A Formalization of the B-Method in Coq and PVS[⋆]

Jean Paul Bodeveix[1], Mamoun Filali[1], and César A. Muñoz[2⋆⋆]

[1] IRIT-Université Paul Sabatier
118 Route de Narbonne, F-31062 cédex, Toulouse, France
{bodeveix,filali}@irit.fr
[2] SRI International
333 Ravenswood Ave, Menlo Park, CA 94025, USA
munoz@csl.sri.com

Abstract. We formalize the *generalized substitution* mechanism of the B-method in the higher-order logic of Coq and PVS. Thanks to the dependent type feature of Coq and PVS, our encoding is compact and highly integrated with the logic supported by the theorem provers. In addition, we describe a tool that mechanizes, at the user level, most of the effort of the encoding.

1 Introduction

In recent years, important work has been done in the design and implementation of general specification languages and theorem provers. The concretization of these efforts is illustrated in systems like HOL [Gor93], Coq [BBC+97], and PVS [ORS92]. These systems provide several automation tools, but they lack methodologies to handle the global process of software construction.

The B-method [Abr96a] is a formal method for software development. It originated in Abrial's work [Abr91] in the 1980s and continues to be developed by industrial and academic working groups. B provides a uniform notation to specify, design, and implement systems. The methodology emphasizes the structural level of software development.

In this paper we are interested in an embedding of the B-method into the higher-order logic of Coq and PVS. Roughly speaking, there are two general techniques of embedding the specialized language of a methodology into a general specification language. These techniques are referred to as *deep* and *shallow* embedding [BGG+92]. In the deep embedding approach, the language framework of the methodology is fully formalized as an object in the logic of the specification language. In this case, it is possible to prove meta-theoretical properties of the

⋆ To be considered for the Foundations and Methodology stream. Mini-Track B-Method. Contact Author: César A. Muñoz. Phone: +1 (650) 859 2784. Fax: +1 (650) 859 2844

⋆⋆ This work was supported by National Science Foundation grant CCR-9712383.

embedded language, but the proof of properties for a particular application usually requires a painful codification. In the shallow approach, there is a syntactic translation of the objects of the source language into semantically equivalent objects in the target language. In this case, meta-theoretical properties cannot be stated, but codification of particular applications is simpler.

Our approach is halfway between a deep and a shallow embedding. It formalizes the generalized substitution mechanism of the B-method inside the logic of Coq and PVS, and thus we can reason about the method in a certain meta-level. Moreover, we provide a compact encoding highly integrated with the logic of Coq and PVS. For instance, some logical elements of the B-method are lifted to the dependent type theory of Coq and PVS. In the case of PVS, we have mechanized our embedding in a way that the Abstract Machine Notation can be used as a layer over the PVS specification language.

This paper is organized as follows. In Section 2, we introduce the B-method. In Section 3, we present our formalization in Coq and PVS. In Section 4, we describe the architecture of the software component that implements our embedding, and we present two case studies. Related work and future research are discussed in Section 5.

2 An Overview of B

B is a state-oriented method which covers the complete life cycle of software development. It provides a uniform language, the Abstract Machine Notation, to specify, design, and implement systems. A usual development in B consists of an abstract specification, followed by some refinement steps. The final refinement corresponds to an implementation. The soundness of the construction is enforced by obligation proofs at each step of the development.

A specification in B is composed of a set of modules called *(abstract) machines*. Each machine has an internal state, and provides services allowing an external user to access or modify its state. Syntactically, a machine consists of several clauses which determine the static and dynamic properties of the state.

Consider the abstract machine `simple` of Figure 1, which specifies a simple system that stores a value and provides services to read and overwrite this value.

The machine `simple`, rather than specifying a system, specifies a family of systems having the same abstract properties with respect to the parameter `T`. In B, a parameter can be a scalar or nonempty abstract set. By convention, a parameter starting with an uppercase letter is an abstract set. The clause VARIABLES defines the state of the machine. In this case, we use only a variable `value`. The clause INVARIANT constrains the domain of that variable. It states that `value` is a member of `T`. Note that at this stage of the development the domain of the stored value is abstract. We just assume that it is nonempty and finite. The initial state of the machine, which must satisfy the invariant, is specified in the initialization clause. In this example, the variable `value` is initialized with any element of `T`.

