ABTools, une suite d'outil pour la méthode B développé avec ANTLR

Jean-Louis BOULANGER

Résumé

Dans cet article, nous présentons l'environnement *ABTools* qui est une suite d'outil pour la méthode B . Cet environnement est développé avec le générateur de compilateur *ANTLR*.

ANTLR est un environnement de génération de compilateur. Il permet de couvrir l'ensemble des phases de compilation : lexing, parsing, typing et génération de code (JAVA, C, HTML, XML, ...).

Keywords : ABTools, ANTLR, Méthode Formelle, Méthode B , Outils. Raffinement.

Table des matières

Introduction

Le but de ce papier est de présenter l'environnement de développement *B* intitulé **Another's B Tools**. Etant donné que nous utilisons JAVA comme langage support pour cette application, cet environnement open source est accessible sur différentes plateformes (Linux, Windows NT et Windows 98). L'utilisation d'ANTLR[1] permet de mettre très rapidement en place une extension de la méthode *B* tout en contrôlant la cohérence de l'ensemble.

Le but de ce travail n'est pas de concurrencer les ateliers commerciaux tels que l'*Atelier B* de la société Clearsy ou le *BTool* de la société BCore, mais bien de mettre à la disposition de la communauté des outils *open source* permettant d'étudier le langage *B* et ses extensions.

1 Présentation de la méthode *B*

La méthode *B* [2], a été développée par Jean-Raymond Abrial, c'est une méthode formelle dite «orientée modèle», c'est à dire de la même famille que *Z* [12] et *VDM* [8] mais qui permet un développement incrémental de la spécification jusqu'au code, au travers de la notion de raffinement [10].

À chaque étape du développement *B* , des obligations de preuves sont générées afin de garantir la validité du raffinement et la consistance de la machine abstraite. La notion de machine abstraite est similaire à la notion de module et/ou d'objet que l'on retrouve dans les langages de programmation classique. Le concept central étant l'encapsulation, l'évolution de l'état d'une machine abstraite ne doit se faire qu'au travers des services/opérations de celle-ci.

Les machines abstraites se répartissent en trois niveaux, les *machines* qui décrivent le niveau de spécification le plus abstrait, les *refinements* qui décrivent les étapes intermédiaires entre la spécification et le code et enfin, les *implementations* qui définissent le codage.

La méthode *B* repose sur un langage unique également appelé «Abstract Machine Notation» (noté AMN) qui permet de décrire ces trois niveaux d'abstraction.

Les machines abstraites sont composées de trois parties : la partie déclarative, la partie de composition et la partie exécutive.

[1] http ://www.antlr.org

La partie déclarative permet de décrire l'état d'une machine abstraite au travers de variables, de constantes, d'ensembles, et surtout de propriétés que doit toujours vérifier l'état de la machine. Cette partie est basée sur la théorie des ensembles et les prédicats du premier ordre.

```
MACHINE
  STACK (max_object)
 CONSTRAINTS
  max_object : NAT1
 SEES
  OBJECT
 VARIABLES
  stack
 INVARIANT
    stack  : seq(Object) &
    size(stack) <= max_object
 INITIALISATION
    stack := <>
 OPERATIONS
   PUSH (XX) =
    PRE XX : Object    &
        size(stack) < max_object
     THEN stack := stack <- XX END;
   XX <-- POP =
    PRE  size(stack) > 0
      THEN
      XX,stack:=last(stack),front(stack)
    END
END
```

Les clauses de composition (SEES, INCLUDES, IMPORTS et EXTENDS) permettent de décrire les différents liens entre machines abstraites, chaque clause introduisant des règles de visibilité sur les variables et les opérations de la machine abstraite concernées.

La partie opérationelle, quand à elle, contient l'initialisation et les opérations de la machine abstraite, elle est basée sur le langage des substitutions généralisées (noté GSL). Les substitutions généralisées sont une extension des commandes gardées de Dijkstra [4] sur les substitutions.

Mais l'ensemble des substitutions n'est pas utilisable à tous les niveaux, par exemple la substitution ANY xx WHERE P(xx) THEN S END ($\forall$ xx.P(xx) => [S]) est utilisable aux niveaux abstraits (Machine et Raffinement) tandis que la substitution WHILE pred DO inst END ne peut être acceptée au niveau le plus abstrait.

Pour terminer cette brève présentation, le $B0$ est le sous langage de l'AMN utilisé dans les implémentations (dernier étape de raffinement). Le $B0$ utilise la notion d'instruction, ce qui permet finalement d'utiliser un générateur de code vers un langage classique (C,ADA).

La mise en place d'un environnement permettant la manipulation du langage B doit prendre en compte toutes ces restrictions.

