Vers un système de modules à la Harper-Lillibridge-Leroy pour les spécifications formelles *B*

Dorian Petit[1,2] & Georges Mariano[1] & Vincent Poirriez[2]

1: INRETS/ESTAS,
20, rue Élisée Reclus
59650 Villeneuve d'Ascq, France
{dorian.petit,georges.mariano}@inrets.fr
2: Université de Valenciennes et du Hainaut Cambrésis
59313 Valenciennes Cedex 9, France,
{vincent.poirriez}@univ-valenciennes.fr

Résumé

Nous nous proposons ici de réutiliser le calcul des modules présenté dans [Leroy00] et de l'appliquer au calcul des modules du langage support de la méthode *B*.

En cherchant à exprimer la modularité de *B* dans un calcul des modules «à la Harper-Lillibridge-Leroy» nous cherchons à obtenir une modélisation claire de celle-ci et *simultanément* un outillage. Cette étude étant expérimentale, nous nous limiterons à une partie du langage *B* : les composant dits «d'implémentation».

1. Introduction

Le langage *B* est le langage développé conjointement à la méthode de spécification formelle *B*. Cette méthode a été développée durant les années 80 par J.-R. Abrial ([Abrial96]), qui était déjà à l'origine de la méthode *Z* ([Spivey98]). L'avantage essentiel de cette méthode, notamment par rapport à *Z*, est la possibilité de générer le code compilable correspondant au logiciel spécifié. Cette méthode a bénéficié de l'effort conjoint de plusieurs industriels français qui ont participé à son développement et lui ont permis d'atteindre un niveau de maturité qui fait de cette méthode l'une des rares qui soit utilisable à l'échelle industrielle ([Behm et al.97], [Lanet et al.98], [Behm et al.99], etc).

Si, initialement, le domaine d'application de la méthode est limité au domaine ferroviaire, désormais, *B* s'ouvre à d'autres horizons comme les protocoles d'authentification ([Bieber et al.94]), les protocoles de communication ([Abrial et al.97]), les bases de données ([Laleau et al.00]) et les applications embarquées type cartes à puce ([Lanet et al.98], [Sabatier et al.99]). Cette ouverture vers divers domaines ne peut se faire sans certaines adaptations de la méthode. Ainsi, le développement d'applications destinées aux systèmes embarqués sur les cartes à puce nécessite une rigueur qui rend l'utilisation des méthodes formelles obligatoire (d'après les critères communs [CC-99]). Ce genre d'applications est confronté à des problèmes de ressources limitées, et quellle que soit la méthode formelle utilisée, il est impératif que le produit final (le code généré) tienne compte de ces contraintes. La maîtrise du processus de génération de code(s) est donc essentielle. Dans tout langage, le mécanisme de génération de code est intimement lié aux aspects de modularité, c'est pourquoi notre objectif de génération maîtrisée du code passe par l'étude de la modularité. Cet article portera sur ces travaux préliminaires, nous n'aborderons pas les aspects génération de code, nous ne cherchons ici qu'à modéliser le système de module de B.

La modularité est un concept essentiel utilisé dans les langages informatiques. En général, le développement d'un langage s'accompagne du développement d'un système de modules qui lui est spécifique, et cela, bien que les notions intégrées aient de nombreux points communs avec d'autres systèmes existants.

Des travaux récents, [Ancona et al.98] et [Leroy00] mettent l'accent sur la distinction qu'il est possible de faire entre le langage de base[1] et la partie du langage consacrée à l'expression de la modularité. [Leroy00] décrit même un langage de modules proche de celui de SML ([Macqueen86]) indépendant du langage de base.

Des exemples de réutilisation du système de modules à la SML sont donnés dans [Leroy00] : application à Prolog [Sannella et al.92] et à SIGNAL [Nowak et al.97]. Dans ces travaux, les langages étudiés sont, à l'origine, dépourvus de modularité, il s'agit donc d'en ajouter. L'idée générale est alors de «greffer» un système de modules dont la sémantique est parfaitement connue sur un langage quasi quelconque.

L'objectif de nos travaux est différent, nous cherchons à clarifier la sémantique d'un système de modules déjà existant en l'exprimant au moyen d'un autre. [Watson97] juge que la prolifération des constructions en B (les constructions de modularité) et des règles qui leurs sont associées sont rebutantes (10 pages de [Abrial96] sont consacrées aux tables de visibilité).

Les difficultés des travaux que nous menons sont de deux ordres. D'une part, exprimer la modularité de B dans un autre système ; la complexité du système de modules de B ne facilite pas le plongement de ce système dans un autre. D'autre part, il faut que cette reformulation s'efforce de clarifier la sémantique associée à la modularité de B.

La présente description de nos travaux s'organise comme suit. Dans la partie 2, nous présenterons brièvement le langage B afin de poser le contexte, et nous aborderons ensuite les aspects plus spécifiques à la modularité. Nous étudierons, en partie 3, le calcul des modules de Xavier Leroy ([Leroy00]) que nous utiliserons en partie 4 pour exprimer la modularité en B. Nous verrons ensuite les travaux en relation avec ce que nous présentons et ce que nous a apporté cette démarche et selon quelle(s) voie(s) nous pouvons orienter la suite de nos travaux.