The services provided by the machine are specified in the clause OPERATIONS. In this case, we specify one operation to overwrite the value — `write`

```
MACHINE simple(T)
  VARIABLES
    value

  INVARIANT
    value : T

  INITIALIZATION
    ANY x WHERE x:T THEN
      value := x
    END;

  OPERATIONS
     write(v) ≜
       PRE v:T THEN
         value := v
       END;

     v ⟵ read ≜
       v := value

END
```

Fig. 1. Abstract Machine `simple`

— and another to access it — `read`. To specify operations, B uses a mechanism of *generalized substitutions*. B defines six basic generalized substitutions: the well-known skip, multi-assignment, selection, bounded choice and unbounded choice, and the preconditioned substitution. A generalized substitution acts as a predicate transformer. For example, the generalized substitution

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

corresponds to the predicate transformer

$$[v \in T \mid value{:=}v]$$

which is defined for any predicate P as follows:

$$[v \in T \mid value{:=}v]P \quad \Leftrightarrow \quad v \in T \wedge [value := v]P$$

In Section 3, we give a more formal presentation to this notion.

Some other clauses allow the introduction of constants (`CONSTANTS`) and different kinds of assumptions (`CONSTRAINTS`). Large software development is supported by several composition mechanisms, for instance, `INCLUDES`, `SEES`, and `IMPORTS`. These mechanisms give different access privileges to the operations or to the local variables of an external machine.

An abstract specification can be materialized in an implementation by a mechanism of refinement. Assume for example that in a real implementation of our system, a value can be stored in a machine register or in memory.

The refinement `simple_ref` of Figure 2 uses two variables to store the value, one for the memory, `val_mem`, and the other for the register, `val_reg`. The variable `flag` states which of them is used. The invariant of a refinement relates the abstract variables to the concrete ones. From a user's point of view, the services provided by `simple` and `simple_ref` are equivalent.

```
REFINEMENT simple_ref(T)
  REFINES simple

  VARIABLES
    val_mem, val_reg, flag

  INVARIANT
    flag:bool ∧ val_mem:T ∧ val_reg:T ∧
    (flag ∧ val_mem = value) ∨ (¬ flag ∧ val_reg = value)

  INITIALIZATION
    ANY x,f WHERE x:T ∧ f:bool THEN
      flag,val_mem,val_reg := f,x,x
    END;

  OPERATIONS
     write(v) ≜
       PRE v:T THEN
         CHOICE
           flag,val_mem := true,v
         OR
           flag,val_reg := false,v
         END
       END;

     v ⟵ read ≜
       IF flag THEN v := val_mem
       ELSE v := val_reg
       END
END
```

Fig. 2. A Refinement of the Abstract Machine `simple`

The soundness of a machine in B is given by *proof obligations* which verify that

– There exists an instantiation for the parameters, sets, and constants satisfying the machine constraints.

– The initial state satisfies the invariant.
– The invariant is preserved by the operations.

Validity of composition and refinement is also guaranteed by proof obligations.

3 Formalization of the B Semantics in Coq and PVS

Coq [BBC+97] is a proof assistant developed by the Coq Project at INRIA (Rocquencourt). The specification language of Coq is based on an intuitionistic higher-order logic called *Calculus of Inductive Constructions* [Wer94]. PVS [ORS92] is a general verification system developed by the Formal Methods group at SRI International. Both Coq and PVS are enhanced with a dependent type theory. However, in contrast to Coq, the type theory of PVS does not support an effective method of type checking. Hence, the type checker of PVS generates Type Correctness Conditions (TCCs) that must be discharged by the user. Specifications in Coq and PVS are structured in modules — *sections* in Coq and *theories* in PVS — which package mathematical and logical objects.

In the following, we give the encoding of most of the B constructs in either Coq or PVS. We give them in both only when they are quite different; otherwise, since the PVS syntax is closer to B, we give the PVS encoding.

