vu la simplicité des constructions du langage B_0 et par là même, des constructions C utilisées, puisque nous visons un processus de traduction qui produise du code le plus proche possible de l'implémentation, il n'y a pas de difficultés propres aux langages. Néanmoins, les points délicats sont que certaines informations sur le modèle B doivent être rendues explicites afin d'assurer une vérification formelle de la traduction. Ce sont :

1) les propriétés du langage B_0 qui permettent d'établir la correction des règles de traduction (absence d'alias, hypothèses sur les paramètres effectifs…) ;

2) les propriétés des programmes B_0 qui assurent la correction totale (pas de « bouclages » et pas d'erreurs à l'exécution) et qui vont permettre d'établir la correction totale des programmes C. Ces propriétés ne sont actuellement pas vraiment explicitées car elles sont garanties par différents mécanismes (preuves de terminaison des substitutions, preuves produites par le vérifieur de préconditions des fonctions partielles et mécanisme d'initialisation des variables).

Ces points sont principalement relatifs au langage B_0, dont nous cherchons à donner une définition opérationnelle complète et à expliciter les hypothèses sur le langage et les programmes. La dernière difficulté est de choisir une sémantique formelle du langage visé C, suffisamment abstraite [GUR 93], mais néanmoins suffisamment proche des mécanismes d'exécution, pour pouvoir garantir certaines propriétés lors de la vérification. Le rapport [BER 03b] décrit la syntaxe abstraite, les conditions de validité et la sémantique opérationnelle d'une implémentation B_0. Il donne la syntaxe abstraite du sous-langage C utilisé dans la traduction et une sémantique opérationnelle. Il décrit les règles de traduction et la représentation des valeurs. Les conditions de vérification des règles sont celles données dans cet article.

La vérification formelle et manuelle de quelques règles est en cours. Elle devrait permettre d'apprécier l'importance des conditions implicites citées ci-dessus et d'étudier comment les expliciter. Elle devrait aussi permettre de formuler plus abstraitement le processus de validation du traducteur, en vue d'une automatisation de la vérification. Cela serait particulièrement utile lors de l'écriture de nouvelles règles de traduction pour les traducteurs spécifiques.

5. Application des traducteurs sur un cas d'étude

Un cas d'étude important a été développé dans le projet BOM [REQ 03]. Il a été présenté en [BER 03a]. Il s'agit de la spécification, du raffinement et de l'implémentation de la machine virtuelle Java Card (JCVM). Plus précisément, nous avons développé entièrement le « Java Card Runtime Environment » (JCRE) et l'interpréteur de byte-code. Le « loader » et « linker » ont été simulés par des programmes écrits directement en C, de même que les interfaces avec les plates-formes sous-jacentes. La taille du cas d'étude est de 10 000 lignes de code B, réparties en 12 machines, 6 raffinements et 12 implémentations. Il a été nécessaire d'écrire 9 machines de base pour réaliser les interfaces. Le partenaire Gemplus a choisi deux plates-formes significatives du domaine : le microcontrôleur RISC 8-bits Atmel AVR[7] et le processeur MIPS 32-bits SmartMIPS[8]. Nous ne détaillons pas davantage les caractéristiques de ces plates-formes. Le lecteur intéressé peut se reporter à [REQ 03].

Le jeu de règles du traducteur standard a été complètement réalisé. Des règles plus spécifiques au contexte des cartes à puce ont été mises au point par Gemplus pour traiter des cas simples d'adaptation. Un optimiseur par expansion de code en ligne a

7. URL : http ://www.atmel.com/products/AVR/.
8. URL : http ://www.mips.com/products/s2p12.html.

également été réalisé de manière à éviter les appels d'opérations pour la génération vers les cartes à puce. Ces deux traducteurs, BOM-G (général) et BOM-C (cartes à puce), ont été évalués sur le cas d'étude. L'évaluation a porté sur la comparaison de la taille du code généré par ces deux traducteurs, par le traducteur standard de l'AtelierB et par un traducteur prototype (SimpleC), qui avait été réalisé de manière *ad hoc* par Gemplus pour tester la faisabilité de l'approche. Le premier tableau compare les résultats des différents traducteurs sur le cas d'étude. La taille du code généré est donnée en octets. Le gain exprime l'avantage obtenu par rapport au code généré par l'AtelierB.