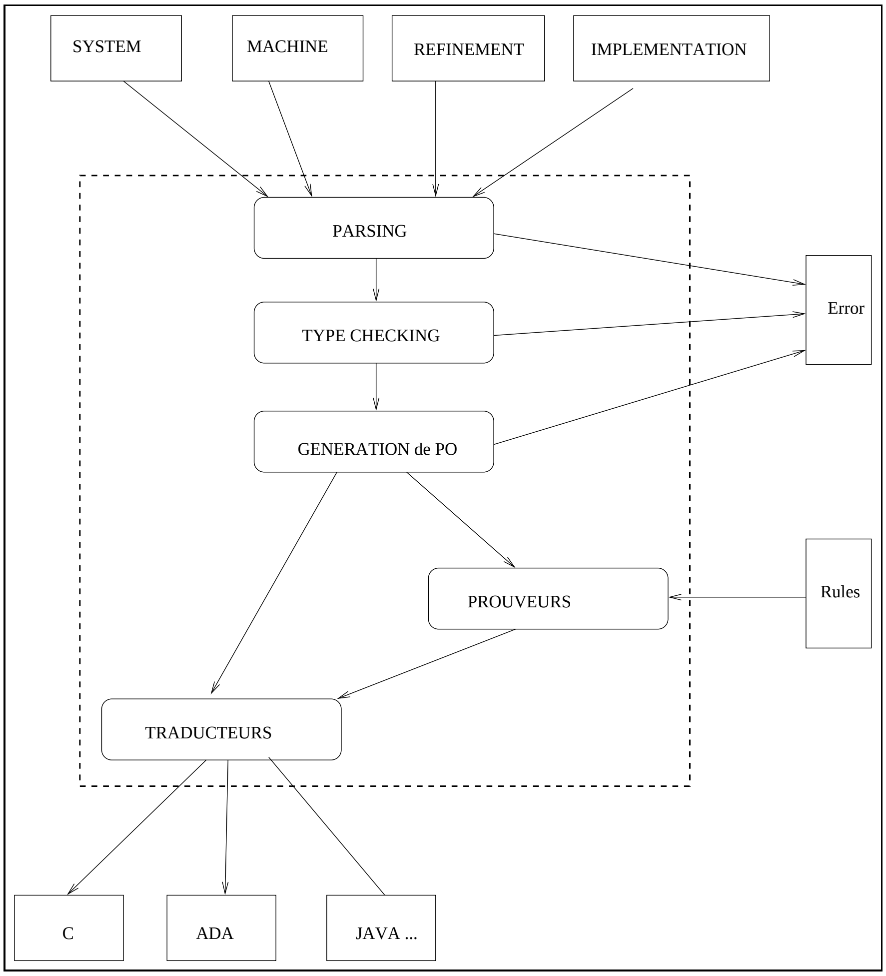

FIG. 1 – Un environnement B

2 Présentation de la problématique

Il existe peu de travaux concrets liés aux aspects syntaxiques, sémantiques de la méthode B . A titre d'exemple, plusieurs grammaires ont déjà été proposées, mais aucune ne semble satisfaisante :

- [2] contient une grammaire BNF plus fonctionnelle qu'opérationnelle, c'est à dire qu'elle est trop imprécise pour pouvoir servir de base au développement complet d'un analyseur syntaxique.
- [7] est publié depuis Avril 1995 par la société B-CORE commercialisant le BToolkit, atelier de développement B. Le document est publiquement disponible sur Internet mais souffre d'un certaine obsolescence compte tenu des évolutions récentes du langage.
- [13] est sans doute le document le plus complet sur le sujet mais reste néanmoins non satisfaisant car, d'une part, la grammaire décrite n'a pas encore été vérifiée par une réalisation adéquate et, d'autre part, elle comporte certaines insuffisances.
- [9] est une grammaire, elle n'est peut être pas complète mais elle a l'avantage d'avoir été outillée.

Actuellement, il existe deux outils commerciaux, l'AtelierB de STERIA[2] et le BToolkit de BCORE[3], et quelques outils universitaires existent.

La réalisation d'outils (mesures de la qualité de projets B [5], éditeur[4] et analyseur syntaxique[5] pour le langage B) nécessite la définition d'une modélisation du texte formel B par une grammaire.

Le but essentiel de nos travaux est de construire un environnement complet prenant en compte l'ensemble du langage B et les restrictions qu'il introduit. Mais il faut pouvoir être capable d'implanter facilement une extension du langage.

3 ANTLR : un générateur de compilateur

ANTLR est un générateur de compilateur qui est mis à disposition, sous licence open source, par Terence Parr. ANTLR est un acronyme pour *ANother Tool for Language Recognition.* ANTLR n'est pas juste un générateur de compilateur mais bien un environnement de développement de compilateur.

A ce jour, cet environnement a permis de mettre en oeuvre plusieurs langages tels que C, JAVA, Verilog et SDL-2000([11]), la plupart des grammaires

[2]http ://

[3]http ://

[4]http ://lib.univ-fcomte.fr/PEOPLE/tatibouet/EditeurB.html

[5]http ://perso-wanadoo.fr/bruno.tatibouet/BParser_en.html

sont disponibles sur le site officiel. L'environnement est même directement intégrable dans les outils du commerce.

	Input Stream	Output Stream
Lexer	Character	Token
TokenStreamFilter	Token	Token
Parser	Token	Astract Syntax Tree
TreeParser	Abstract Syntax Tree	Astract Syntax Tree

TAB. 1 – Flot de données

La table 1 présente les quatre étapes permettant la reconnaissance d'un langage avec ANTLR. Lors de la première étape, le *Lexer* analyse le fot de caractère d'entrée et produit en sortie des *Tokens*, ces tokens ont tous un type et une valeur. En général, ce flot de tokens est directement en entrée d'un *parser* qui construit l'arbre syntaxique (noté AST). Mais il est possible de manipuler le flot de tokens afin d'introduire, de détruire ou de modifier des tokens au travers d'un filtre (*TokenStreamFilter*). Pour finir, un parcoureur d'arbre (*TreeWalker*) permet d'évaluer l'AST. Cette évaluation peut éventuellement produire un nouvelle AST

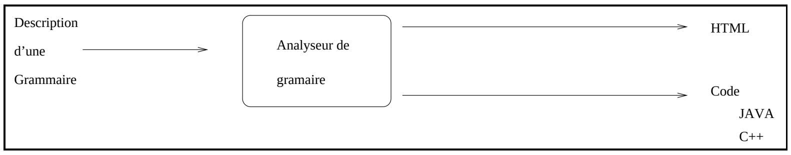

FIG. 2 – L'environnement ANTLR

ANTLR met à disposition un langage unique pour décrire chaque étape de la reconnaissance. Ce langage est basé sur la notion d'objet, ainsi il est possible de construire de nouveaux parsers par héritage.