3.1 Basic notation

The B-method has departed from the before-after relation introduced by the generalized substitution mechanism. While the usual binary relation for modeling nondeterministic commands does not distinguish between non termination and halting, the preconditioned substitution specifies terminating points. Furthermore, a computation starting from a state not satisfying the precondition predicate leads anywhere. Consequently, the basic computational model of B, a substitution, can be represented by a dependent record consisting of a predicate `pre` and a binary relation `rel` such that[1]

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

The substitution `S` can also be characterized by a predicate transformer `[S]` linked to `pre` and `rel` by the following equations:

```
[S](p) = λ e1: pre(S)(e1) ∧ ∀e2: rel(S)(e1,e2) ⇒ p(e2)

pre(S) = [S](TRUE)
rel(S) = λ e1,e2: ¬ ([S](λ e: e ≠ e2))(e1)
```

[1] We lift the usual boolean operators to predicates.

However, it should be noted that not all predicate transformers can be associated to substitutions, a sufficient condition being its monotonicity. In the following, we represent a substitution with `pre` and `rel` since they are more intuitive for the definition of new substitutions.

Formally, a substitution is modeled by a transition between state environments, that is a record containing the fields `pre` and `rel`, and the proof that they satisfy Equation 1. In Coq, we use a dependent record, and in PVS, we use the subtype as a predicate feature.

Substitutions in Coq

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

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

Substitutions in PVS

```
Transition_0: TYPE = [#
  pre: pred[Env],
  rel: [Env,Env -> bool]
#]

Is_Transition: [Transition_0->bool] =
  (λ (tr:Transition_0):
    (∀ ((e1,e2: Env) (¬ pre(tr)(e1)) ⇒ rel(tr)(e1,e2))))

Transition: TYPE = (Is_Transition)
```

3.2 Machine states

Most of the substitutions are independent of the representation of the state space. However, some of the substitutions require knowing that the environment is a set of typed variables (for instance, in the case of the parallel substitution [Section 4.2]). With regard to the typing system underlying Coq, such an environment is easily encoded [Hey97] through a function mapping each variable to its type. In PVS, the encoding is not as straightforward as in Coq, since in PVS, types are not terms.

Machine states in Coq In Coq, the definition of an environment `Env` parameterized by a set of variables `Id` and a mapping function `typeof` is declared as follows:

```
Section modele.
  Variable Id: Set.
  Variable typeof: Id->Set.
  Definition Env := (i:Id)(typeof i).
  ...
End modele.
```

For instance, let us consider the state consisting of `x:nat; b:bool`. To instantiate the definitions introduced in the `modele` section, we introduce the following declarations:

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

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

It should be noted that such a definition is valid in Coq since types are terms and, consequently, we can elaborate type expressions.

Machine states in PVS As types are not terms, we cannot use a similar encoding of `typeof` as in Coq. However, we can use the notion of subtype as predicate introduced by PVS. Thus, for each machine we synthesize a new supertype from which the types of the machine variables will be derived as subtypes.

In PVS, the module defining the type `Env` is declared as follows, where `Val` represents the type union of the state variables, and `typeof` is a function from variables to subsets of `Val`:

```
modele [Id,Val: TYPE, typeof: [Id -> pred[Val]]]:THEORY
BEGIN
  Env : TYPE = [ i:Id -> (typeof(i))]
  ...
END modele
```

For instance, if we consider the state `x:nat; b:bool`, `Val` is encoded as a datatype with a variant for each variable of the state. Each variant declares a constructor (`d_x, d_b`), a destructor (`v_x,v_b`), and an observer (`d_x?, d_b?`):

```
  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) : pred[Val] = IF i = id_x THEN d_x? ELSE d_b? ENDIF
```

The use of such a state encoding is rather clumsy since we must manipulate constructors and destructors of the union type `Val`. Thus, at the user level, we also introduce a record encoding the state[2] and conversions between the two representations.

With respect to our previous experience in encoding machine states in higher-order logics, it appears that dependent types are necessary; otherwise, either the state space is untyped [vW90] or we must resort to complicated encoding [BF95].

