Formalisation de la mthode B en Coq et PVS

Jean-Paul Bodeveix et Mamoun Filali
IRIT-Universit Paul Sabatier
118 route de Narbonne, F-31062 cdex, Toulouse, France
mail : {bodeveix,filali}@irit.fr

Csar A. Muñoz *
Institute for Computer Applications in Science and Engineering (ICASE)
Mail Stop 132C
NASA Langley Research Center Hampton VA 23681-2199
mail : munoz@icase.edu

Résumé

Nous formalisons le mcanisme de *substitution gnralise* de la mthode B dans la logique d'ordre suprieur de Coq et PVS. D'une part, cette tude permet de formaliser les constructions de B et de valider leurs proprits. D'autre part, elle permet d'envisager des dveloppements B au sein des environnements de preuve Coq et PVS. Cette intgration est supporte par l'outil PBS qui traduit une machine abstraite B en une thorie PVS. Nous illustrons les deux domaines d'application par la proposition et la validation d'un nouvel oprateur de composition parallle et par l'tude d'un protocole de cohrence mmoire atomique.

1 Introduction

Au cours de ces dernires annes, beaucoup de travaux ont port sur la conception et la mise en oeuvre d'assistants de preuves. Ces efforts ont t concrtiss par la ralisation de systmes tels que HOL [Gor93], Coq [BBC+97], et PVS [ORS92]. Ces systmes disposent de tactiques de preuve automatique, mais ils manquent de mthodologie pour aborder globalement le processus de dveloppement d'un logiciel.

La mthode B [Abr96a] est une mthode formelle de dveloppement de logiciels. Elle est l'aboutissement des travaux mens par J.-R. Abrial [Abr91] depuis 1980 et continue tre dveloppe par des industriels et des universitaires. B fournit une notation uniforme pour la spcification, la conception, et l'implantation effective de systmes. La mthodologie incite la structuration du dveloppement d'un logiciel.

Dans cet article, nous nous sommes intresss au plongement de la mthode B dans la logique d'ordre suprieur de Coq et PVS. Il existe principalement deux techniques de plongement du langage spcialis d'une mthodologie dans un langage de spcification usage gnral. Elles sont connues sous le nom de plongement *profond (deep)* et *superficiel (shallow)* [BGG+92]. Dans le cas d'un plongement profond, la mthodologie est compltement formalise comme un objet de la logique du langage de spcification. Dans ce cas, il est possible de prouver des proprits mta-thoriques du langage considr, mais la preuve de proprits d'une application particulire exige en gnral un codage dlicat. Dans le cas de l'approche

*Cette tude a t finance par *National Science Foundation*, subvention CCR-9712383, pendant que l'auteur tait SRI International (Menlo Park), et par *National Aeronautics and Space Administration*, subvention NAS1-97046, pendant que l'auteur tait ICASE - NASA Langley Research Center.

superficielle, il y a traduction au niveau syntaxique des objets du langue source l'aide d'objets smantiquement quivalents du langage cible. Dans ce cas, les proprits mta-thoriques ne peuvent pas tre vrifies mais le codage d'applications particulires est plus simple.

Notre approche est mi-chemin entre un plongement profond et un plongement superficiel. Elle formalise le mcanisme de substitution gnralise de la mthode B dans les systmes Coq et PVS, et donc nous pouvons raisonner sur la mthode un niveau mta-thorique. De plus, nous fournissons un codage compact fortement intgr la logique de Coq et PVS. Par exemple, des lments logiques de la mthode B sont exprims l'aide de la thorie des types dpendants de Coq et PVS. Dans le cas de PVS, nous avons mcanis notre plongement de sorte que la notation B puisse tre utilise en tant que sur-couche du langage de spcification de PVS.

Cet article est organis comme suit. Dans la section 2, nous introduisons la mthode B et les systmes Coq et PVS. Dans la section 3, nous prsentons notre formalisation. La section 4 dcrit l'architecture du composant logiciel qui met en oeuvre notre plongement et prsente deux tudes de cas. La section 5 conclut sur quelques perspectives.

2 Prsentation de B, Coq et PVS

2.1 La mthode B

B est une mthode oriente-tat qui couvre tout le cycle de vie du dveloppement d'un logiciel. Elle fournit un langage unique, le formalisme des machines abstraites (*Abstract Machine Notation*), pour spcifier, concevoir, et implanter effectivement une application. Un dveloppement B consiste en une spcification abstraite suivie de quelques tapes de raffinement. Le dernier raffinement correspond une mise en oeuvre. La validit de la construction repose sur la vrification des obligations de preuve associes chaque tape de dveloppement.

Une spcification B est compose d'un ensemble de modules appels *machines (abstraites)*. Chaque machine a un tat interne, et fournit des services permettant un utilisateur externe d'accder son tat ou de le modifier. Syntaxiquement, une machine consiste en plusieurs clauses dterminant les proprits statiques et dynamiques de l'tat.