2. Modularité en B

2.1. Généralités

Le langage B ([Abrial96]) est basé sur la théorie des ensembles pour exprimer les structures mathématiques manipulées, les substitutions généralisées pour les actions sur ces entités, et la logique axiomatique du premier ordre avec égalité pour définir les relations entre ces entités et pour exprimer les preuves de correction du développement (obligations de preuves).

Un *texte B* (ou *composant B*) est composé de clauses. Certaines sont utiles à définir les éléments constituant le composant, comme les variables, les ensembles, les propriétés, etc (clauses VARIABLES, SETS, etc). D'autres clauses sont qualifiées de clauses de liaisons : elles servent à exprimer les dépendances entre composants (clauses INCLUDES, IMPORTS, etc). Nous ne détaillerons pas ces diverses clauses, cela n'étant pas nécessaire à la compréhension générale de notre démarche.

Le système de modules fourni par la méthode B se base sur la notation en machines abstraites (AMN : Abstract Machine Notation) afin de décomposer le problème en modules et d'exprimer les dépendances entre les différents composants. C'est cet aspect du langage que nous allons détailler.

[1] Nous désignons par «langage de base» ou «langage basique» un langage dépourvu de toute construction de modularité. Pour insister sur cette caractérisation, nous utiliserons désormais l'expression «langage basique».

Sans nous étendre sur la méthodologie de développement d'un projet *B* ou même sur les aspects théoriques de la méthode, nous rappelons simplement que le développement d'un *module B* se fait par raffinements successifs. Au sein d'un module, à partir d'une machine, qui correspond à la spécification abstraite des services rendus par ce module, d'autres composants sont développés successivement (étapes de raffinement) jusqu'à un niveau de développement concret directement traduisible dans un langage informatique impératif classique (actuellement C, C++ ou Ada).

Nous appelons donc *module B* un ensemble de composants reliés entre eux par une relation de raffinement. Un composant est soit une machine (abstraite), soit un raffinement, soit une implantation.

2.2. Expression de la modularité en *B*

De manière classique, un module est constitué d'une interface définissant les différentes entités du module (accessibles ou non de l'extérieur), et d'un corps implantant ces entités.

La particularité du système de modules de *B* vient de la vision de ces modules de l'extérieur. Pour un module MA donné, l'accès à une entité de ce module MA à partir d'un module MB dépendra de l'endroit où, *dans le module MB*, elle est référencée. Cela revient à avoir plusieurs visions d'un même module. À chaque vision correspondra une interface différente pour le module. Suivant l'endroit (du module MB) où la référence à une entité du module MA sera faite, il faudra choisir l'interface du module MA adéquate.

La figure 1 correspond à un exemple (simplifié) de machines *B* non correctes. L'erreur vient de l'implantation `Nimporte` où dans la clause VALUES, une référence non autorisée à la variable `YY` de la machine `Mimportee` est faite. Par contre, dans la clause INITIALISATION, l'accès à cette même variable `YY` ne pose pas de problème.

```
MACHINE
        Mimportee
SETS XX
CONCRETE_VARIABLES
        YY
INVARIANT
        YY : INT
INITIALISATION
        YY := 10
END
```

```
MACHINE
        Nexemple
END
```

```
IMPLEMENTATION
        Nimporte
REFINES
        Nexemple
IMPORTS
        Mimportee
SETS
        TT
CONCRETE_VARIABLES
        ZZ
VALUES
        TT = 1..YY
INITIALISATION
        ZZ:=YY
INVARIANT
        ZZ : INT

END
```

FIG. 1 – Exemple d'accès interdit

Cette notion d'accessibilité n'étant pas intuitive, nous allons faire une analogie avec les relations d'héritage en C++. Nous considérerons l'héritage comme une relation de composition de classes, ce qui en toute rigueur n'est pas le cas.

Trois types d'entité peuvent être définies dans une classe C++ : `public`, `private` et `protected`. Ces types d'entité se rapprochent en B des clauses constituants un composant B.

Trois types d'héritage sont possibles[2] : `hpublic`, `hprivate` et `hprotected`. Ces différents types d'héritage sont à rapprocher des différentes clauses de liaisons en B.

En C++, le statut des entités dans une classe dérivée change en fonction de son type dans la classe de base et du type d'héritage (voir le tableau 1). Ainsi si une classe `B` héritait d'une classe `A` de manière `hprotected`, une entité `public` de la classe `A` aurait un statut `protected` dans la classe `B`.

	hpublic	**hprotected**	**hprivate**
public	public	protected	private
protected	protected	protected	private
private	inacessible	inacessible	inacessible

TAB. 1 – relation entité héritage

Pour "émuler" la modularité de B sur la base de ce qui existe en C++, il est nécessaire d'ajouter une troisième dimension au tableau 1. Cette dimension serait constituée du statut de l'entité qui fait référence à une autre provenant de la classe dont elle hérite.

Pour simuler les types de dépendances de B, nous différencions deux types de méthodes : des `exclusives` et des `nonexclusives`. Une méthode `exclusive` ne pourra accéder qu'aux entités de sa classe alors qu'une méthode `nonexclusive` pourra également accéder aux entités héritées.

Dans la suite, nous mettrons entre parenthèses ce qui se réfère à la figure 1. Si une classe `B` (l'implantation `Nimporte`) hérite d'une classe `A` (la machine `Mimportee`) de manière `hprotected` (le lien `IMPORTS`), une entité `public` de la classe `A` (la variable `YY`) qui est référencée dans une méthode `exclusive` (la clause `VALUES`) a un statut `inacessible` dans la classe `B`. Si elle est référencée dans une méthode `nonexclusive` (la clause `INITIALISATION`), elle a un statut `protected` (cet aspect protected n'existant pas en B, nous dirons simplement que la variable a un statut accessible) dans la classe `B`.