3.3 The basic substitution

The basic substitution, which is in fact a multi-assignment, is defined from a function between environments as follows:

```
COM_PT(f:[Env->Env]): Transition =
  (# pre := TRUE, rel := λ (e1,e2: Env): e2 = f(e1) #)
```

The definition in Coq is similar. However, the fact that the transition verifies Equation 1 must be proved before the definition of the record. In PVS, this condition is generated by the type checker when the record is declared.

3.4 Generalized substitutions

As we have said, the six basic generalized substitutions of B can be defined by the predicates `pre` and `rel`. For instance, we have for the preconditioned substitution noted $P \mid S$ where P is a precondition predicate and S a substitution:

```
pre(P|S) = P ∧ pre(S)
rel(P|S) = λ e1,e2: P(e1) ⇒ rel(S)(e1,e2)
```

We prove that

```
[P|S] = λ p: P ∧ [S](p)
```

In the case of the parallel substitution, B proposes [Abr96b] a $\|_B$ construct whose purpose is mainly to act as a structuring tool. We have introduced this operator by the following PVS definition:

```
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)))
```

[2] Such a record encoding cannot be stated at the meta level.

```
                  ⇒
                  (rel(trl)(PROJ_1(elr1),PROJ_1(elr2)) ∧
                   rel(trr)(PROJ_2(elr1),PROJ_2(elr2))))
    #)
```

where `PROJ_1(e1,e2) = e1`, and `PROJ_2(e1,e2) = e2`.

An interesting property of this parallel construct is its behavior with respect to the sequence

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

The intuitive reason of the validity of such a property is that the F_i substitutions and the substitutions F_i and G_i are built on disjoint state spaces. Thus, they do not interfere.

3.5 Invariants

Machine invariants are introduced as a constraint predicate `Inv` over the state space. In PVS, such a constraint is expressed using the subtype as a predicate facility of its underlying type theory:

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

In Coq, we use a dependent type to introduce the proof of the constraint over the state space.[3]

```
Record Typed_Env: Set := {
  env:> Env;
  ctr: (Inv env);  (*  proof of the constraint *)
}.
```

We note again the different approaches of Coq and PVS with respect to the introduction of such constraints in a dependent product.

- The type-as-predicate facility of PVS leads to undecidability and, consequently, the PVS type checker produces proof obligations to check subtyping. While it is up to the user to discharge such obligations, it should be noted that the powerful decision procedures of PVS are usually quite sufficient.
- In Coq, the proof (`ctr`) that the state space verifies the constraint `Inv` must be provided *before* the construction of the "typed" state space. However, the `Refine` facility of Coq can be used to postpone this proof.

Now, in PVS, we define the predicate `Is_Tr_Inv` typing invariant preserving transitions as follows:

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

[3] The notation `:>` also introduces an implicit conversion from `Typed_Env` to `Env`.

3.6 Refinements

Intuitively, a machine M_1 is refined by a machine M_2 if the latter can replace the former with respect to a machine user. In the context of the B method, machine refinement is defined in terms of substitution refinement as follows:

– Machines M_1 and M_2 have the same signature (set of operations).
– There is a refinement relation between the respective implementations by M_1 and M_2 of any external substitution (in fact, a program) built from their common signature.

In B, the refinement between two substitutions is defined as an implication between the corresponding predicate transformers.

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

It should be noted that the basic property that bridges the gap between machine refinement and substitution refinement is the monotonicity of each basic substitution with respect to the refinement. Since a program is a composition through basic substitutions of the operations of the common signature, the refinement between the respective implementations of the operations is a sufficient condition. Consequently, when introducing a new operator, we must check the monotonicity property. In Section 4.2, we show that the new $||$ operator is actually monotone.

We have proved in PVS and Coq the monotonicity of the operators with respect to refinement. For instance, the monotonicity of the preconditioning operator is expressed in PVS as

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

3.7 Summary

Table 1 summarizes the key aspects of the encoding of B in Coq and PVS.