Considrons la machine abstraite suivante spcifiant un systme trs simple qui mmorise une valeur et fournit les services *lire* et *crire* sur cette valeur.

```
MACHINE simple(T)
  VARIABLES
    valeur
  INVARIANT
    valeur: T
  INITIALIZATION
    ANY x WHERE x:T THEN
      valeur := x
    END;
  OPERATIONS
     write(v) ≙
       PRE v:T THEN
         valeur := v
       END;
     v ⟵ lire ≙
       v := valeur
```

END

La machine `simple`, plutt que de spcifier un systme, spcifie une famille de systmes dpendant du paramtre `T`, ayant les mmes proprits abstraites. En B, un paramtre peut tre un scalaire ou un ensemble abstrait non vide. Par convention, un paramtre qui commence par une lettre majuscule est un ensemble abstrait. Les clauses VARIABLES dfinissent l'tat de la machine. Dans ce cas, nous utilisons seulement une variable, `valeur`. La clause INVARIANT contraint le domaine de cette variable. Il affirme que `valeur` est un lment de `T`. Notons qu' ce stade du dveloppement le domaine de la valeur est abstrait. Nous supposons uniquement qu'il est fini et non vide. L'tat initial de la machine, qui doit satisfaire l'invariant, est spcifi dans la clause d'initialisation. Dans cet exemple, la variable `valeur` est initialise par un lment quelconque de `T`.

Les services fournis par la machine sont spcifis dans la clause OPERATIONS. Dans ce cas, nous spcifions une opration `ecrire` pour crire une nouvelle valeur et une autre appel `lire` pour y accder. Pour spcifier une opration, B introduit la notion de *substitution gnralise*. On distingue six substitutions de base : l'identit, l'affectation multiple, la slection, le choix born ou non born, et la substitution avec prcondition. De nouvelles substitutions peuvent alors tre construites par composition de ces dernires. Une substitution gnralise se comporte comme un transformateur de prdicat[1]. Par exemple, la substitution gnralise

```
PRE v:T THEN
  valeur := v
END;
```

correspond au transformateur de prdicat $[v \in T \mid valeur := v]$ qui se dfinit pour tout prdicat P comme suit :

$$[v \in T \mid valeur := v]P \quad \Leftrightarrow \quad v \in T \wedge [valeur := v]P$$

Dans la section 3, nous donnons une dfinition plus prcise de cette notion.

D'autres clauses permettent l'introduction de constantes (`CONSTANTS`) et d'hypothses sur les paramtres de la machine (`CONSTRAINTS`). Le dveloppement de gros logiciels est support par plusieurs mcanismes de composition, comme par exemple, `INCLUDES`, `SEES`, et `IMPORTS`. Ces mcanismes donnent des droits d'accs diffrents aux oprations ou aux variables locales partir d'une machine externe.

Une spcification abstraite peut tre matrialise dans une mise en oeuvre par un mcanisme de raffinement. Supposons par exemple que dans une vraie mise en oeuvre de notre systme, une valeur peut tre stocke dans un registre ou en mmoire. Nous pouvons alors raffiner la machine prcdente comme suit :

```
REFINEMENT simple_ref(T)
  REFINES simple
  VARIABLES
    val_mem, val_reg, etat
  INVARIANT
    etat:bool ∧ val_mem:T ∧ val_reg:T ∧
    (etat ∧ val_mem = valeur) ∨ (¬ etat ∧ val_reg = valeur)
  INITIALIZATION
    ANY X,f WHERE x:T ∧ f:bool THEN
      etat,val_mem,val_reg := f,x,x
```

[1]Un transformateur de prdicat est une fonction qui associe un prdicat un prdicat.

```
    END;
  OPERATIONS
    ecrire(v) ≙
      PRE V:T THEN
        CHOICE
          etat,val_mem := true,v
        OR
          etat,val_reg := false,v
        END
      END;
    v ⟵ lire ≙
      IF etat THEN v := val_mem
      ELSE v := val_reg
      END
END
```

Le raffinement `simple_ref` utilise deux variables pour stocker la valeur, une pour la mmoire, `val_mem`, et une autre pour le registre, `val_reg`. La variable `etat` indique laquelle des deux est utilise. Les variables de la machine `simple_ref` sont dites concrtes, celles de la machine `simple` sont dites abstraites. L'invariant d'un raffinement connecte les variables abstraites et concrtes. Du point de vue de l'utilisateur, les services fournis par `simple_ref` peuvent se substituer ceux fournis par `simple`.

La validit d'une machine B est donne par des *obligations de preuve* spcifiant que :

- Il existe une instantiation des paramtres, ensembles, et constantes satisfaisant les contraintes de la machine.
- L'tat initial satisfait l'invariant.
- L'invariant est conserv par les oprations.

La validit de la composition et celle du raffinement sont aussi garanties par des obligations de preuve.

2.2 Les systmes Coq et PVS

Coq [BBC+97] et PVS [ORS92] sont deux systmes de vrification usage gnral qui intgrent un langage de spcification bas sur la logique d'ordre suprieur type et un assistant de preuve interactif.

Coq est un assistant de preuve dvelopp par le Projet Coq l'INRIA (Rocquencourt). PVS est un systme de vrification dvelopp par le groupe de Mthodes Formelles du SRI International (Menlo Park).[2]

PVS et Coq ont une architecture similaire. Les deux systmes fournissent une interface interactive dans laquelle l'utilisateur dveloppe une thorie et nonce des proprits exprimes dans le langage de spcifications. Les thories sont structures en modules – *sections* en Coq et *thories* en PVS – qui regroupent des objets mathmatiques et logiques. Le langage de spcification est la logique d'ordre suprieur. Le systme de types vrifie la correction des spcifications et produit ventuellement des obligations de preuve (PVS).