2.3. Cas des composants d'implantations

2.3.1. Restriction

Étant donnée l'étendue du système de modules de B, il nous est impossible de traiter ici l'intégralité du langage. Nous allons donc nous restreindre aux implantations, composants directement concernés par la génération de code.

2.3.2. Étude des implantations

Nous allons étudier les clauses d'implantation les plus caractéristiques : les clauses IMPORTS et SEES. Les solutions que nous apporterons pour traiter ces deux clauses suffiront à lever les problèmes susceptibles d'être rencontrés avec les autres clauses.

La clause IMPORTS permet d'exprimer la réalisation des traitements d'un module au moyen d'autres modules «importés».

[2]Nous préfixons par `h` volontairement les noms des héritages dans un souci de clarté.

La clause SEES permet le partage de données dans un projet. L'architecture des dépendances entre composants (ici implantations) peut être vue comme un arbre (figure 2), les liens IMPORTS formant les branches de cet arbre. La clause SEES permet d'établir un lien transversal de visibilité dans cette architecture de base.

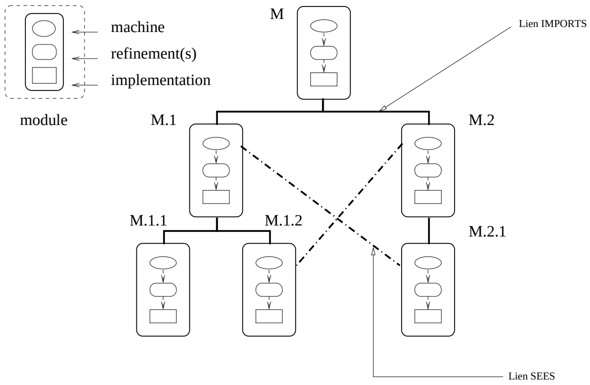

FIG. 2 – Structure d'un développement B

Comme nous l'avons vu en 2.2, la principale originalité du système de modules de B vient de la visibilité des diverses entités. En se basant sur le document de référence de la méthode, le BBook ([Abrial96]), il est possible d'extraire les tableaux 2 et 3 qui résument l'accessibilité des entités par rapport aux clauses d'une implantation.

Dans ces tableaux, RO correspond à un droit d'accès à une catégorie d'entité alors que X indique l'impossibilité d'y accéder. Notons que par droit d'accès à une entité, nous entendons droit d'accès en lecture uniquement. En effet, les machines B sont soumises au “hiding principle” qui consiste à systématiquement “passer” par une opération de la machine où est déclarée une variable pour modifier cette même variable.

Exemple : Le RO dans la case correspondante à la colonne VALUES et la ligne SETS, signifie que dans la clause VALUES de la machine qui importe, l'accès en lecture aux éléments de la clause SETS de la machine importée est autorisé.

Relatif au IMPORTS	IMPORTS	PROPERTIES	VALUES	INVARIANT/ ASSERTIONS
SETS	X	RO	RO	RO
CONCRETE_CONSTANTS	X	RO	RO	RO
ABSTRACT_CONSTANTS	X	RO	X	RO
CONCRETE_VARIABLES	X	X	X	RO
ABSTRACT_VARIABLES	X	X	X	RO
INVARIANT	X	X	X	X
OPERATIONS	X	X	X	X

TAB. 2 – Visibilité des entités d'une machine importée dans une implantation

À ces deux tableaux (2 et 3) manquent deux colonnes, celles concernant les clauses INITIALISATION et OPERATIONS. Elles ne sont pas intégrées dans les tableaux, car elles ont une particularité : elles font intervenir une nouvelle distinction (une quatrième dimension) pour le langage, les invariants de boucle. Dans la suite, nous ne tiendrons pas compte de cette possibilité d'accéder à certaines entités uniquement dans les invariants de boucle, et nous en interdirons l'accès.

Relatif au SEES	IMPORTS	PROPERTIES	VALUES	INVARIANT/ ASSERTIONS
SETS	RO	RO	RO	RO
CONCRETE_CONSTANTS	RO	RO	RO	RO
ABSTRACT_CONSTANTS	X	RO	X	RO
CONCRETE_VARIABLES	X	X	X	X
ABSTRACT_VARIABLES	X	X	X	X
INVARIANT	X	X	X	X
OPERATIONS	X	X	X	X

TAB. 3 – Visibilité des entités d'une machine vue dans une implantation

composant vu ou importé	composant qui importe	composant qui voit
SETS	RO	RO
CONCRETE_CONSTANTS	RO	RO
ABSTRACT_CONSTANTS	dans les invariants de boucle	dans les invariants de boucle
CONCRETE_VARIABLES	RO	RO
ABSTRACT_VARIABLES	dans les invariants de boucle	dans les invariants de boucle
INVARIANT	X	X
OPERATIONS	X	RO

TAB. 4 – Visibilité des entités : cas particuliers des clauses INITIALISATION et OPERATIONS

Remarque. Dans le cas de l'accès en lecture uniquement à une opération, cela signifie que seules les opérations ne modifiant pas les entités du composant sont accessibles (voir la dernière ligne du tableau 4).