This study has led us to the following conclusions:

– Dependent types and subtyping allow a simple encoding of abstract machines. Coq lacks the subtyping mechanism, but its dependent type theory is more powerful than the PVS one, notably with respect to type constructors.
– PVS type obligations match the concept of B proof obligations. In Coq, theorems must be stated explicitly. However, the use of the `Refine` tactic could help.
– A more technical aspect concerns data structure updates with copy. For this purpose, PVS proposes the `WITH` construct, which allows the copy of structures, except the specified substructures for which a new value is given. Coq lacks such a construct. However, grammar extension features of Coq should allow the definition of the construct.

Table 1. Encoding of B in Coq and PVS

B features	Coq	PVS
machine	section	theory
state	functional dependent type	functional dependent type + abstract data type
substitution	constrained record {pre,rel}	constrained record {pre,rel}
invariant	constrained states and substitutions	constrained states and substitutions
refinement	relation between transitions	relation between transitions
proof obligation	stated theorem	type correctness condition

4 Case Studies

To effectively exploit the formalization, we have extended the front-end tool PBS [Muñ98] to support our embedding. The PBS system allows the use of the Abstract Machine Notation inside PVS. Moreover, we have written some tactics in the PVS theorem prover in order to deal with the proof obligations generated by the method. Figure 3 illustrates the general architecture of the system.

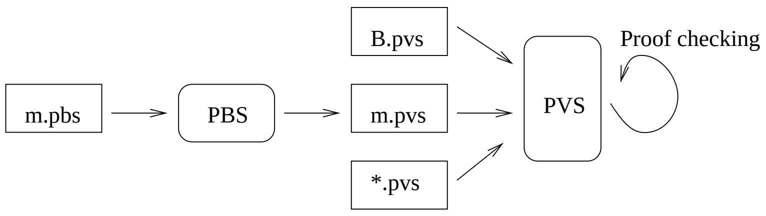

Fig. 3. The PBS tool

PBS works like a compiler. It takes as input a file *m.bps* containing an abstract machine and generates its corresponding embedding as a PVS theory in the file *m.pvs*. The file *B.pvs* contains our formalization of the B-method. As the diagram suggests, a PBS machine can import any PVS theory. Therefore, the user is not constrained to a limited set of data types.

Some features of PBS deserve some comments. First of all, the syntax of the Abstract Machine Notation supported in PBS is based on the PVS syntax. As a result, it does not match the syntax of B defined in [Abr96a]. In addition, PBS uses a typing approach of the Abstract Machine Notation. Instead of a specification language based on the set theory proposed by Abrial in [Abr96a], the PBS notation is based on the higher-order logic and type theory of PVS.

The PBS system can be used as well at the development level as at the meta level, as we show through two case studies. In the first case study, we relate our experience in the development of so-called "atomic memory" protocols, and in the second case study, we consider some alternative semantics for the parallel operator.

Currently, the PBS environment generates PVS source theories. With respect to Coq, we have only encoded the semantics of all the B constructs.

4.1 Development of an atomic memory protocol

We have expressed in PBS the development steps of atomic memory protocols [Ste90] (the PBS sources are given in the appendix). We have found it interesting to express such a development, since atomic memory protocols belong to the class of concurrent parameterized programs, and their validation is an active research topic.

We have appreciated the use of the B methodology for expressing developments. With respect to our previous experience on the validation of such problems [BF95] in HOL, the relevant aspect is the structuring of development steps by means of operations, invariants, constraints, and refinements.

With respect to concurrency, we note the lack of an indexed parallel construct expressing the replication of substitution. However, it is possible to simulate such a feature with B lambda expressions.

The invariant feature of B is sufficient for naturally expressing most safety properties. It is interesting to note that B extensions have been proposed for specifying new features such as liveness and more generally temporal properties [AM98]. We believe that the PBS environment can be used for

- validating the semantics of proposed constructs and the relations between these constructs and the existing ones.
- experimenting with new developments, using the proposed extensions.

It is now widely acknowledged that the use of a given assistant theorem prover (e.g. Coq, PVS, HOL, etc) is a matter of taste. The PBS approach can be considered as a unifying one: we can envision developments where the proofs of an invariant or a refinement can be done in either Coq or PVS or in any other tool offering the basic concepts we rely on (table of Section 3.7).