Les langages de spcification de PVS et de Coq comportent les oprations standard des langages de programmation fonctionnelle; par exemple, les conditionnelles, les fonctions (dnotes par `[x: A]M`

[2] Les deux systmes sont librement distribus depuis les pages internet des deux projets : `http://coq.inria.fr` et `http://pvs.csl.sri.com`.

en Coq et par λ(x: A): M en PVS); les fonctions rcursives doivent suivre des schmas de dfinition assurrant leur existence. Les deux systmes sont fortement typs. Les types supports par les deux systmes incluent les entiers relatifs, les enregistrements (dfinis par Record R: Type: mk_R{x: A; ... } en Coq et par R: TYPE=[# x: A, ... #] en PVS) , les ensembles et les types de donnes abstraits (appels inductifs en Coq).

Malgr les similitudes mentionnes, les deux systmes sont fondamentalement diffrents. PVS est bas sur la logique *classique* d'ordre suprieur enrichie par une thorie des types. La thorie des types de PVS supporte le sous-typage par prdicat: si T est un type et P est un prdicat sur T ($P : T \rightarrow$ bool), $\{x : T|P(x)\}$ est le type de tous les lments de T qui satisfont P. Ce sous-type est aussi not (P). Le sous-typage s'avre trs pratique pour crire des spcifications, mais il rend le typage indcidable. Pour imposer la discipline de typage, le systme engendre des obligations de preuve que l'utilisateur doit rsoudre l'aide de l'assistant de dmonstration. De plus, PVS possde des procdures de dcision puissantes pour la logique classique avec galit et l'arithmtique linaire.

Quant Coq, il est bas sur le *calcul des constructions inductives*, lui mme une extension du calcul des constructions qui correspond une logique *intuitionniste* d'ordre suprieur. Le calcul des constructions est une thorie des types dpendants trs puissante, mais toujours dcidable. Si T_1 et T_2 sont des types, il est possible de construire un type $(x : T_1)T_2$ o T_2 peut dpendre de x. Si la variable x n'est pas libre dans T_2 on notera $T_1 \rightarrow T_2$. Contrairement PVS, une des priorits du dveloppement de Coq est la certification des rponses donnes par le systme. En effet, chaque thorme tabli, est associ un terme de preuve permettant de reconstruire l'nonc du thorme partir des rgles d'infrence de la logique[3].

Dans la suite, nous donnons le codage de la plupart des constructions B en Coq ou en PVS. Nous les donnons toutes les deux lorsqu'elles diffrent de manire significative; autrement, la syntaxe PVS tant plus proche de celle de B, nous ne donnons que le codage PVS.

3 Formalisation de la smantique de B en Coq et PVS

3.1 Les notations de base

La mthode B repose sur la relation avant-aprs entre deux tats d'une machine, introduite par le mcanisme de substitution gnralise. Alors que la relation binaire habituelle modlisant les instructions non-dterministes ne distingue pas non terminaison et arrt, la substitution avec prcondition spcifie les tats pour lesquels un calcul termine. En outre, un calcul lanc partir d'un tat ne satisfaisant pas la prcondition conduit n'importe o. Par consquent, le modle de calcul de base de B, la substitution, peut tre reprsent par une structure champs dpendants comprenant un prdicat pre et une relation binaire rel tels que :

$$\forall e_1 : \neg \text{pre}(S)(e_1) \Rightarrow \forall e_2 : \text{rel}(S)(e_1, e_2) \tag{1}$$

La substitution S peut aussi tre caractrise par le transformateur de prdicat [S] li pre et rel par les quations suivantes :

$$[S](p) = \lambda e_1 : \text{pre}(S)(e_1) \wedge \forall e_2 : \text{rel}(S)(e_1, e_2) \Rightarrow p(e_2) \tag{2}$$

$$\text{pre}(S) = [S](\text{TRUE}) \tag{3}$$

$$\text{rel}(S) = \lambda e_1, e_2 : \neg([S](\lambda e : e \neq e_2))(e_1) \tag{4}$$

[3]Il s'agit en fait des rgles de typage.

Cependant, il est noter que tous les transformateurs de prdicats ne peuvent tre associs des substitutions, une condition suffisante tant sa monotonie. Dans la suite, nous reprsentons une substitution par les prdicats `pre` et `rel`, plus intuitifs pour la dfinition de nouvelles substitutions.

Formellement, une substitution est modlise par une transition entre tats de type gnrique `Env` l'aide d'une structure comprenant les champs `pre` et `rel`, et la preuve qu'ils satisfont l'quation (1). En Coq, nous utilisons un structure champs dpendants, et en PVS, la possibilit de dfinir un sous-type par un prdicat. Dans les dfinition suivantes, nous introduisons le prdicat `Is_Transition` spcifiant la proprit (1).