Après cette présentation des spécificités des implantations, le paragraphe suivant sera consacré à la présentation du système de modules à la "Harper-Lillibridge-Leroy".

3. Présentation du système de modules «HLL»

Il existe divers systèmes de modules, chacun ayant leurs spécificités. Nous avons choisi le système de modules présenté dans [Leroy96] et dans [Leroy00], un système de modules «à la SML».

En effet, les systèmes de modules à la SML sont reconnus parmi les plus puissants. Le choix d'un système à la Harper-Lillibridge-Leroy (HLL) est suggéré par la relative simplicité de sa mise en oeuvre [Leroy00], intégrer un système de modules à un langage basique se résume à :
- déterminer la syntaxe abstraite du langage basique,
- déterminer la sémantique statique qui l'accompagne (typage).

En ce qui concerne notre démarche, l'intérêt essentiel du système HLL réside dans son indépendance vis à vis du langage cible. Il peut donc être «ajouté» à un langage pré-existant.

Nous ne faisons pas ici une présentation exhaustive du système HLL, mais nous nous plaçons du point de vue utilisateur du système, les détails sur l'implantation ou les fondements du système HLL pourront être trouvés dans [Leroy00].

Attention ! *La présentation du système de modules HLL est fortement liée à la terminologie utilisée dans les langages fonctionnels de la famille (S)ML*[3]*. Or cette présentation est (volontairement) de nature «constructive» car elle repose sur une implantation fonctionelle du système de modules. Le*

[3]Plus précisément, de même que X.Leroy, nous utilisons le langage Ocaml, «dialecte» de cette même famille

terme «module» étant une construction primitive du langage fonctionel utilisé, il est essentiel de ne pas confondre les «modules», concept que nous souhaitons ajouter à un langage basique, et les «modules» écrits en langage fonctionnel implantant la construction du système HLL. Dans cette partie, le terme «module» désigne un code fonctionnel.

3.1. Vision globale

Le système HLL utilise les foncteurs pour construire un vérificateur de sémantique statique. Nous rappelons qu'un foncteur est aux modules ce qu'est une fonction à une valeur : un foncteur permet de construire un module à partir d'un autre module qui lui est passé en paramètre.

L'utilisation du système HLL nécessite de fournir des modules définissant le langage basique. C'est par l'utilisation des foncteurs définis dans [Leroy00] que l'on peut construire l'intégralité du vérificateur de la sémantique statique.

Nous allons décrire dans les parties 3.2 et 3.3 la manière d'exprimer le langage que l'on veut munir des constructions modulaires. Dans un premier temps, nous allons donner une vision générale de ce qu'est le développement complet d'un vérificateur de type pour un langage quelconque.

Soient deux modules `A` et `ATyping` correspondant respectivement au langage basique A (A : un langage quelconque) et aux vérifications de types l'accompagnant.

Pour augmenter le langage basique des constructions de modularités du système HLL, il suffit de faire appel au foncteur `Mod_syntax` :

```
module AMod = Mod_syntax(A)
```

`Mod_syntax` est le foncteur essentiel qui transforme une syntaxe abstraite d'un langage en syntaxe abstraite de ce même langage muni de constructions modulaires.

Pour effectuer les vérifications de type associées au langage, il faut construire un environnement dans lequel se feront ces vérifications ; pour cela, il faut faire appel à `Env` :

```
module AEnv = Env(AMod)
```

Il est maintenant possible de construire le module qui effectuera les vérifications par appel au foncteur `Mod_Typing` :

```
module AModTyping = Mod_Typing(AMod)(AEnv)(ATyping)
```

Pour l'utilisation du système HLL, le travail se situe donc uniquement au niveau de la spécification du langage basique (spécification de la syntaxe abstraite et de la sémantique statique).

3.2. Syntaxe abstraite du langage basique

Pour utiliser le système HLL, il faut que la syntaxe abstraite du langage basique respecte la signature suivante :

```
module type CORE_SYNTAX =
  sig
    type term
    type val_type
    type def_type
    type kind
```

```
    val subst_valtype: val_type -> Subst.t -> val_type
    val subst_deftype: def_type -> Subst.t -> def_type
    val subst_kind: kind -> Subst.t -> kind
  end
```

Le type `term` correspond à la syntaxe abstraite des entités que l'on peut définir. Ces entités auront pour type `val_type` . `def_type` désigne les expressions de type qui peuvent être définies.

En ce qui concerne les trois fonctions `subst_valtype`, `subst_deftype` et `subst_kind`, le lecteur intéressé peut se référer à [Leroy00, parties 2.3 et 2.4] pour plus de détails. Nous nous contentons de signaler que ces trois fonctions sont indispensables suite à l'utilisation de chemins d'accès pour référencer les entités (fonctions, type, etc) ; l'utilisation des chemins d'accès est préférable aux noms pour des raisons d'unicité des entités.