4.2 Alternative semantics for the || operator

The parallel construct currently proposed for B has the following signature:

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

Since the state spaces of the components are distinct, it does not allow any sharing. Consequently, the mapping onto shared memory concurrent architectures is not well supported. We propose an alternative semantics for the $\|$ construct, allowing some sharing. The proposed semantics is based on the definition of the variables that can be changed by a substitution, which we call its *support*. For a simple substitution `x:= e` the support is defined as the singleton {x}. For all the other substitutions we define the support as the union of its components. For example, the support of `x:=e` $\|$ `y:=f` is defined as the set $\{x, y\}$.

To define the new semantics of the parallel substitution, we introduce a new transition type which "inherits" the first transition type and owns the new field `support` for representing the set of variables which can be modified by the transition. Then, the new transition type is defined as follows:

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

Concerning the support field, it is synthesized by the PBS compiler. For instance, we have

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

Then, we define the new `PAR` operator as follows:

```
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) ⇒ ∃ e2l, e2r:
                     rel(l)(e1,e2l) ∧ rel(r)(e1,e2r) ∧
                     ∀ i: IF i ∈ support(l) ∩ support(r) THEN
                                e2(i) = e2l(i) ∨ e2(i) = e2r(i)
                          ELSIF i ∈ support(l) THEN e2(i) = e2l(i)
                          ELSIF i ∈ support(r) THEN e2(i) = e2r(i)
                          ELSE  e2(i) = e1(i) END
           #)
  #)
```

Such a definition deserves some comments:

- `x := y` $\|$ `y := x` has the usual meaning: `x,y := y,x`.
- `x := x` $\|$ `x := y` is defined and has the intuitive meaning of a nondeterministic assignment to `x`.
- Unlike the $\|_B$ construct, the proposed one does not introduce any new space type. Moreover, the proposed definition is commutative and associative.

- The components of the $\|$ are atomic: only their overall effect is considered, their execution is not interleaved.
- Provided that the support is preserved through refinement, the `PAR` operator is monotonic with respect to refinement. Consequently, as explained in Section 3.6, it can be used as a basic substitution within the B framework.

```
MONOTONE_PAR: THEOREM
  ∀ (U,V,S,T: modele2.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))
```

With respect to the interferences allowed by the model, we can compare the proposed $\|$ with the one (hereafter denoted $\|_{AO}$) proposed in [AO91] for deterministic programs. The latter is defined for two statements S_1 and S_2 when the following conditions hold:

$$\begin{array}{l}\mathrm{change}(S_1) \cap \mathrm{var}(S_2) = \emptyset \\ \mathrm{var}(S_1) \cap \mathrm{change}(S_2) = \emptyset\end{array}$$

where

- $\mathrm{change}(S_i)$ is the set of modified variables by S_i.
- $\mathrm{var}(S_i)$ is the set of variables accessed by S_i.

The previous restrictions were motivated by the quest for a compositionality rule; more precisely, $\|_{AO}$ admits the following proof rule:

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

where

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

In the proposed model, we have the following similar proof rule:

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

5 Related Work and Conclusion

Some other formalizations of the B-method are available in the literature. Chartier has formalized the Abstract Machine Notation of B in Isabelle/HOL [Cha98] by using a deep embedding. One of the aims of Chartier's work is the formalization of tools like proof obligation generators or proof checkers. In the context of PVS, Pratten presents in [Pra95] a tool that generates a PVS representation of abstract machine proof obligations. The goal of Pratten's work is to assist the validation process of B specifications by means of the general theorem prover of PVS. A shallow embedding of the Abstract Machine Notation into a set constraint language has been studied by Tellez-Arenas in [TA98]. The main interest of this approach is to use the logical translation as a prototype of the abstract machines, for example, allowing a test of whether or not some states of a machine are reachable. In [Muñ98], Muñoz proposes a shallow embedding of the Abstract Machine Notation of the B-method in PVS. Rather than translate the notation in PVS, he adds the notation as a layer over the PVS language. This embedding has been implemented in the PBS system.