Substitutions en Coq

```
Definition Is_Transition: (Env → Prop) → (Env → Env->Prop) → Prop :=
 [p:(Env → Prop)] [tr: Env → Env → Prop]
    (e1,e2:Env) (~(p e1) → (tr e1 e2)).

Record Transition: Type := mk_Transition {
  pre: Env → Prop;
  rel: Env → Env → Prop;
  obl: (Is_Transition pre rel)
}.
```

Substitutions en PVS

```
PreTransition: TYPE = [#
  pre: [Env → bool],
  rel: [Env,Env → bool]
#]

Is_Transition: [PreTransition → bool] =
  λ (tr:PreTransition): ∀ (e1,e2: Env) ¬ pre(tr)(e1) ⇒ rel(tr)(e1,e2)

Transition: TYPE = (Is_Transition)
```

3.2 L'espace d'tat d'une machine

La plupart des substitutions sont indpendantes de la reprsentation de l'espace d'tat. Cependant, certaines utilisent le fait que l'environnement est un ensemble de variables types (par exemple, la substitution parallle [Section 4.2]). La notion d'environnement typ se dfinit simplement l'aide du systme de type de Coq [Hey97]. Il suffit de considrer une fonction associant chaque variable son type. En PVS, le codage n'est pas aussi simple puisqu'un type n'est pas un terme.

3.2.1 Codage de l'espace d'tats en Coq

En Coq, l'environnement `Env`, paramtr par un ensemble d'identificateurs de variables `Id` et une fonction de typage `typeof`, se dfinit comme suit :

```
Section modele.
  Variable Id: Set.
  Variable typeof: Id → Set.
  Definition Env := (i:Id)(typeof i).
```

```
  ...
End modele.
```

Un environnement de type `Env` associe un identificateur `i` de type `Id` une valeur dont le *type* (`typeof i`) dpend de `i`.

Par exemple, considrons l'tat constitu de deux variables {`x: nat; b: bool`}. Pour instancier les dfinitions introduites dans la section `modele`, nous introduisons les dclarations suivantes o `Id` est un type numr contenant les constantes `x` et `b`, et `typeof` une fonction prenant en argument un identificateur `i` de type `Id` et retournant son type, ici `nat` ou `bool`.

```
Inductive Id: Set := x: Id | b: Id.

Definition typeof: Id → Set := [i:Id] Cases i of x => nat | b => bool end.
```

Notons qu'une telle dfinition est valide en Coq puisque les types sont des termes et qu'il est donc possible d'laborer des expressions de type.

3.2.2 Codage de l'espace d'tat en PVS

Un type n'tant pas assimil un terme, nous ne pouvons pas utiliser un codage de `typeof` semblable celui de Coq. Cependant, nous pouvons utiliser l'identification sous-type - prdicat introduite par PVS. Ainsi, pour chaque machine nous synthtisons un nouveau super-type dont les types des variables de la machine seront sous-types.

En PVS, le module qui dfinit le type `Env` est dclar comme suit, o `Val` reprsente l'union des types des variables d'tat, et `typeof` une fonction des identificateurs de variables vers les sous-ensembles de `Val`:

```
modele [Id,Val: TYPE, typeof: [Id → Val → bool]: THEORY
BEGIN
  Env: TYPE = [i:Id → (typeof(i))]
  ...
END modele
```

Par exemple, si nous considrons l'tat {`x: nat; b: bool`}, `Val` est cod comme un type de donnes (`DATATYPE`) avec un variant pour chaque variable d'tat. A chaque variant est associ un constructeur (dans ce cas, `d_x` et `d_b`), un destructeur (dans ce cas, `v_x` et `v_b`), et un observateur (dans ce cas, `d_x?` et `d_b?`). La dfinition du type de donne `Val` a la forme suivante :

```
Id: TYPE = {x,b}

Val: DATATYPE
BEGIN
  d_x(v_x:nat):d_x?
  d_b(v_b:bool):d_b?
END Val

typeof(i:Id): [Val → bool] = IF i = id_x THEN d_x? ELSE d_b? ENDIF
```

Cet encodage de l'tat est d'un usage plutt dlicat puisque nous devons manipuler constructeurs et destructeurs du type `Val`. Donc, d'un point de vue utilisateur, nous introduisons aussi une structure encodant l'tat[4] et des conversions entre les deux reprsentations.

[4]Un tel encodage ne peut tre exprim au niveau mta-thorique.

D'aprs notre exprience antrieure dans le codage des tats d'une machine en logique d'ordre suprieur, il apparat que les types dpendants sont ncessaires; autrement, soit l'espace d'tat est non typ [vW90], soit il ncessite un codage complexe [BF95].

3.3 Les substitutions de base

La substitution de base, rduite en fait l'affectation multiple, se dfinit partir d'une fonction `f` sur les environnements comme suit :

```
BASIC_PT(f:[Env → Env]): Transition =
  (# pre := TRUE,
     rel := λ (e1,e2: Env): e2 = f(e1)
  #)
```

Les lments de type enregistrement sont dfinis en PVS en utilisant les symboles `(#` et `#)`.

La dfinition en Coq est semblable. Cependant, le fait que la transition vrifie l'quation (1) doit tre prouv avant la dfinition de la structure. En PVS, une obligation de preuve est automatiquement produite par le contrleur de type au moment de la dclaration de la structure.