3.3. Vérifications statiques

Après avoir fournie la syntaxe basique du langage, il faut fournir le module de vérification de la sémantique statique ; ce module doit suivre la signature suivante :

```
module type CORE_TYPING =
  sig
    module Core: CORE_SYNTAX
    module Env: ENV with module Mod.Core = Core
(* Typing functions *)
    val type_term: Env.t -> Core.term -> Core.val_type
    val kind_deftype: Env.t -> Core.def_type -> Core.kind
    val check_valtype: Env.t -> Core.val_type -> unit
    val check_kind: Env.t -> Core.kind -> unit
(* Type matching functions *)
    val valtype_match: Env.t -> Core.val_type -> Core.val_type -> bool
    val deftype_equiv:
          Env.t -> Core.kind -> Core.def_type -> Core.def_type -> bool
    val kind_match: Env.t -> Core.kind -> Core.kind -> bool
    val deftype_of_path: path -> Core.kind -> Core.def_type
  end
```

Le module `Env` garde en mémoire l'environnement nécessaire pour effectuer les vérifications. La fonction `type_term` type un terme dans un environnement donné. `check_valtype` vérifie la bonne formation d'un type. `valtype_match`, `deftype_equiv` et `kind_match` testent la compatibilité de deux types ou deux kinds.

3.4. Système de modules obtenu

L'instanciation de [Leroy00] construit un langage modulaire "au dessus" d'un langage de base. Ces modules peuvent être paramêtrés par d'autres modules, pour ainsi obtenir des foncteurs. Chaque module de ce langage est constitué de modules, de types ou de valeurs.

À un module est associée une signature qui permet de "cacher" certaines entités du module. Les définitions de type (ou de modules) peuvent être abstraites : c'est à dire que de l'extérieur du module seule l'existence du type (ou du module) peut être connue. Certains composants du module peuvent être masqués pour ne fournir comme services (ce qui sera utilisable de l'extérieur) que ce qui est nécessaire.

Deux contraintes sont imposées pour l'utilisation de HLL qui nous intéresse. La première est la limitation du langage de base à deux catégories syntaxiques : type et valeurs. Une troisième catégorie, les modules, est ajoutée lors de la construction du langage modulaire. La seconde contrainte impose que la définition d'une entité appartenant à l'une de ces trois catégories, n'entraine pas la définition implicite d'une entité d'une autre catégorie. Par exemple, en OCaml, la définition d'un type somme s'accompagne de la définition des constructeurs constituant ce type.

4. Système de modules pour les implantations B

Dans cette partie, nous allons aborder l'adaptation du système HLL pour B. Nous commencerons par suivre les deux étapes vues en partie 3, la définition de la syntaxe abstraite (langage basique pour B) puis la sémantique statique associée.

4.1. Syntaxe abstraite

Nous proposons ci dessous une syntaxe abstraite pour le langage de base que nous considérons. Nous ne présenterons pas la totalité de la syntaxe. La présentation réduite qui en est faite ici ne sert qu'à présenter l'aspect global du langage et certaines spécificités que nous détaillerons.

```
module B = struct
  type ids = Ident.t list
  type predicate = ...                           (* Predicats du langage B     *)
  and expr = ...                                 (* Expressions du langage B   *)
  and substitution = ...                         (* Substitutions du langage B *)
  type btype =
      Z | Bool | String                            (* Types de base B            *)
    | Power of btype | Prod of btype * btype       (* Types <<construits>> B     *)
    | TypeIdentifier of path                       (* <<Given sets>>             *)

  type term =
                              (* Clauses constitutives d'un composant B *)
      Invariant of expr
    | ConstantConcrete of btype * expr
    | ConstantAbstract of btype
    | Properties of predicate
    | VariablesConcrete of btype * substitution
    | VariablesAbstract of btype * substitution
    | Assertions of predicate
    | Operations of ids * ids * substitution

  type val_type = btype

  type def_type =
      SetAbstract of expr option         (* Ensembles abstraits destinés à être valués *)
    | SetEnumerate of ids                (* Ensembles enumérés  *)

  type kind = unit

  let subst_valtype vty subst = ...
  let subst_deftype dty subst = ...
end
```

Les deux catégories syntaxiques qu'il est possible de définir dans le langage de base, sont les types et les valeurs. Ici les types définissables sont les ensembles enumérés ou abstraits et les valeurs correspondent "au reste" (variables, constantes, opérations, ...). Contrairement à ce que nous avons vu en partie 3.4 la définition d'un nouveau type (en l'occurence un type énuméré) entraîne la définition de nouvelles valeurs (les constituants de l'énumération). [Leroy00] indique les modifications nécessaires à effectuer : il faut se définir un module environnement dans lequel cette spécificité est prise en compte (au niveau de la fonction `add_value`).

4.2. Vérifications statiques

La deuxième étape du processus de construction du système de modules à la HLL concerne la sémantique statique du langage (voir la partie 4.1 pour la signature du module). Nous ne détaillons pas ici l'écriture de la vérification statique associée au langage B. Le module est obtenu par adaptation de la contribution faite par Mamoun Filali-Amine de l'IRIT, dans le cadre du projet BCaml [4]. Cette contribution est elle-même inspirée de [Abrial96] et de [Steria98].

```
module BTyping = struct
   module Core = B
   module Env = BEnv
   open BMod
   open B
   ...
end
```

Une fois le module du langage basique et sa vérification statique définis, il reste à utiliser les foncteurs :

```
 module BMod = Mod_syntax(B)

 module BEnv = ... (* reprise du code du foncteur et adaptation *)
```

et

```
 module BModTyping =
Mod_Typing(BMod)(BEnv)(BTyping)
```

À ce stade de développement, nous possédons un système de module à la HLL pour les spécifications «basiques» B.

Ce système doit néanmoins être adapté en raison des particularités de la modularité B décrites précédemment (partie 2.3.2).