A partir de la description textuelle des analyseurs (lexical,syntaxique ou parcoureur d'arbre), l'environnement ANTLR est capable de générer le code cible d'un outil implantant cet analyseur. A ce jour, il peut générer du code source pour deux langages cibles suivants : le JAVA et le C++. Ce choix a un impact direct sur la grammaire car le langage d'action, utilisé dans le corps des règles, est celui du langage cible. ANTLR dispose de fonctions de documentation compatibles avec *javadoc* et il permet la génération d'une version html de l'analyseur.

La *ré-entrance* est une caractéristique particulière des parseurs générés par ANTLR. Le terme *ré-entrance* signifie que dans le cadre d'une analyse, il est possible de rappeler n'importe quelle règle du parser dans un bloc d'action. En fait, chaque règle de la grammaire est associée à une méthode, cela permet la *ré-entrance*.

4 Another's B Tools

4.1 Description de l'environnement

La figure 3 décrit ce que devrait être les grandes fonctionnalités de l'environnement ABTools.

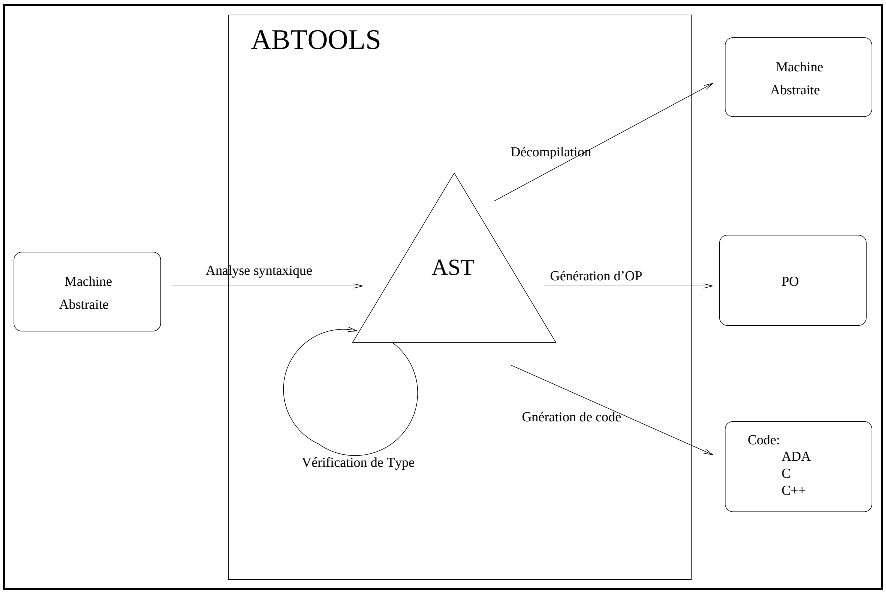

FIG. 3 – L'environnement ABTools

Afin de garantir la portabilité de notre environnement, nous avons décidé d'opter pour le développement d'une application JAVA. JAVA permettra la mise en place d'une interface graphique et le passage à une application WEB au travers de la mise en place d'une *APPLET*. En cas de problème d'efficacité, nous avons toujours la possibilité de compiler notre application avec *gcj*[6].

4.2 Analyse lexicale et syntaxique

4.2.1 Présentation

Les analyseurs lexical et syntaxique sont définis au travers d'une grammaire LALR(k). Les grammaires du langage B que nous avons implantées

[6]gcj est une application incluse dans GCC qui permet de passer d'un coe JAVA à un code C

sont du type LALR(k). Pour les premières, nous avions $k = 10$, cela était du à un grand nombre de conflits syntaxiques, maintenant nous avons $k = 2$.

Comme nous l'avions démontré dans [6], l'écriture d'ue grammaire pour la méthode B , doit nous amener à régler certains problèmes syntaxiques.

A titre d'exemple, pour réaliser une analyse syntaxique de composant B complexe ; nous avons fait le choix de considérer que les DEFINITIONS étaient des objets syntaxiques à part entière, cela nous a amené à induire certaines limitations sur ce qu'il est possible d'écrire dans une définition.

Plus généralement, ces travaux nous obligent à formaliser certains aspects du langage B qui restent parfois imprécis.

4.2.2 Caractéristique de notre grammaire

Notre grammaire définit un parser qui est une extension du parser de base d'ANTLR, comme le montre l'extrait de code suivant. Le parser construit export un vocabulaire intitulé **B**, il produit un arbre syntaxique dont chaque noeud est un objet de la classe **MyNode**. La classe **MyNode** permet de gérer une structure de donnée plus complexe et fournit les services associés.

```
class BParser extends Parser;
options
{
 exportVocab                = B;      // Call its vocabulary "B"
 k                          = 2;      // k tokens lookahead
 codeGenMakeSwitchThreshold = 2;      // Some optimizations
 codeGenBitsetTestThreshold = 3;
 buildAST                   = true;
 ASTLabelType               = "MyNode";
}
```

Comme le montre l'extrait de code suivant, notre grammaire permet de prendre en compte les trois types de machine abstraîte et de traiter les fchiers vides. Dans le cas de fichier vide, une erreur syntaxique est générée.

```
composant :
     machine
|    refinement
|    implementation
|    /* Empty source files are *not* allowed.  */
{
 errors.WSyntaxic   ( "B.g", "The file is empty" );
};
```

Afin d'obtenir une grammaire $LALR(2)$, il nous a fallu utiliser toutes les possibilités d'ANTLR. La meilleure façon de régler les problèmes liés à la syntaxe consiste à conditionner les règles problèmatiques.