3.4 Les substitutions gnralises

Comme nous l'avons dit prcdemment, les substitutions gnralises de B peuvent tre dfinies par leurs attributs `pre` et `rel`. Par exemple, pour la substitution avec prcondition note $P \mid S$ o P est un prdicat et S une substitution, nous avons[5] :

$$\text{pre}(P|S) = P \wedge \text{pre}(S) \tag{5}$$

$$\text{rel}(P|S) = \lambda e_1, e_2 : P(e_1) \Rightarrow \text{rel}(S)(e_1, e_2) \tag{6}$$

On montre que :

$$[P|S] = \lambda p : P \wedge [S](p) \tag{7}$$

Dans le cas de la substitution parallle, B propose [Abr96b] une construction $\|_B$ ayant principalement un rle de structuration. Nous avons introduit cette construction par la dfinition PVS suivante :

```
PAR(trl: Transition[Envl],trr: Transition[Envr]):
      Transition [[Envl,Envr],[Envl,Envr]] =
  (# pre:=(λ (el:Envl,er:Envr): pre(trl)(el) ∧ pre(trr)(er)),
     rel:=(λ (elr1:[Envl,Envr],elr2:[Envl,Envr]):
       (pre(trl)(PROJ_1(elr1)) ∧ pre(trr)(PROJ_2(elr1))) ⇒
          (rel(trl)(PROJ_1(elr1),PROJ_1(elr2)) ∧
           rel(trr)(PROJ_2(elr1),PROJ_2(elr2))))
  #)
```

o `PROJ_1(e1,e2) = e1`, et `PROJ_2(e1,e2) = e2`.

Une proprit intressante d'une telle construction est son comportement par rapport la squence

$$(F_1 \parallel G_1); (F_2 \parallel G_2) = (F_1; F_2) \parallel (G_1; G_2)$$

La validit de cette proprit est justifie par le fait que les substitutions F_i et G_i sont dfinies sur des espaces disjoints. Par suite, il n'y a pas d'interfrence. Cette proprit a t prouv en Coq et en PVS.

[5] Les oprateurs boolens habituels sont surchargs pour les prdicats.

3.5 Invariants

Les invariants de la machine sont introduits comme une contrainte `Inv` sur l'espace d'tat. En PVS, cette contrainte est exprime naturellement en dfinissant un sous type l'aide d'un prdicat :

```
Typed_Env: TYPE = (Inv)
```

En Coq, nous utilisons un type dpendant pour introduire la preuve de la contrainte sur l'espace d'tat. [6]

```
Record Typed_Env: Set := {
 env:> Env;
 ctr: (Inv Env); (* contrainte de validit *)
}.
```

Notons encore les approches diffrentes de Coq et PVS en ce qui concerne la dfinition d'un espace d'tat contraint.

- La possibilit de dfinir un type par un prdicat entrane l'indcidabilit du typage de PVS. Par suite le systme de type de PVS produit des obligations de preuve lors de la vrification de type. Il appartient alors l'utilisateur de "dcharger" ces obligations en les prouvant. Il est noter que ces preuves sont pour la plupart automatiquement ralises grce aux puissantes procdures de dcision de PVS.
- En Coq, la preuve (`ctr`) que l'espace d'tat vrifie la contrainte `Inv` doit tre fournie lors de la construction de l'espace. Cependant, la tactique `Refine` de Coq peut tre utilise pour diffrer une telle preuve.

En PVS, le prdicat de typage d'une transition `Is_Tr_Inv` devant prserver un invariant s'exprime comme suit :

```
Is_Tr_Inv: [Transition → bool] =
  λ (tr: Transition):
     ∀ (e1: Typed_Env, e2: Env):
        (pre(tr)(e1) ∧ rel(tr)(e1, e2)) ⇒ Inv(e2)
```

3.6 Raffinements

Intuitivement, une machine M_1 est raffine par une machine M_2 si la machine M_2 peut remplacer la machine M_1 dans le contexte o M_1 est utilise. Le *raffinement des machines* est dfini comme suit :

- les machines M_1 et M_2 ont la mme signature (ensemble d'oprations).
- Il y a une relation de raffinement entre deux substitutions composes de manires identiques respectivement partir des oprations de M_1 et M_2.

En B, le raffinement entre deux substitutions est dfini comme une implication entre les transformateurs de prdicat associs aux substitutions.

```
IS_REFINED_BY (tr1, tr2: Transition): bool =
  ∀ (p: [Env → bool]): |- ([|tr1|](p) ⇒ [|tr2|](p))
```

[6] La notation `:>` introduit aussi une conversion implicite de `Typed_Env` vers `Env`.

Tableau 1: Codage de B en Coq et PVS

constructions B	Coq	PVS
machine	section	thorie
tat	type dpendant	type dpendant + type abstrait
substitution	structure contrainte {pre,rel}	
invariant	tats contraints et substitutions	
raffinement	relation entre transitions	
obligation de preuve	nonc de thorme	condition de correction du type

Il est noter que le raffinement de machine repose sur la monotonie du raffinement de substitution. En effet, tant donn qu'un programme dfini partir des oprations d'une machine est en fait une substitution qui rsulte de la composition de substitutions offertes par cette dernire, la proprit de monotonie et le raffinement respectif des substitutions offertes assure le raffinement entre programmes composs de manire identique. Il s'en suit que la monotonie est une proprit essentielle tablir lors de l'introduction d'un nouvel oprateur. Dans la section 4.2, nous illustrons cette dmarche sur l'introduction d'un nouvel oprateur de paralllisme que nous montrons effectivement monotone.