4.3. Résolution des problèmes de visibilité

4.3.1. Problème lié à l'accessibilité des entités

Dans la partie 2.2, nous avions remarqué que, au sein d'un même composant et *selon la clause considérée*, les droits d'accès à une même entité externe ne sont pas nécessairement identiques.

Deux solutions sont envisageables. La première est d'intégrer au sein du module de vérification des types les contraintes inhérentes aux règles de visibilité. Mais cette solution ne nous paraît pas acceptable, car elle implique des modifications majeures du code de [Leroy00], ce qui nous priverait de toutes possibilités de réutilisation des développements faits autour de ce système de modules : nous tenons à utiliser le code de X. Leroy comme un composant externe.

[4]http ://www3.inrets.fr/B@INRETS/BCaml/

La deuxième solution se base sur une constatation : dans un module B, les entités sont regroupées par catégorie : les clauses. Si pour chaque catégorie, il est nécessaire d'avoir des règles de visibilités différentes, il faut intégrer une notion de bloc dans les composants B. En langage de modules, un bloc est lui-même un module (ou sous module). Comme les systèmes de modules à base de modules SML permettent d'avoir plusieurs visions d'un même module (signatures multiples), il est possible d'exprimer les contraintes de visibilité par simple restriction. Restreindre un module revient à exporter (rendre visible de l'extérieur) moins d'entités qu'avant la restriction.

Un groupe d'entité sera donc un module. *Un module B*, qui est un ensemble de groupe d'entités *sera donc un module composé de modules.*

Remarque. En se référant à la partie 2.2, l'utilisation de visions multiples d'un module nous permet d'obtenir la deuxième dimension (celle de `hpublic`, `hprotected` et `hprivate`), alors que l'utilisation de modules dans les modules nous permet d'obtenir la troisième dimension.

Notre approche consiste donc en une première phase de transformation syntaxique d'une implantation B en un module HLL. Cette technique est illustrée dans la figure 3, où sont mises en correspondance une implantation B et sa représentation dans un système HLL. Cette représentation est écrite en Objective Caml ([Weis et al.99]), dont le système de modules est très proche de celui de HLL.

`IMPLEMENTATION IMP`	`module` IMP = `struct`
`IMPORTS MCH1`	

`INVARIANT` inv	`module` Invariant = `struct`
	`module` Inv_Imports = (`MCH1` : IMPORTS_INVARIANT)
	`open` Inv_imports
	`let` invariant = inv
	`end`

`OPERATIONS`	`module` Operations = `struct` `open` Invariant `module` Op_imports = (`MCH1` : IMPORTS_OPERATIONS) `open` Op_imports
op1 = ...	`let` op1 = ... `end`
`END`	`end`

FIG. 3 – Transformation d'une machine B en syntaxe HLL

À chaque type de lien de composition du langage B, nous associons donc un type de module qui intègre les restrictions d'accès.

Dans l'exemple précédent, `MCH1` est importé par le composant `IMP`. Comme la vision de `MCH1` doit être différente suivant l'endroit où l'on se trouve dans `IMP`, dans chaque module constituant `IMP`, le module `MCH1` est restreint au bon type de module (`IMPORTS_INVARIANT` par exemple).

4.3.2. Problème lié au hiding principle

Les systèmes de modules SML permettent l'encapsulation par l'utilisation de signatures spécifiant les données exportées. En B, le principe d'encapsulation n'est pas appliqué totallement : l'accès en lecture est autorisé, mais pas celui en écriture.

Ce problème peut facilement être résolu lors du typage du " :=". Il suffit de n'accepter en partie gauche de cet opérateur d'affectation que des "path" (référence au type `path` défini dans [Leroy00]) construits grâce au constructeur `Pident`.

4.4. Signatures des modules pour les restrictions

Afin de transformer les implantations B en module HLL, il faut définir un certain nombre de masques (ou types de modules) pour les différents liens. Cela revient à exprimer les tableaux 2 et 3 en terme de signature de modules.

Dans ce qui suit, un type de module de nom `XXX_YYY` correspondra à la restriction qui provient d'une clause de liaison `XXX` dans le module `YYY`.

Le tableau 2 est en partie transcrit par les signatures de module suivantes :

```
module type IMPORTS_INVARIANT =
  sig
    module Sets               : BMod.mod_type
    module Concrete_constant  : BMod.mod_type
    module Abstract_constant  : BMod.mod_type
    module Concrete_variables : BMod.mod_type
    module Abstract_variables : BMod.mod_type
  end

module type SEES_INITIALISATION =
  sig
    module Sets               : BMod.mod_type
    module Concrete_constant  : BMod.mod_type
    module Concrete_variables : BMod.mod_type
  end
```

`BMod.mod_type` est le type des modules B. `BMod` est un module obtenu en 4.2. Étant donnée l'analogie entre les différentes cases des tableaux et par souci de synthèse, nous ne donnons pas ici la transcription des autres cases.

```
module type IMPORTS_OPER =
  sig
    module Sets               : BMod.mod_type
    module Concrete_constant  : BMod.mod_type
    module Abstract_constant  : BMod.mod_type
    module Concrete_variables : BMod.mod_type
    module Abstract_variables : BMod.mod_type
    module Operations         : BMod.mod_type
  end
```

5. Travaux en relation

L'étude de la sémantique des clauses de liaisons en B a déjà fait l'objet d'autres publications : [Bert et al.96], [Potet et al.98], [Buchi et al.99], [Dimitrakos et al.00].