Dans le cadre des clauses de visibilité entre machines abstraîtes, il est possible de renommer ou pas une machine. La règle précédente gère les deux cas. Il faut remarquer que dans la partie action de la conditionnelle, nous introduisons la gestion de la table des machines abstraites.

```
nameRenamedWithSave[String type]        :
 (B_IDENTIFIER B_POINT)
    =>
 B_IDENTIFIER (B_POINT^ nameRenamedWithSave[type] )
|
 t1:B_IDENTIFIER
{
 add_AM((MyNode)#t1,type);
};
```

Le générateur de token (lexer) est défini comme une extension du lexer de base. Conformément au langage B , dans la description de ce lexer nous faisons mention au fait qu'il y a une sensibilité à la "casse" des identificateurs. La caractéristique k vaut 5, cette valeur est directement liée à la plus longue sous-chaîne commune aux mot-clés du langage B .

```
class BLexer extends Lexer;
options
{
 exportVocab           = B    ; // Call its vocabulary "B"
 caseSensitive         = true ; // In B, the case is signifiant
 caseSensitiveLiterals = true ; // In B, the case is signifiant
 testLiterals          = true ; // automatically test for literals
 k                     = 5    ; // k characters of lookahead
}
```

Chaque mot-clés du langage B peut être entré explicitement au niveau de la description du lexer comme le montre l'extrait de code suivant, ou implicitement au niveau du parser.

```
B_PARTIAL               : "+->" ;
B_RELATION              : "<->" ;
B_TOTAL                 : "-->" ;
B_PARTIAL_INJECT        : ">+>" ;
B_TOTAL_INJECT          : ">->" ;
B_PARTIAL_SURJECT       : "+->>";
B_TOTAL_SURJECT         : "-->>";
B_BIJECTION             : ">->>";
```

Les analyseurs lexical et syntaxique décrivant la méthode *B* sont définis au travers d'une grammaire qui se trouve dans le fichier *B.g*.

Afin de traiter les liens de visibilité, nous avons introduit une table qui a pour but de gérer les machines abstraîtes vu par une machine donnée ou définit dans le cadre d'un projet. La mise en place de cette table et la lecture de toutes les dépendances est réalisée sur demande de l'utilisateur (switch -loadLinked).

4.2.3 Non déterminisme

Comme nous l'avions indiqué dans [6], plusieurs tokens sont utilisés dans des contextes différents (voir la table 4).

Symbole	les différentes utilisations	langage
« ; »	le séquencement de substitutions la composition de relations la séparation dans les listes d'ensemble ou de définition	Substitution Expression Clause ...
« = »	le test d'égalité dans une condition la définition d'ensemble une valuation le début du corps d'une opération	Predicate Expression Clause
« : »	l'appartenance le devient tel que la valuation de champ de record	Predicate Substitution Expression
« || »	la composition parallèle de substitutions la composition parallèle de relations	Substitution Expression
« - »	la différence entre deux expressions arithmétiques l'opérateur unaire de signe négatif la différence entre deux expressions d'ensemble	Expression Expression Expression

FIG. 4 – Recensement des surcharges d'opérateurs dans le langage *B*

Les règles de décision permettent de revenir à une situation maitrisée, mais l'arbre syntaxique contiendra toujours ces ambiguités. ANTLR permet de renommer les tokens lors de la construction de l'arbre. A titre d'exemple, la règle suivante reconnait une opération *B* et effectue un renommage sur le token = (B_EQUAL).

```
operation_Mch   :
operationHeader
```

```
  c:B_EQUAL^
{
        #c.setType(OP_DEF);
}
  substitution_Mch;
```

Comme précisé en 4.2.1, afin rendre déterministe le parsing nous avons du réaliser une restriction sur a notion de définition.

```
formalText_Mch  :
        expression
|       substitution_Mch
|       operation_Mch
```

Cette définition introduit la notion de niveau, le langage utilisé dans une machine n'est pas le même que celui d'un raffinement ou d'une implantation.

4.3 Manipulation d'arbre

Le parcours et la manipulation des arbres syntaxiques, avec ANTLR, sont réalisés au travers de grammaires qui sont appelées *TreeWalker*.

Le parcours d'arbre est réalisé en seconde phase, il n'y a plus lieu de vérifier la conformité de l'AST en entrée, c'est pourquoi nos grammaire ne s'intéresse qu'au parcours de l'arbre.

4.3.1 Décompilation d'arbre syntaxique

La décompilation est réalisée au travers d'un `TreeWalker`.

```
class TreeWalker extends TreeParser;
 options
 {
        importVocab     = B;
        buildAST        = false;
        ASTLabelType    = "MyNode";
        codeGenMakeSwitchThreshold = 3;
        codeGenBitsetTestThreshold = 4;
        k = 1;
 }
```

Nous avons utilisé la technologie objet pour ne réaliser qu'un seul `TreeWalker`. Actuellement, il permet de générer une version ASCII de l'AST, mais une version XML est en cour de développement.

Les règles suivantes montre que le `TreeWalker` est avant tout un ensemble de règles permettant le parcours de l'AST, des portions de code peuvent être associés à ces règles au travers de blocs d'actions.

```
machine :
#("MACHINE"
{
        index.Add();
        printToStringln(out.Clause("MACHINE"));
}
paramName
clauses
{
        printToString(out.Clause("END"));
}
);
```

Notre *TreeWalker* de décompilation est décrit au sein du fichier *Treewalker.g*.