Nous avons prouv en Coq et PVS la monotonie des oprateurs B par rapport au raffinement. Par exemple, la monotonie de l'oprateur B de prcondition s'exprime ainsi :

```
MONOTONE_PRE: THEOREM
  ∀ (p:[Env → bool],tr1,tr2:Transition):
     IS_REFINED_BY(tr1,tr2) ⇒ IS_REFINED_BY(PRE(p,tr1),PRE(p,tr2))
```

3.7 Rsum

Le tableau 1 rsume les aspects cls du codage de B en Coq et PVS.

Cette tude nous a amen aux conclusions suivantes :

- les types dpendants et le sous typage permettent de coder simplement les tats d'une machine. De ce point de vue, Coq ne dispose pas d'un mcanisme de sous typage mais sa thorie des types est plus puissante que celle de PVS notamment en ce qui concerne les constructeurs de type.
- Les obligations de preuve de PVS implantent naturellement celles de B. Dans Coq, il appartient l'utilisateur d'noncer les thormes relatifs ces obligations. Cependant, la nouvelle tactique Coq `Refine` permet de faciliter cette dmarche.
- Un aspect plus technique concerne la mise jour des donnes structures (e.g. `record`). De ce point de vue, PVS propose la construction `with` qui construit une copie o seuls certains champs sont modifis. Cette construction manque Coq; cependant, elle se dfinit grce aux possibilits d'extension de la grammaire Coq.

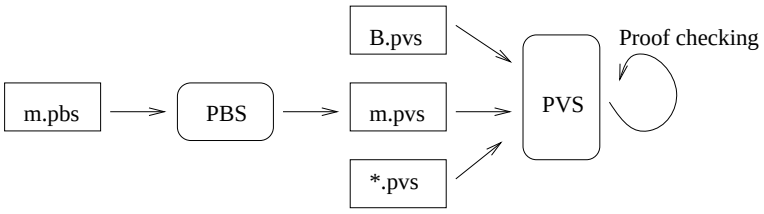

Figure 1: L'outil PBS

4 Etudes de cas

Pour exprimenter les aspects prsents dans cet article, nous avons ralis un outil permettant d'automatiser le plongement de B dans PVS [Muñ98]. Le systme PBS permet la dfinition de machines abstraites B au sein de PVS. De plus, nous avons crit des tactiques de preuve associes aux obligations gnres. La figure 1 illustre l'architecture gnrale du systme.

PBS peut tre vu comme un traducteur. Il prend en entre un fichier *m.pbs* contenant la description d'une machine abstraite et produit une thorie PVS dans le fichier *m.pvs*. Le fichier *B.pvs* contient notre formalisation de la mthode B. Comme suggr par le diagramme, une bote PBS associe une machine peut importer une thorie PVS. Ainsi, l'utilisateur n'est pas restreint quant aux types de donnes disponibles.

Lors de la conception de PBS nous avons fait des choix qui mritent d'tre comments. En premier lieu, la syntaxe des machines abstraites PBS est base sur celle de PVS. En consquence, ce n'est pas la syntaxe de rfrence dfinie dans [Abr96a]. Ce choix est motiv par le fait qu'il nous parait important que les lments de la machine abstraite sur lesquels la preuve sera mene soient facilement identifiables dans la machine abstraite source.

Une dcision plus importante concernait le typage d'une machine abstraite. Au lieu du langage de spcification base sur la thorie des ensembles [Abr96a], notre notation est base sur la logique d'ordre suprieur et la thorie de types de PVS. Dans une machine PBS, nous utilisons la clause TYPES plutt que la clause SET originelle de B. Notre choix de typer les *machines abstraites* est aussi motiv par les conclusions d'tudes de cas dveloppes dans [B$^+$97]. Ces dernires concernent diverses applications et couvrent plusieurs aspects du cycle de dveloppement d'un logiciel. Il en ressort notamment que :

- Les conditions de typage peuvent tre gres par la thorie des ensembles sous-jacentes de B. Toutefois, il s'avre qu'en ce qui concerne certains types d'objets (fonctions) la discipline de typage de B mne un style de spcification peu clair.
- Dans la mthode B, les constructeurs de types de donnes ne sont pas nombreux : jusqu' rcemment les structures n'taient pas dfinies en B.
- Les obligations de preuve concernent souvent le typage. Ces obligations pourraient tre rsolues par un vrificateur de type.

Notre avis est que la thorie des types peut combler certaines de ces lacunes.

Le systme PBS peut aussi bien tre utilis au niveau dveloppement qu'au niveau mta. Dans ce qui suit, nous illustrons la premire utilisation par le dveloppement d'un protocole de cohrence mmoire atomique et la seconde utilisation par la proposition d'une nouvelle smantique pour l'oprateur de paralllisme.

Actuellement, l'environnement PBS produit des thories PVS. En ce qui concerne Coq, nous avons uniquement formalis les constructions de la mthode B et valid la plupart des relations nonces dans le manuel B.

4.1 Dveloppement d'un protocole de cohrence mmoire atomique

Nous avons exprim dans PBS les tapes de dveloppement d'un protocole de cohrence mmoire atomique[Ste90]). Ce type de protocole appartient la classe des programmes ractifs paramtrs et la validation de ces problmes est un sujet de recherche trs actif.