In this paper, we have proposed an embedding of the B-method in Coq and PVS. Our approach is halfway between a deep and a shallow embedding. We can prove meta-theoretical properties about generalized substitutions, but we can not reason about abstract machines. However, the encoding is highly integrated with the logic supported by the theorem provers. In fact, at the user level most of the work about the encoding can be mechanized, as we have shown in the case of PVS with the PBS tool.

Our work can be extended in several ways, for instance

- Validation of extensions to the B-method. With respect to concurrency in B, we have proposed a $||$ construct and used our framework to validate it. In the same way, we plan to validate extensions for the expression of liveness and temporal properties [AM98].
- Validation of B tools, for instance, code generators, obligation generators, and proof checkers for B.

In the long term, we will pursue this work by the study of the integration of the abstraction technique to the B method. Actually, this technique [BLO98,CGL94,LS97] has been shown to be powerful for the verification of infinite and parameterized systems. The presented framework could be reused for studying the integration of such a technique to the B method. Moreover, this would offer a smooth cooperation between the B development method and model checkers.

References

[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.

[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.

[BLO98] S. Bensalem, Y. Lakhnech, and S. Owre. Computing abstractions of infinite state systems compositionnaly and automatically. In *Computer-Aided Verification (CAV'98)*, volume 1427 of *Lecture Notes in Computer Science*, pages 319–331, Vancouver, BC, Canada, june 1998. Springer-Verlag. http://www.csl.sri.com/~owre/cav98.html.

[CGL94] E.M. Clarke, O. Grumber, and D.E. Long. Model checking and abstraction. *ACM Transactions on Programming Languages and Systems*, 16(5):1512–1542, september 1994.

[Cha98] P. Chartier. Formalisation of B in Isabelle/HOL. In *Proc. Second B International Conference, Montpellier, France*, 1998.

[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.

[LS97] D. Lessens and H. Saïdi. Abstraction of parameterized networks. *Electronic notes in theoretical computer science*, 9:12, 1997. http://www.elsevier.nl/locate/entcs/volume9.html.

[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.

[Pra95] C.H. Pratten. An introduction to proving AMN specifications with PVS and the AMN-PROOF tool. In Henri Habrias, editor, *Proc. Z Twenty Years On - What Is Its Future*, pages 149–165. IRIN-IUT de Nantes, October 1995.

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

[TA98] A. Tellez-Arenas. Set constraints for program validation. Manuscript, 1998.

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

[Wer94] B. Werner. *Une théorie des constructions inductives.* Thèse de doctorat, U. Paris VII, 1994.

A PBS Development Steps of Atomic Memory Protocols

A.1 The atomic memory machine

```
Atomic[NbProc:posnat, Word: TYPE]: MACHINE
BEGIN
  TYPES
    Proc = ‘below(NbProc)’
  VARIABLES
    m : Word
  OPERATIONS
     v: Word <- read(p:Proc) =
        v := m
     write(p:Proc,v:Word) =
       m := v
     skip = SKIP
END Atomic
```

Remark: With respect to the B example given in Section 2, we do not need to give the initialization typing statement, since PBS is typed.