4.3.2 Vérification du Typage

La phase de *Vérification du typage* s'applique à un AST, elle permet de vérifier la bonne utilisation des types. Cette phase permet de compléter l'AST avec des informations de types.

La figure 5 présente l'arborescence des types B que nous avons implantés. Chaque constructeur de type B est associé à une classe, l'arborescence est modélisée par un graphe d'héritage.

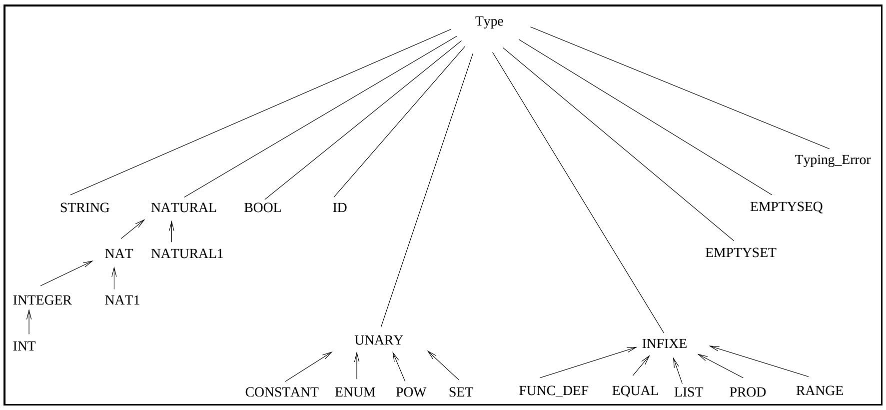

FIG. 5 – L'arborescence des types

Le vérificateur de type est codé au travers d'un *Treewalker*, intitulé *Typing.g*, comme le montre l'entête suivant.

```
class Typing extends TreeParser;
options
```

```
{
        importVocab     = B;
        buildAST        = false;
        ASTLabelType    = "MyNode";
        codeGenMakeSwitchThreshold = 3;
        codeGenBitsetTestThreshold = 4;
        k = 1;
}
```

Le vérificateur de type parcours l'AST et réalise plusieurs actions :

- la déclaration des objets rencontrés (variables, ensembles, constantes, opérations), cette déclaration s'accompagne d'une vérification de l'unicité de chaque identificateur,
- la vérification du typage,
- l'inférence de type pour des cas très précis (clause `VAR IN END`)
- la vérification du typage de tous les éléments déclarés.

Toutes ces activités sont faites au travers de la partie action des règles.

```
set_interval_value:
#(tt:B_EQUAL
  a:B_IDENTIFIER
{
  Type newType = new Type();
  pushScope(#a.getText());
}
  newType =interval_declaration
{
  newtype.setLineNumber(#tt.getLineNum());
  #tt.setBType(newtype);
  addId(a,newType);
  popScope();
}
);
```

Le type peut être calculé localement ou faire l'objet d'un calcul dans une autre règle, nous utilisons alors les règles avec retour de paramètre pour transférer les types. Afin de pouvoir tracer au mieux les anomalies, nous propageons le numéro de ligne affecté au token au type lui-même.

La prise en compte d'un nouvel identificateur est réalisé au travers de la gestion du contexte de définition, celui-ci est géré comme une pile, voir les opérateurs *pushScope(id)* et *popScope()*. Tout au long du processus de vérification du typage, une table des symboles annexe est mise à jour, voir l'instruction *addId(id,type)*.

```
MACHINE        any
CONSTANTS      xx
PROPERTIES     xx : INT
INITIALISATION
      ANY    xx
      WHERE  xx : INT & xx = 0
      THEN   skip
      END
END
```

Afin d'illustrer notre propos, le passage de la machine abstraîte précédente [7] au sein de notre environnement (commande *ABTOOLS -symbolTable any.mch*) produit la table des symboles suivante :

```
list of variable  :
key  any::ANY::xx Element xx(INT)
key  any Element any(FUNC_DEF(CONSTANT(Not Defined),Not Defined))
key  any::xx Element xx(INT)
```

Le nom de la machine fait lui-même partie intégrante de la table des symboles, son type `FUNC_DEF(CONSTANT(Not Defined),Not Defined)` signifie que c'est une fonction sans paramètre. De même, l'identificateur `xx` est défini dans deux contextes distincts.

Nous profitons du fait qu'ANTLR permet d'avoir des grammaires réentrantes pour vérifier que tout objet déclaré est typé. Il y a deux phases, la première permet de déclarer tous les objets (constantes et variables), la seconde permet de récupérer les informations de typage et de déceler les objets (constantes et variables) non typés et/ou non utilisés.

A titre d'exemple, ci-dessus ce trouve un extrait de la règle properties. La partie action vérifie le typage de chaque constante (abstraîte ou concrète).

```
properties      :
#("PROPERTIES"
{
        Type type = new Type();
}
 type = tt:predicate
{
 typeControleTreatement (tt, type);

 if (constant != null) list_var_bis(constant);
 if (abstract_constant != null) list_var_bis(abstract_constant);
 if (visible_constant != null) list_var_bis(visible_constant);
```

[7] Le bien fondé de cette machine abstraîte n'est pas discuté ici

```
 if (hidden_constant != null) list_var_bis(hidden_constant);
 if (concrete_constant != null) list_var_bis(concrete_constant);
}
);
```