Par rapport des dveloppements de problmes similaires que nous avions mens prcdemment [BF95] l'aide de l'assistant de preuves HOL, les aspects qui nous paraissent importants dans B sont la structuration des dveloppements par l'expression d'invariants, de contraintes et de raffinements. En ce qui concerne le paralllisme, nous avons constat le manque d'une construction parallle indexe qui exprimerait la rplication d'une substitution. Il est toutefois possible d'exprimer smantiquement cette construction l'aide des lambda expressions. D'un point de vue expression de proprits de problmes parallles, il s'avre que la notion d'invariant est suffisante pour noncer la plupart des proprits de sret. Il est intressant de noter que des extensions de B permettant d'exprimer de nouvelles proprits telles que la vivacit et de manire plus gnrale des proprits temporelles sont actuellement tudies [AM98].

Pour conclure sur le dveloppement l'aide de l'approche PBS, remarquons qu'il est maintenant largement admis que l'utilisation d'un assistant de preuve est en grande partie une *question de got*. De ce point de vue, l'approche adopte par PBS est unificatrice : on peut envisager un dveloppement o par exemple la preuve d'un invariant est faite en Coq et celle d'un raffinement en PVS.

4.2 Proposition d'un nouvel oprateur $\|$

La construction parallle actuellement propose par B a la signature suivante :

Syntaxe	Condition	Type
$F \parallel_B G$	$F \in \mathcal{P}(s) \to \mathcal{P}(s)$ $G \in \mathcal{P}(t) \to \mathcal{P}(t)$	$\mathcal{P}(s \times t) \to \mathcal{P}(s \times t)$

Les espaces d'tat sur lesquels oprent les composantes de cet oprateur tant distincts, il n'est pas possible d'envisager du paralllisme avec partage de donnes. Il est donc difficile de mener des dveloppements pour une machine cible architecture parallle avec mmoire partage. Nous proposons donc une nouvelle smantique de l'oprateur $\|$ admettant le partage de donnes. Cette smantique est base sur la dfinition des variables pouvant tre modifies par une substitution. Cet ensemble de variables est appel *support* de la substitution. Pour une substitution simple `x : = e` le support est dfini comme le singleton $\{x\}$. Pour toutes les autres substitutions nous dfinissons le support comme l'union des supports de ses composants. Par exemple, le support de `x: =e` $\|$ `y: =f` est l'ensemble $\{x, y\}$.

Pour dfinir la nouvelle smantique de la substitution parallle, nous introduisons par hritage un nouveau type transition qui possde le champ `support` reprsentant l'ensemble des variables modifies par la transition. Ce nouveau type est dfini comme suit :

```
Transition_1: TYPE = [#
  support: setof[Id],
  tr: {Transition | ∀  e1 e2:
       (Pre(tr)(e1) ∧ rel(tr)(e1,e2) ∧ e1(i) ≠ e2(i))
          ⇒ i ∈ support }
#]
```

Le champ support est synthtis par PBS. Par exemple, nous avons

```
  CHOICE_1(tr1,tr2: Transition_1): Transition_1 =
    (# support := support(tr1) ∪  support(tr2),
       tr := CHOICE(tr(tr1),tr(tr2))
    #)
```

Nous dfinissons alors le nouvel oprateur $\|$ comme suit :

```
PAR(l,r:Transition_1): Transition_1 =
  (# support := support(l) ∪ support(r),
     tr := (#
       pre := pre(l) ∧ pre(r),
       rel := λ e1 e2: (pre(l) ∧ pre(r))(e1) ⇒ ∃ el, er:
                rel(l)(e1,el) ∧ rel(r)(e1,er) ∧
                ∀ i:
                    IF i ∈ support(l) ∩ support(r) THEN
                         e2(i) = el(i) ∨ e2(i) = er(i)
                    ELSIF i ∈ support(l) THEN e2(i) = el(i)
                    ELSIF i ∈ support(r) THEN e2(i) = er(i)
                    ELSE e2(i) = e1(i) END
          #)
  #)
```

Cette dfinition mrite quelques commentaires :

- `x := y` $\|$ `y := x` a la signification habituelle : x,y `:= y,x`.
- `x := y` $\|$ `x := z` est dfini et exprime une affectation non dterministe `x`.
- Contrairement la construction $\|_B$ de B, nous n'introduisons pas un nouvel espace d'tat; de plus la construction propose est associative et commutative.
- Les composantes de $\|$ sont atomiques : seul un entrelacement de chacune de leur excution totale est considr.
- condition que le support soit prserv par raffinement, l'oprateur $\|$ est monotone par rapport au raffinement. Par suite, comme expliqu dans la section 3.6, il peut tre utilis comme une substitution de base dans la mthode B.

```
MONOTONE_PAR: THEOREM
 ∀ (U,V,S,T: Transition):
  (support(U) = support(V) ∧ support(S) = support(T)) ⇒
    (IS_REFINED_BY(U,V) ∧ IS_REFINED_BY(S,T)) ⇒
 IS_REFINED_BY(PAR(U,S),PAR(V,T))
```

Nous pouvons aussi comparer le $\|$ propos avec celui de [AO91] pour les programmes dterministes (not $\|_{AO}$ dans ce qui suit) . Cet oprateur est dfini pour deux instructions dterministes S_1et S_2 dont les interfrences vrifient la proprit suivante :

$$\text{change}(S_1) \cap \text{var}(S_2) = \emptyset$$
$$\text{var}(S_1) \cap \text{change}(S_2) = \emptyset$$

o

- change(S_i) est l'ensemble de variables modifies par S_i.
- var(S_i) est l'ensemble de variables accdes par S_i.