Atmel AVR	Atelier B	BOM-G	BOM-C	SimpleC
Taille du code	17 994	5 760	5 216	5 915
Gain		68 %	71 %	67 %
SmartMIPS	Atelier B	BOM-G	BOM-C	SimpleC
Taille du code	14 218	8 880	6 268	8 704
Gain		38 %	56 %	39 %

Une autre comparaison porte sur le JCRE, dont Gemplus possédait une version en C. Ce deuxième tableau compare la taille du code généré pour ce module avec l'implémentation écrite directement en C. Le surplus indique la perte de place par rapport à cette implémentation manuelle. On remarque que le traducteur BOM-C produit un code acceptable. De plus, l'augmentation de la taille est compensée par l'avantage que peut apporter la méthode B du point de vue de la certification.

Atmel AVR	Code C	BOM-C	BOM-G	Atelier B
Taille JCRE	537	596	634	1 704
Surplus		11 %	18 %	217 %
SmartMIPS				
Taille JCRE	536	588	652	904
Surplus		10 %	22 %	69 %

6. Conclusion

L'objectif du travail décrit dans cet article, qui est celui du projet RNTL BOM, est de proposer une approche de la traduction du langage B_0 qui soit validable et adaptable. L'aspect adaptabilité est offert à la fois par le mécanisme des machines de base et par le paramétrage de la phase de traduction à l'aide de règles.

La première forme d'adaptabilité permet, dans une certaine mesure, d'étendre le langage B_0. Ce mécanisme apparaît assez général puisqu'il permet de définir des types abstraits. Une limitation est liée au fait que le seul moyen de paramétrer les obligations de preuve est d'introduire, par l'intermédiaire des machines de base, des conditions à vérifier (précondition d'opération et opérateur partiel). Il n'est donc pas possible de

contraindre par exemple l'affectation ou le passage de paramètres, de l'extérieur du langage B. La validation de cette forme d'extension repose principalement sur le fait que les spécifications des machines de base sont correctes, c'est-à-dire correspondent aux notions du langage cible.

La seconde forme d'adaptabilité, sur la forme de la traduction, est rendue possible par l'architecture choisie, basée sur des règles de traduction et un interpréteur de règles. Dans ce cas la validation est nécessairement complexe, puisqu'elle fait intervenir plusieurs langages et leur sémantique. Le principe de la validation pour le traducteur de B_0 vers C a été présenté. La validation se limite à une validation règle par règle. On a, de cette manière, des conditions suffisantes pour assurer la correction des programmes traduits. Notre technique ne permet pas de prouver l'équivalence de programmes B_0 et C dans toute leur généralité, mais seulement ceux qui sont source et cible du processus de traduction donné.

Rappelons, pour terminer, que la problématique abordée dans ce papier — adaptabilité d'un processus de traduction/compilation — est d'actualité pour la plupart des développeurs d'outils pour systèmes embarqués. La décomposition de la traduction en plusieurs phases, la simplicité du langage source, la proximité entre le langage source et le sous-ensemble du langage cible utilisé nous ont permis de développer des solutions effectives pour la réalisation des outils et la validation. Le schéma de traduction et certaines parties du traitement sont très largement réutilisables pour d'autres langages cibles, ce qui confère une certaine généralité à l'architecture définie pour ce traducteur.

7. Bibliographie

[ABR 96] ABRIAL J.-R., *The B Book - Assigning Programs to Meanings*, Cambridge University Press, August 1996.