4.4 Génération des Obligations de preuves

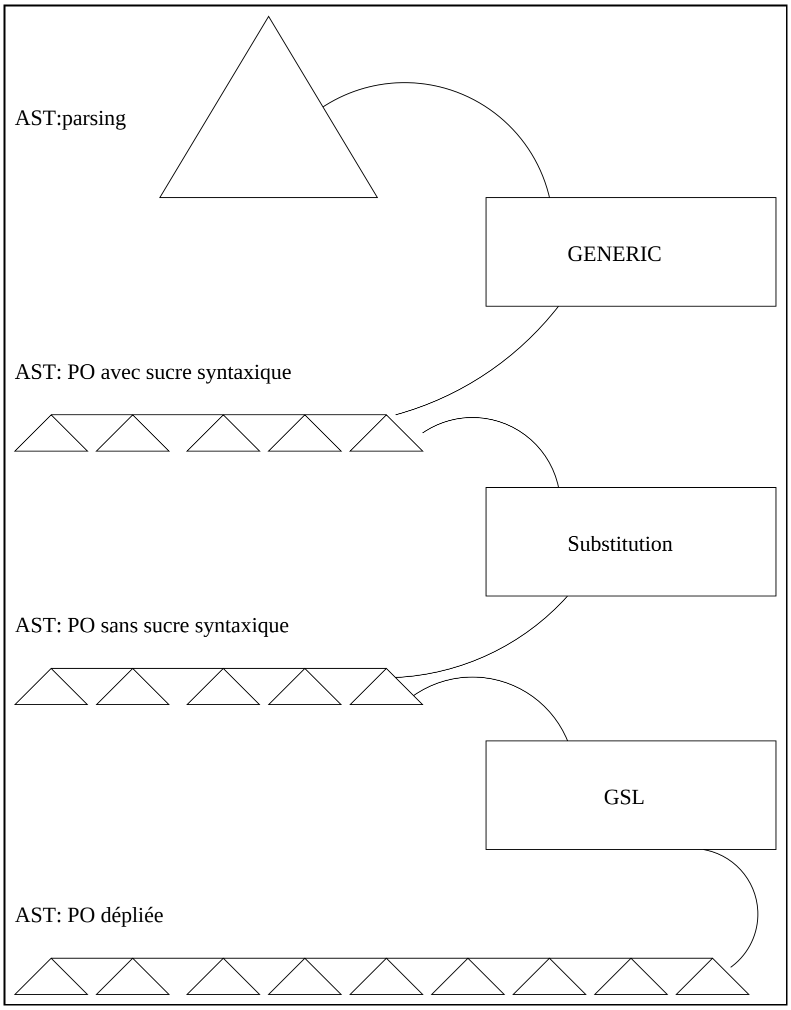

FIG. 6 – La génération des PO en trois étapes

Comme le montre la figure 6, la génération d'obligation de preuve (noté PO) est réalisée en trois temps. A chaque étape est associée un *treewalker*, chacun introduisant son propre vocabulaire.

– construction d'un nouvel arbre représentant l'obligation de preuve théorique,
– remplacement du sucre syntaxique par les substitutions de base,

– réduction de cet arbre par application des substitutions.

Dans le cadre de la seconde phase, le sucre syntaxique introduit par JR-Abrial pour définir le langage B est retiré afin de revenir au langage des substitutions généralisées. Ci-dessus la règle *goal* rappele que l'on passe de $[INSTRUCTION]P$ à $[GSL]P$.

```
goal! :
#(SUBST_TO ii:instruction pp:predicate
 {
  #goal = #(SUBST_TO, #pp,  #ii);
 }
);
```

BEGIN S END	S
PRE Q THEN S END	P | S
ANY z WHERE P THEN S END	@ z.(P=>S)
VAR x IN S END	@ x.S

TAB. 2 – Le sucre syntaxique

L'équivalence entre un sous-ensemble de substitutions utilisées dans le langage B et les substitutions généralisées est présenté au sein de la table 2. Le codage de cette équivalence est présenté dans l'extrait ci-dessous de la grammaire *substitution.g*.

```
instruction     :
  #("skip")
|
 ! #("BEGIN" i4:instruction
     {
      #instruction = #i4;
     }
    )
|
 ! #("PRE" p5:predicate i5:instruction
     {
      #instruction = #(GSL_SUCH, p5, i5);
     }
    )
|
 ! #("ANY" l10:listTypedIdentifier p10:predicate i10:instruction
     {
      #instruction = #(GSL_FOR_SUCH,l10, #(GSL_GUARD, p10,i10));
     }
```

```
    )
|
 ! #("VAR" l14:listTypedIdentifier i14:instruction
     {
      #instruction = #(GSL_FOR_SUCH,l14,i14);
     }
    )
```

Une fois les POs normalisées, il est possible de les dépliées. Cette étape est basée sur la sémantique des substitutions généralisées. Le tableau 3 présente les principales substitutions et la sémantique assciée.

$[skip]R$	R
$[S \| T](P \wedge Q)$	$[S]P \wedge [T]Q$
$[P \mid S]R$	$P \wedge [S]R$
$[S[]T]R$	$[S]R \wedge [T]R$
$[P \implies S]R$	$(P \implies [S]R)$
$[@x.S]R$	$\forall z.[S]R$
$[x, y := E, F]R$	$[x := E \| y := F]R$

TAB. 3 – Sémantique des substitutions généralisées

L'extrait de code suivant, met en évidence l'équivalence qui existe entre les règles formelles et la traduction sous forme de grammaire.