Ces restrictions sont en fait motives par un souci de compositionalit. En effet, elles permettent d'tablir la rgle de preuve suivante :

$$\frac{\{p_1\}S_1\{q_1\}, \{p_2\}S_2\{q_2\}}{\{p_1 \wedge p_2\}S_1 \parallel_{AO} S_2\{q_1 \wedge q_2\}}$$

o

$$\text{free}(p_i, q_i) \cap \text{change}(S_j) = \emptyset, \text{pour } i \in \{1,2\}, j \in \{1,2\} \text{et } i \neq j$$

La construction propose admet la rgle de preuve semblable suivante :

$$\frac{p_1 \Rightarrow [S_1](q_1), p_2 \Rightarrow [S_2](q_2)}{p_1 \wedge p_2 \Rightarrow [S_1 \parallel S_2](q_1 \wedge q_2)}$$

5 Conclusion

Dans cette tude, nous avons propos une mcanisation de la mthode B en Coq et PVS . Notre approche est mi-chemin entre un plongement syntaxique et smantique et un plongement smantique de la mthode B. D'une part, elle permet d'envisager la preuve de mta proprits relatives aux substitutions gnralises. Les preuves de ces proprits sont alors ralises dans le contexte de la logique d'ordre suprieur de Coq ou PVS. D'autre part, elle permet aussi d'envisager des dveloppements B. Il est noter que bien que l'approche ne soit pas tout fait transparente, la plupart du travail de traduction est ralis par l'outil PBS. De plus, les choix d'implantation font que les entits manipules au niveau d'une preuve sont identiques celles manipules au niveau du texte source.

Notre travail peut tre tendu dans plusieurs directions. A titre d'exemple citons :

- La validation d'extensions de la mthode B : nous avons labor ici une nouvelle construction $\parallel$ et montr comment sa validation peut tre mene. Dans le mme ordre d'ide, la validation des extensions pour exprimer des proprits dynamiques[AM98] peuvent tre envisages.
- La validation d'outils B comme par exemple les gnrateurs d'obligations de preuve et les gnrateurs de code.

Références

[Abr91] J.-R. Abrial. The B method for large software: Specification, design and coding (abstract). In Soren Prehn and Hans Toetenel, editors, *Proc. Formal Software Development Methods (VDM '91)*, volume 552 of *LNCS*, pages 398–405, Berlin, Germany, October 1991. Springer.

[Abr96a] J.-R. Abrial. *The B-Book – Assigning Programs to Meanings*. Cambridge University Press, 1996.

[Abr96b] J.-R. Abrial. *Mathematical Methods in Program Development*, chapter Set-theoretic Models of Computations. Advanced Studies Institute. Institut für Informatik Technische Universität München, Marktoberdorf, August 1996.

[AM98] J.-R. Abrial and L. Mussat. Introducing dynamic constraints in B. In *2nd Conference on the B Method*, April 1998.

[AO91] K.R. Apt and E.-R. Olderog. *Verification of Sequential and Concurrent Programs*. Texts and monographs in computer science. Springer-Verlag, 1991.

[B+97] J. Bicarregui et al. Formal methods into practice: Case studies in the application of the B method. *IEEE Proc. Software Engineering*, 144(2), 1997.

[BBC+97] B. Barras, S. Boutin, C. Cornes, J. Courant, J.C. Filliatre, E. Giménez, H. Herbelin, G. Huet, C. Muñoz, C. Murthy, C. Parent, C. Paulin, A. Saïbi, and B. Werner. The Coq Proof Assistant Reference Manual – Version V6.1. Technical Report 0203, INRIA, August 1997.

[BF95] J.-P. Bodeveix and M. Filali. On the refinement of symmetric memory protocols. In *Higher Order Logic Theorem Proving and Its Applications*, volume 971 of *Lecture Notes in Computer Science*, pages 58–74. Springer-Verlag, September 1995.

[BGG+92] R. Boulton, A. Gordon, M.J.C. Gordon, J. Herbert, and J. van Tassel. Experience with embedding hardware description languages in HOL. In *Proc. International Conference on Theorem Provers in Circuit Design: Theory, Practice and Experience*, pages 129–156, Nijmegen, June 1992. IFIP TC10/WG 10.2, North-Holland.

[Gor93] M.J.C. Gordon. *Introduction to HOL: A Theorem Proving Environment*. Cambridge University Press, 1993.

[Hey97] B. Heyd. *Application de la théorie des types et du démonstrateur Coq à la vérification de programmes parallèles*. Thèse de doctorat, Université Henri Poincaré Nancy I, December 1997.

[Muñ98] C. Muñoz. PBS: Support for the B-method in PVS. Submitted as CSL-SRI report, 1998.

[ORS92] S. Owre, J.M. Rushby, and N. Shankar. PVS: A prototype verification system. *Lecture Notes in Computer Science*, 607, 1992.

[Ste90] P. Stenstrom. A survey of cache coherence schemes for multiprocessors. *Computer*, 23(6): 11–25, June 1990.

[vW90] J. von Wright. *A lattice-theoretical basis for program refinement*. PhD thesis, Abo Akademi Finland, 1990.