[Bert et al.96] développent une algèbre des composants pour y exprimer les clauses INCLUDES et USES. Cet article propose également l'idée d'une possible corrélation entre une machine B et un module, ainsi que le besoin d'indépendance entre le langage B et ses constructions de modularité.

[Buchi et al.99] proposent un nouveau lien qui étend les possibilités de partage des données entre les composants B, en transformant le paradigme «un écrivain et plusieurs lecteurs», actuellement utilisé en B, en «plusieurs écrivains et plusieurs lecteurs».

[Dimitrakos et al.00] étudient également les différentes clauses de liaisons et mettent l'accent sur les conditions à remplir pour conserver les propriétés des composants.

[Potet et al.98] étudient les clauses de liaisons IMPORTS et SEES et les problèmes qui peuvent être engendrés par leur usage. En particulier, cet article démontre qu'il est possible de construire un projet B dont la structuration («pathologique») induit que le développement est considéré à tort comme correct. Les auteurs proposent néanmoins des critères, basés sur une analyse du graphe des dépendances, pour une élimination des configurations incorrectes.

Par ailleurs, d'autres travaux se sont consacrés à l'expression des caractéristiques de B au moyen d'autres formalismes ; citons les exemples d'un début de formalisation de B par Isabelle/HOL [Chartier98], de B par PVS et Coq [Bodeveix et al.99], [Bodeveix et al.00], [Pratten95].

Sans entrer dans une présentation détaillée de ces travaux, remarquons un point essentiel : certes, l'objectif d'une (méta) formalisation externe de B constituerait un apport conséquent au fondement théorique de B, cependant les travaux cités n'envisagent pas la méthode B dans sa globalité et ne traitent pas de la modularité. La formalisation reste cantonnée au langage basique. *Sans que cela soit une critique*, ce constat démontre bien que l'expression formalisée de la modularité de B constitue à la fois un objectif et une difficulté centrale.

Dans ce contexte, notre approche est orthogonale et pragmatique, car ayant pour finalité la réalisation d'un outillage, nous ne pouvons éviter d'aborder le langage B dans sa globalité et nous traitons la modularité de B **sans** faire intervenir (directement) les fondements théoriques de B.

6. Conclusions

L'utilisation du système HLL de X. Leroy, devrait nous permettre non seulement d'exprimer la sémantique des clauses de liaisons en B, mais également, moyennant une adaptation de la syntaxe abstraite et du type-checking (pour "coller" avec les types des modules de HLL), d'obtenir une implantation.

Du point de vue mise en œuvre, le travail se situe plus en amont (au niveau de l'automatisation des transformations de composants B vers modules $BHLL$) que sur la spécification du langage elle même. En effet, la plateforme *BCaml* fournit l'analyseur syntaxique, une syntaxe abstraite et une implantation du typage. Pour développer un système prenant en entrée des composants B et vérifiant la sémantique statique, il suffit d'écrire le module (module au sens composant logiciel) qui effectuera la transformation des arbres obtenus par l'analyseur syntaxique en modules $BHLL$.

Du point de vue du problème que nous souhaitons traiter, il s'agit de répondre aux deux questions :
- *Est-il possible d'exprimer la modularité des spécifications B dans un système de modules HLL ?*
- *L'expression que nous avons fait de la modularité permet-elle de mieux comprendre les dépendances en B ?*

Notre étude étant limitée à une sous-partie du langage B, nous devons nous contenter d'une réponse partielle en ce qui concerne la première question, réponse partielle, mais affirmative. La deuxième question est plus suggestive et c'est à chacun d'en décider. Notre avis est qu'exprimer la modularité de B dans un système "intuitif" comme HLL permet de le clarifier. C'est pour cela que notre choix s'est orienté vers HLL plutôt que vers des systèmes de modules plus complexes, comme celui de [Ancona et al.] par exemple, système proche de celui de HLL, mais moins "intuitif" car possédant des concepts supplémentaires.

L'extension de notre approche à l'ensemble du langage B est notre prochaine étape avec la difficulté centrale que constitue l'intégration de la notion de raffinement. Dans le contexte du langage SIGNAL, [Nowak et al.97] expriment ce raffinement de spécifications comme du sous-typage de signature. Cependant, en plus de la notion de raffinement algorithmique, *B* utilise également la notion de raffinement de données. Et dans ce cas, la notion de sous-typage de signature ne suffit pas.

Références

[Abrial et al.97] Abrial (Jean-Raymond) et Mussat (Louis). – Specification and design of a transmission protocol by successive refinements using B. *Mathematical Methods in Program Development*, éd. par Broy (Manfred) et Schieder (Birgit). pp. 129–200. – Springer, 1997.

[Abrial96] Abrial (Jean-Raymond). – *The B Book - Assigning Programs to Meanings.* – Cambridge University Press, août 1996.

[Ancona et al.] Ancona (David) et Zucca (Elena). – A calculus of module systems. *Journal of Functional Programming.* – To appear.

[Ancona et al.98] Ancona (Davide) et Zucca (Elenna). – A theory of mixin modules : Basic and derived operators. *Mathematical structures in computer science*, vol. 8, n 4, August 1998, pp. 401–446.