L'ensemble des substitutions est modélisé sous forme d'une seule règle intitulé *gsl*, qui est constitué de plusieures alternatives.

```
gsl [MyNode pr]  returns [MyNode result]
{
        MyNode res ;
        result = new MyNode();
}           :

 !#(GSL_SUCH      p1:predicate   res=gsl[pr]
  {
        result  = #(B_AND, #p1, #res);
  })
|
 !#(GSL_FOR_SUCH  l1:listTypedIdentifier   res=gsl[pr]
  {
        result  = #(B_FORALL, #l1, #res);
  })
|
 !#(GSL_GUARD     p2:predicate  res=gsl[pr]
  {
```

```
        result  = #(B_IMPLIES, #p2, #res);
  })
|
 !#("skip"
  {
        result  = pr;
  })
```

4.5 Structure de l'environnement

L'environnement ABTools est actuellement décomposé en quatre éléments : l'anlyseur lexical, l'analyseur syntaxique, le décompilateur et le vérificateur de type.

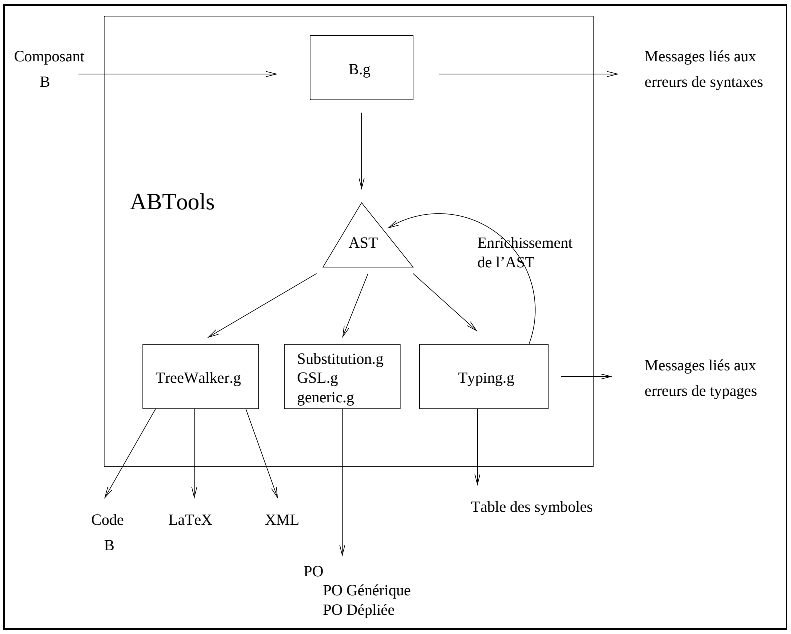

FIG. 7 – L'environnement ABTools

5 Évolutivité

Plusieurs travaux présentent des extensions intéressantes de la méthode B , afin de valider celles-ci, il est intéressant de pouvoir rapidement et facilement les implanter.

L'utilisation d'ANTLR permet d'atteindre facilement cet objectif car les grammaires peuvent être étendues par héritage. La construction d'une nouvelle grammaire se fait par surcharge de certaines règles et par ajout de règles complémentaires. Comme premier exemple d'évolutivité, nous avons implanté certaines extensions que nous avions proposées dans [6]. Dans ce papier, nous décrivions une version améliorée du langage B que nous appelions «Bprime».

5.1 B Prime

Cette version du langage B permet de typer les variables lors de leur déclaration. La figure 8 introduit un exemple où l'on a retiré de l'invariant l'information de typage, cela permet de réserver la clause INVARIANT pour des propriétés plus importantes ou plus critiques.

VARIABLES
    N1, N2, N3
INVARIANT
    $N1 \in \mathrm{NAT} \wedge$
    $N2 \in \mathrm{NAT} \wedge$
    $N1 > N2 \wedge$
    $N3 \in \mathrm{NAT} \wedge$
    $(N1 + N2) > N3$
	VARIABLES
	    $N1 \in \mathrm{NAT} \wedge$
	    $N2 \in \mathrm{NAT} \wedge$
	    $N3 \in \mathrm{NAT}$
	INVARIANT
	    $N1 > N2 \wedge$
	    $(N1 + N2) > N3$

FIG. 8 – Typage explicite

La mise en place de cette extension au sein de notre environnement, se traduit par la mise en place d'une nouvelle grammaire nommée *BPrime.g*. Cette grammaire hérite de la grammaire décrite précédemment et surcharge certaines règles. Elle est de la forme :

```
class BPrimeParser extends BParser;
options
{
   exportVocab=BPrime;
   k=10 ;
   buildAST    = true;
   ASTLabelType        = "MyNode";
}

variables           : (      "VARIABLES"^
                           | "ABSTRACT_VARIABLES"^
                           | "VISIBLE_VARIABLES"^
                      )
                      listTypedIdentifier ;

listTypedIdentifier : typedIdentifier (B_COMMA^ typedIdentifier)* ;

typedIdentifier     : nameRenamed (B_INSET^ cbases)? ;
```

Cette nouvelle grammaire hérite de l'ensemble des règles liées au parser *BParser*, ce parser est défini par la grammaire *B.g.* Elle surcharge la règle *VARIABLES* et elle introduit les deux nouvelles règles *listTypedIdentifier* et *typedIdentifier*.

Il faut reporter cette évolution aux seins des autres outils (décompilateur et vérificateur de type). Pour les outils manipulant les arbres syntaxiques (treewalker), il n'est pas utile de toujours séparer les grammaires, en effet ces outils supposent que les arbres abstraits sont corrects.

En fait, le langage *BPrime* étend cette notion de typage implicite aux clauses *CONSTANTS* (abstaites ou non), aux profils d'opérations et aux substitutions *ANY*, *VAR* et *WHILE*.

Le langage *BPrime* introduit deux autres évolutions :
- La prise en compte de commentaire "à la C++",
- La prise en compte du caractère "," comme séparateur d'ensemble dans la clause *SETS*.