A.2 First refinement

In this first refinement, we introduce cached copies of the memory and their associated states. However, in order to have an abstract description of write-invalidate and write-update protocols, we consider classes of states and express their properties.

```
Atomic_r0[NbProc:posnat, Word: ‘TYPE+’, State:TYPE, invalid:State,
          shared:State, coh_set:‘set[State]’, excl_set: ‘set[State]’]:
  REFINEMENT OF Atomic[NbProc,Word]
BEGIN

  CONSTRAINTS
   ‘    shared /= invalid
    AND NOT member(invalid,coh_set)
    AND NOT member(invalid,excl_set)
    AND NOT member(shared,excl_set)
    AND FORALL (st:State): (st = invalid) OR
                           member(st,coh_set) OR member(st,excl_set)
   ’

  VARIABLES
    state0:  ‘ARRAY[Proc -> State]’
    cache0:  ‘ARRAY[Proc -> Word]’
    m0    :  Word

  INVARIANT
    ‘    FORALL (p:Proc,q:Proc):
          (member(state0(p),excl_set) AND
          (state0(q) /= invalid) => p = q) AND
          ((state0(p) = shared AND state0(q) = shared) =>
           cache0(p) = cache0(q)) AND
          (member(state0(p),coh_set) => cache0(p) = m0)
  '

REFINE_INVARIANT
  `   ((FORALL (p:Proc): cache0(p) /= invalid => cache0(p) = m)
  AND ((FORALL (p:Proc): cache0(p) = invalid) => m0 = m)
  '

INITIALIZATION
  state0 := `LAMBDA(p:Proc): invalid'

OPERATIONS
   v:Word <- read_hit(p:Proc) =
     SELECT `state0(p) /= invalid' THEN
       v := `cache0(p)'
     END

   v:Word <- read_miss_1(p:Proc) =
     SELECT `(FORALL (p:Proc): state0(p) = invalid)' THEN
       ANY coh_state: State WHERE `member(coh_state,coh_set)' THEN
         state0(p) := coh_state || cache0(p):= m0 || v := `m0'
       END
     END

   v: Word <- read_miss_2(p:Proc) =
     SELECT `state0(p) = invalid AND
           (EXISTS (q:Proc): state0(q) = shared)' THEN
         ANY q:Proc WHERE `state0(q) = shared' THEN
       v :=  `cache0(q)'
            || cache0 := `cache0 WITH [(p) := cache0(q)]'
            || state0 := `LAMBDA (r:Proc):
                            IF r = p THEN shared
                            ELSIF state0(r) /= invalid THEN shared
                            ELSE state0(r) ENDIF'
            || m0 := `cache0(q)'
         END
      END

   v : Word <- read_miss_3(p:Proc) =
     SELECT `state0(p) = invalid AND
           (EXISTS (q:Proc): member(state0(q),coh_set))' THEN
         ANY q:Proc WHERE `state0(q) /= invalid' THEN
               cache0(p) := `cache0(q)' || v := `cache0(q)'
            || state0 := `LAMBDA (r:Proc):
                             IF r = p THEN shared
                             ELSIF state0(r) /= invalid THEN shared
```

```
                           ELSE state0(r) ENDIF'
      END
  END

v: Word <- read(p:Proc) =
  CHOICE
    v <- read_hit(p)
  OR
    v <- read_miss_1(p)
  OR
    v <- read_miss_2(p)
  OR
    v <- read_miss_3(p)
  END

write_1(p:Proc,v:Word) =
  SELECT `member(state0(p),excl_set) AND
          NOT member(state0(p),coh_set)' THEN
    ANY s:State WHERE `member(s,excl_set) AND
                       NOT member(s,coh_set)' THEN
            cache0,state0 := `cache0 WITH [(p) := v]',
                             `state0 WITH [(p) := s]'
    END
  END

write_2(p:Proc,v:Word) =
  ANY s:State WHERE `member(s,excl_set) AND
                     NOT member(s,coh_set)' THEN
       cache0,state0 :=
          `cache0 WITH [(p) := v]',
          `LAMBDA (q:Proc): IF (p = q) THEN s ELSE invalid ENDIF'
  END

write_3(p:Proc,v:Word) =
   ANY s:State WHERE `member(s,coh_set)' THEN
        cache0,state0,m0 :=
          `cache0 WITH [(p) := v]',
          `LAMBDA (q:Proc): IF p = q OR state0(q) /= invalid
                            THEN s ELSE invalid ENDIF',
          v
   END

write(p:Proc,v:Word) =
  CHOICE
    write_1(p,v)
  OR
    write_2(p,v)
  OR
    write_3(p,v)
  END
```

```
     flush_1 =
       SELECT ‘EXISTS (p:Proc): member(state0(p),coh_set)’ THEN
           ANY p: Proc WHERE ‘member(state0(p),coh_set)’ THEN
             state0(p) := invalid
           END
       END

     flush_2 =
       SELECT ‘EXISTS (p:Proc): state0(p) /= invalid’ THEN
           ANY p: Proc WHERE ‘state0(p) /= invalid’ THEN
               state0,m0 := ‘LAMBDA (q:Proc):invalid’,‘cache0(p)’
           END
       END

     skip =
       CHOICE
         flush_1
       OR
         flush_2
       END
END Atomic_r0
```