[Behm et al.97] Behm (Patrick), Desforges (Pierre) et Mejia (Fernando). – *Application de la méthode B dans l'industrie ferroviaire*, chap. III, pp. 59–88. – Observatoire Français des Techniques Avancéees & Lavoisier TEC & DOC, 1997.

[Behm et al.99] Behm (Patrick), Benoit (Paul), Faivre (Alain) et Meynadier (Jean-Marc). – METEOR : A successful application of B in a large project. In Wing et al. [Wing et al.99], pp. 369–387.

[Bert et al.96] Bert (Didier), Potet (Marie-Laure) et Rouzaud (Yann). – A study on components and assembly primitives in B. *Proceedingsof 1st Conference on the B method*, éd. par Habrias (Henri). IRIN Institut de recherche en informatique de Nantes, pp. 47–62. – 3 rue du Maréchal Joffre, BP 34103, 44041 Nantes Cedex 1, novembre 1996.

[Bieber et al.94] Bieber (P.) et Boulahia-Cuppens (N.). – Formal development of authentication protocols. *Proceedingsof BCS-FACS Sixth Refinement Workshop.* – 1994.

[Bodeveix et al.99] Bodeveix (Jean-Paul), Filali (Mamoun) et Munoz (César). – A formalization of the B method in Coq and PVS. *FM'99 – B Users Group Meeting – Applying B in an industrial context : Tools, Lessons and Techniques*, pp. 32–48. – 1999.

[Bodeveix et al.00] Bodeveix (Jean-Paul), Filali (Mamoun) et Munoz (César). – Formalisation de la méthode B en COQ et PVS. *AFADL'2000*, pp. 96–110. – 2000.

[Buchi et al.99] Büchi (Martin) et Back (Ralph). – Compositional symmetric sharing in B. In Wing et al. [Wing et al.99], pp. 431–451.

[CC-99] *Common Criteria for Information Technology Security Evaluation.* – August 1999. Part 3 : Security Assurance Requirements, version 2.1.

[Chartier98] Chartier (Pierre). – Formalisation of B in Isabelle/HOL. *B'98 : The 2nd International B Conference*, pp. 66–82. – 1998.

[Dimitrakos et al.00] Dimitrakos (Theo), Bicarregui (Juan), Matthews (Brian), Maibaum (Tom) et Robinson (Ken). – Compositional structuring in the B method : A logical viewpoint of the static context. *ZB'2000 – International Conference of B and Z Users.* – 2000.

[Harper et al.86] Harper (Robert), MacQueen (David B.) et Milner (Robin). – *Standard ML.* – Rapport technique, University of Edinburgh, 1986. ECS LFCS 86-2.

[Laleau et al.00] Laleau (Regine) et Mammar (Amel). – *Using a Formal Refinement to Derive Relational Database Implementations from B Specifications.* – Rapport technique, CNAM-CEDRIC, février 2000.

[Lanet et al.98] Lanet (Jean-Louis) et Requet (Antoine). – Formal proof of smart card applets correctness. *Proceedings of the Third Smart Card Research and Advanced Application Conference (CARDIS'98)*, p. ? ? – Louvain-la-Neuve, (be), septembre 1998.

[Leroy96] Leroy (Xavier). – *A modular module system.* – Research report n2866, INRIA, avril 1996.

[Leroy00] Leroy (Xavier). – A modular module system. *Journal of Functional Programming*, vol. 10, n3, 2000, pp. 269–303.

[Macqueen86] MacQueen (David B.). – *Modules for Standard ML.* – Rapport technique, University of Edinburgh, 1986. in [Harper et al.86].

[Nowak et al.97] Nowak (D.), Talpin (J.P.) et Gautier (T.). – *Un système de modules avancé pour SIGNAL.* – Rapport technique, IRISA, 1997.

[Potet et al.98] Potet (Marie-Laure) et Rouzaud (Yann). – Composition and refinement in the B method. *B'98 : The 2nd International B Conference.* – 1998.

[Pratten95] Pratten (C.H.). – An introduction to proving AMN specifications with PVS and the AMN-PROOF tool. *Proc. Z Twenty Years on - What is its Future*, éd. par HABRIAS (Henri). IRIN-IUT de Nantes, pp. 149–165. – October 1995.

[Sabatier et al.99] Sabatier (Denis) et Lartigue (Pierre). – The use of the B formal method for the design and the validation of the transaction mechanism for smart card applications. In Wing et al. [Wing et al.99], pp. 348–368.

[Sannella et al.92] Sannella (D.T.) et Wallen (L.A.). – A calculus for the construction of modular prolog programs. *Journal of logic programming*, 1992.

[Spivey98] Spivey (Michæl). – *The Z notation : a reference manual.* – University of Oxford, 1998. second edition.

[Steria98] STERIA. – *Manuel de référence du langage B version 1.7*, mars 1998.

[Watson97] Watson (Geoffrey Norman). – A comparison of modularity in B and Cogito. *Proceedings of FMP'97*, éd. par Groves (L.) et Reeves (S.). – 1997. Proceedings of FMP'97.

[Weis et al.99] Weis (Pierre) et Leroy (Xavier). – *Le langage Caml.* – Dunod, 1999.

[Wing et al.99] Wing (Jeannette M.), Woodcock (Jim) et Davies (Jim) (édité par). – *Proceedings of FM'99 : World Congress on Formal Methods.* – Springer Verlag, septembre 1999.