[ABR 02] ABRIAL J.-R., MUSSAT L., « On Using Conditional Definitions in Formal Theories », BERT D., BOWEN J. P., HENSON M. C., ROBINSON K., Eds., *ZB2002 : Formal Specification and Development in Z and B, LNCS 2272*, Springer-Verlag, 2002, p. 242-269.

[BEH 98] BEHM P., BURDY L., MEYNADIER J.-M., « Well Defined B », BERT D., Ed., *Proceedings of the Second International B Conference*, LNCS 1393, Springer-Verlag, 1998.

[BEH 99] BEHM P., BENOIT P., FAIVRE A., MEYNADIER J.-M., « Météor : A Successful Application of B in a Large Project », *FM'99 - Formal Methods, LNCS 1708*, Springer-Verlag, 1999, p. 369-388.

[BER 03a] BERT D., BOULMÉ S., POTET M.-L., REQUET A., VOISIN L., « Adaptable Translator of B Specifications to Embedded C Programs », *Proc. of the 12th International FME Symposium, FM 2003, LNCS 2805, Pise*, Springer-Verlag, 2003, p. 94-113.

[BER 03b] BERT D., BOULMÉ S., POTET M.-L., Sémantique du B_0 BOM aplati et définition de la traduction de B_0 en B, rapport n° D10, 2003, projet RNTL BOM.

[BOY 90] BOYER R. S., MOORE J. S., « A Theorem Prover for a Computational Logic », *10th Conference on Automated Deduction, LNCS 449*, Springer-Verlag, 1990.

[BUR 00] BURDY L., Traitement des expressions dépourvues de sens de la théorie des ensembles. Application à la méthode B, Thèse de Doctorat, Conservatoire National des Arts et Métiers, 2000.

[BUT 92] BUTH B., BUTH K.-H., FRÄNZLE M., VON KARGER B., LAKHNECH Y., LANGMAACK H., MÜLLER-OLM M., « Provably Correct Compiler Development and Implementation », KASTENS U., PFAHLER P., Eds., *CC'92 : Compiler Construction, LNCS 641*, Springer-Verlag, 1992, p. 141-155.

[CLE 01] CLEARSY, B Language Reference Manual, version 1.8.5, rapport, 2001, ClearSy System Engineering, URL :http ://www.clearsy.com/.

[CLE 02a] CLEARSY, Atelier B, version 3.6, rapport, 2002, ClearSy System Engineering, URL : http ://www.atelierb.societe.com/.

[CLE 02b] CLEARSY, Manuel de référence du B0-BOM, version 1.1, rapport, 2002, ClearSy System Engineering.

[CUR 92] CURZON P., Of What Use is a Verified Compiler Specification ?, rapport n° 274, 1992, Univ. of Cambridge, Computer Laboratory.

[FOR 89] FORIN P., « Vital Coded Microprocessor : Principles and Application for Various Transit Systems », *Proc. IFAC-GCCT*, 1989.

[GRO 79] GROUP S. V., Stanford Pascal verifier user manual, rapport n° 11, 1979, Stanford University.

[GUR 93] GUREVITCH Y., HUGGINS J. K., « The Semantics of the C Programming Language », *Selected papers from CSL'92 (Computer Science Logic), LNCS 702*, Springer-Verlag, 1993, p. 274-308.

[GUT 95] GUTTMAN J. D., RAMSDELL T. D., WAND M., « VLISP : A Verified Implementation of Scheme », *Lisp and Symbolic Computation*, vol. 8, n° 1-2, 1995.

[HOA 98] HOARE C. A. R., JIFENG H., *Unifying Theories of Programming*, Prentice Hall, 1998.

[MOO 89] MOORE J. S., « A Mechanically Verified Language Implementation », *Journal of Automated Reasoning*, vol. 5, n° 4, 1989, p. 461-492, Kluwer Academic Publishers.

[NOR 97] NORRISH M., An Abstract Dynamic Semantics of C, rapport n° 421, mai 1997, Univ. of Cambridge, Computer Laboratory.