La première modification impacte le *LEXER* au travers de la règle :

```
CPPComment
   options {
             paraphrase = "a C++ comment";
           }:
                "//"
                ( ~('\n') )*
                {_ttype = Token.SKIP;}
           ;
```

Cette règle indique que tout caractère suivant le token \\ est ignoré jusqu'au caractère fin de ligne.

La figure 9 présente un exemble de machine abstraîte décrite dans le langage *BPrime*.

5.2 B Système

Dans [3], Jean-Raymond Abrial a montré qu'il était possible d'utiliser la méthode *B* pour spécifier des systèmes distribués. Mais c'est dans [1] qu'il propose une véritable extension de la méthode *B* à la notion d'événement et de contraintes dynamiques.

Cette extension introduit de nouvelles clauses (*SYSTEM*, *EVENTS*, *DYNAMICS*, *VARIANT* et *MODALITIES*), de nouvelles substitutions et un nouveau type de prédicat.

```
// AUTHOR : Boulange Jean-Louis
/* Exemple de machine decrite sous BPRIME */

MACHINE     Exemple_BPrime

VARIABLES   aa : NAT , bb : NAT  // Typage explicite des variables

INVARIANT   aa < bb  // L'invariant ne contient que les proprietes

OPERATIONS
            r1 : NAT, r2 : NAT <-- EX ( p1 : NAT, p2 : NAT) =
            PRE
                 p1 < p2
            THEN
                 ANY
                      a1 : NAT, a2 : NAT
                  WHERE
                      a1 < a2
                   &  a1 < p1
                   &  a2 < p2
                  THEN
                      r1, r2 := a1, a2
                  END
            END
END
```

Fig. 9 – Typage explicite

La figure 5.2 présente un exemple qui est repris de [1].

```
SYSTEM           toy_with_scheduler_dynamics_and_modality
VARIABLES        xx, yy, cc, dd
INVARIANT        xx, yy, cc, dd : NA*NAT*NAT*NAT
             &  (cc >0 or dd>0)
DYNAMICS         xx <= xx' & yy <= yy'
INITIALISATION   xx,yy := 0,0 || cc,dd :: NAT1*NAT1
EVENTS
evt_xx = SELECT  cc>0 THEN xx,cc:=xx+1,cc-1 || dd::NAT1 END;
evt_yy = SELECT  dd>0 THEN yy,dd:=yy+1,dd-1 || cc::NAT1 END

MODALITIES
SELECT  cc > 0 LEADSTO cc = 0 WHILE evt_xx VARIANT cc END;
SELECT  dd > 0 LEADSTO dd = 0 WHILE evt_yy VARIANT dd END
END
```

FIG. 10 – Un *System*

6 Conclusions

Cette première étape de réalisation a permis de montrer la puissance de l'environnement de génération de compilateur nommé ANTLR. Nous avons pu dans un premier temps réaliser une première version de notre environnement B et dans un deuxième temps nous avons pu tester certaines extensions.

Les parseurs (B , BPRIME et BSystèe) de l'environnement *ABTools* sont décrit par une grammaire $LALR(2)$. Les autres outils sont décrit par une grammaire $LALR(1)$

Actuellement, nous sommes en phase de finalisation du *vérificateur de type* et nous avons entammé les travaux liés à la génération des obligations de preuve. Pour la génération de code, nous attendrons la fiabilisation des travaux en cour. La génération de code pourra être réalisée, après formalisation du passage $B0$ vers le(s) langage(s) cible(s), par adaptation du décompilateur.

Aucune difficulté majeure n'a été rencontrée et l'utilisation de l'environnement ANTLR pour la construction de notre environnement B semble un succès.

L'ensemble des travaux décrits dans ce papier est disponible sur le site *http ://www.chez.com/abtools*.

Références

[1] J-R Abrial and L Mussat. Introducig dynamic constraints in b.

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

[3] Jean-Raymond Abrial. Extending B without changing it (for developing distributed systems). In Henri Habrias, editor, *Proceedingsof 1st Conference on the B method*, Putting into Practice methods and tools for information system design, pages 169–191, 3 rue du Maréchal Joffre, BP 34103, 44041 Nantes Cedex 1, November 1996. IRIN Institut de recherche en informatique de Nantes.

[4] E.W. Dijkstra. *A Disciple of Programming.* Prentice Hall, 1976.

[5] Mariano Georges. *Évaluation de logiciels critiques développés par la méthode B : une approche quantitative.* PhD thesis, Universitée de Valenciennes et du Hainaut-Cambrésis, Dec 1997.

[6] Boulanger Jean-Louis, Mariano George, and Tatibouet Bruno. Revisiting b language syntax. Technical Report 99-07, CNAM Laboratoire CEDRIC, 1999.

[7] Dick Jeremy. AMN notation. Technical report, BCore Ltd, April 1995.

[8] Cliff B. Jones. *Systematic Software Development Using VDM.* Prentice-Hall International, Englewood Cliffs, New Jersey, second edition, 1990. ISBN 0-13-880733-7.

[9] Georges Mariano. Une grammaire BNF pour le formalisme B - v0.3. Technical Report 96-24, INRETS–ESTAS, Septembre 1996.

[10] C. Morgan. *Programming from Specifications.* International Series in Computer Science. Prentice Hall, 1990.

[11] Michael Schmitt. The development of a parser for sdl-2000. Technical report, Institute for Telematics, Medical University of Lubeck, Ratzeburger Allee 160 23538 Lubeck Germany, 1998.

[12] J. M. Spivey. *The Z Notation : A Reference Manual.* Prentice Hall International Series in Computer Science, 2nd edition, 1992.

[13] Stéria. Le langage B : Manuel de référence. Technical Report V 1.8, RATP SNCF INRETS, June 1998.