[POL 81] POLACK W., « Compiler Specification and Verification », *LNCS 124*, Springer-Verlag, 1981.

[POT 02] POTET M.-L., « Spécifications et développements formels : Etude des aspects compositionnels dans la méthode B », Habilitation à Diriger des Recherches, INPG, 2002.

[REQ 03] REQUET A., Évaluation du traducteur C, rapport n° D11, 2003, projet RNTL BOM.

[STA 99] STANDARD I., « Programming languages —C », ISO/IEC 9899 :1999 (E), 1999.

[STR 02] STRECKER M., « Formal Verification of a Java Compiler in Isabelle », VORONKOV A., Ed., *18th Conference on Automated Deduction, CADE-18, LNCS 2392*, Springer-Verlag, 2002.

[VOI 02] VOISIN L., Calcul sur entiers en C et Java. Modélisation en B, rapport, février 2002, ClearSy System Engineering, URL :http ://www.clearsy.com/.

[YOU 89] YOUNG W. D., « A Mechanical Verified Code Generator », *Journal of Automated Reasoning*, vol. 5, n° 4, 1989, p. 493-518, Kluwer Academic Publishers.

[ZIM 97] ZIMMERMANN W., PAUL T., « On the Construction of Correct Compiler back-ends : an ASM approach », *Journal of Universal Compiler Science*, vol. 3, n° 5, 1997.

Article reçu le 10 avril 2003
Version révisée le 26 janvier 2004
Rédacteur responsable : Jeanine Souquières

Frédéric Badeau *est ingénieur chez ClearSy System Engineering où il a notamment travaillé sur le langage B. Actuellement, il anime des sessions de formation à la méthode B et participe à des projets industriels en rapport avec la sûreté de fonctionnement et l'utilisation de méthodes de modélisation, dont notamment la méthode B.*

Didier Bert *est chargé de recherche au CNRS. Membre du laboratoire Logiciels, Systèmes, Réseaux à l'IMAG, ses activités de recherche portent sur les méthodes de spécification algébrique et le développement rigoureux de logiciels avec la méthode B. Il coordonne un groupe de travail sur la méthode B au GRD ALP du CNRS. Il est également membre du groupe WG1.3 (Foundations of Software Specification) de l'IFIP.*

Sylvain Boulmé *est maître de conférences à l'Ecole Nationale Supérieure d'Informatique et de Mathématiques Appliquées de Grenoble, et membre du laboratoire Logiciels, Systèles, Réseaux. Son activité de recherche porte sur la preuve de programmes.*

Christophe Métayer *est ingénieur chez ClearSy System Engineering où il a notamment travaillé sur les outils de l'atelier B. Actuellement, il mène des travaux de recherche, en collaboration avec le laboratoire Triskelle de l'IRISA, sous la direction d'Yves le Traon et de Jean-Marc Jézéquel. La thèse a pour objectif d'étudier les utilisations possibles des statecharts dans la modélisation en B événementiel.*

Marie-Laure Potet *est maître de conférences habilitée de l'Ecole Nationale Supérieure d'Informatique et de Mathématiques Appliquées de Grenoble, et membre du laboratoire Logiciels, Systèmes, Réseaux. Ses travaux portent sur la construction formelle de logiciels, et notamment sur le raffinement vu comme un outil permettant de maîtriser et tracer le processus de développement, de la construction des spécifications au code qui sera exécuté.*

Nicolas Stouls *est doctorant en première année de thèse à l'INP de Grenoble. Sous la tutelle de Marie-Laure Potet et Sylvain Boulmé, il effectue une thèse, financée conjointement par le CNRS et ST Microelectronics, sur les outils formels pour la spécification et le développement modulaires de systèmes.*

Laurent Voisin *est ingénieur chez ClearSy System Engineering, dont il dirige le pôle outils (développement et maintenance). Cette activité l'amène à s'intéresser tout particulièrement aux techniques de génie logiciel et aux applications de la méthode B dans un contexte industriel.*