PBS: Support for the B-Method in PVS

César Muñoz*
Computer Science Laboratory
SRI International
333 Ravenswood Avenue
Menlo Park, CA 94025, USA
Email: munoz@csl.sri.com
Tel: +1 (650) 859-2784, Fax: +1 (650) 859-2844

February 23, 1999

Abstract

The B-method is a state-oriented formal method for software development. It provides a uniform language, the *abstract machine notation*, to specify, design, and implement systems. The underlying logic of the method is a set theory with a first-order predicate calculus. On the other hand, PVS is a specification language integrated with a theorem prover. The logical framework of PVS is a higher-order logic with a type system. PVS does not come with a particular built-in software construction methodology.

We show how the abstract machine notation of the B-method can be supported by PVS. We advocate a shallow embedding, but rather than translate the abstract machine notation in PVS, we add the notation as a layer over the PVS language. The embedding has been fully implemented in a front-end tool called PBS.

1 Introduction

Several methods, notations, and tools concur in the goal to produce safe systems. Most of the methodologies emphasize the structural level of software development. In this way, they are supported by a language framework, but they are not very sensitive to the logic used by the language. On the other hand, general specification languages supported by theorem provers are highly integrated with their logic, but they lack methodologies to handle the global process of software construction. In [10], a group of researchers in formal methods pointed out how the integration of different languages is necessary in the handling of different aspects of a system.

*This research was supported by National Science Foundation grant CCR-9509931.

In this paper we are interested in an embedding of a formal method, namely the B-method, into the logic of a verification system, namely PVS.

The B-method [2] originated in Abrial's work [1] in the 1980s and continues to be developed by industrial and academic working groups. It provides a uniform notation to specify, design, and implement systems.

PVS [17] is a verification system developed by the Formal Methods group at SRI International. It provides a rich specification language and a general proof checker.

Roughly speaking, there are two general techniques of embedding the specialized language of a methodology into a general specification language supported by a theorem prover [6]. These techniques are referred to as *deep* and *shallow* embedding. 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 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.

We advocate a shallow embedding, but rather than translate the abstract machine notation into PVS, we add the notation as a layer to the PVS language. In other words, we fit the structural language elements of the B-method around the higher-order logic of PVS. In this way, we focus on the structural support provided by the B notation and on the logic support provided by the PVS specification language.

The contributions of this work point in two directions. In one sense, PVS is enhanced with a methodology for software development and a notation for specifications (those of B), and in the other sense, the abstract machine notation is enriched with a fully typed specification language and a powerful theorem prover (those of PVS).

This paper is organized as follows. In Section 2, we introduce the B-method and the abstract machine notation by means of a very simple example. This example is used as a reference throughout the paper. In Section 3, we show how we support the B-method in PVS. In Section 4, we give the semantics of our notation in terms of the PVS type theory. In Section 5, we describe the architecture of a software component called PBS that implements our embedding. Section 6 presents some related work and summarizes our own work.

2 An Overview of B

B is a state-oriented method mainly intended to model sequential systems. A system in B is described by a set of collaborating entities. Each entity has an internal state and provides some services. The behavior of the system is specified by defining how the state of each entity can be modified by the services.

In the terminology of B, an entity is an *abstract machine*, and the services provided by each machine are called *operations*. The information defining the state of a machine is contained in local *variables*, and general requirements about the state of a machine can be expressed by a proposition called *invariant*.

Let us explain all these concepts by means of a simple example written by S. Buxton.[1] Consider the problem of managing a golf club. For each member of the club we store information about his handicap, and only persons having a certain maximum handicap can be accepted as members of the club. The capacity of the club is limited. The services that we will model are to add a person as a member, and to provide a list of members with a handicap less than a given value.

This simple system can be described in B by means of a single machine representing the club entity. Consider the machine of Figure 1.

To have more general specifications, machines can be parameterized. In this case, for example, rather than specify one golf club, we specify a family of clubs having the same abstract properties, but differing in the maximum number of members considered and the maximum handicap required to be a member of the club.

The constraint section of the machine describes the logical properties of the parameters. The theoretical framework of the abstract machine notation is a first-order predicate calculus with a simplification of the classical set theory. Thus, the constraint `maxmembers:NAT & maxhandicap:NAT` must be understood as an ASCII notation of the mathematical proposition ($\mathtt{maxmembers} \in \mathtt{NAT} \wedge \mathtt{maxhandicap} \in \mathtt{NAT}$) where `NAT` is the pre-defined set of natural numbers.

At this stage of the development, the nature of the set of persons is not important. Hence, in the set section of `golfclub` we declare the set `PERSON` as an abstract set supposed to be finite and nonempty. In the variable section of the machine, two state variables are declared: `members` and `handicaps`. The former contains the club members and the latter the handicaps of each member.

The invariant section describes the general property of the machine. In this case it states that `members` is a subset of `PERSON` (in B, `<:` is the ASCII notation of $\subseteq$), `handicaps` is a total function from members to the range $0..\mathtt{maxhandicap}$, and the cardinality of `members` is less than or equal to `maxmembers`.

The initialization section fixes the initial values of the variables, that is, `members` and `handicaps` are empty sets. Note that in set theory a function is simply a set of pairs.

The services provided by the machine are specified in the operation section. Operations can be parameterized and are of two kinds: procedures that modify the internal state of the machine, such as `add_member`, and functions that in addition return a value, such as `inquire_low_handicaps`. As is usual in state-oriented methods, operations are specified by constraining the input and output states. However, in contrast to other methods, B uses pre-conditions and (generalized) substitutions as post-conditions instead of before-after predicates. For instance, the pre-condition of `add_member(m,h)` is $\mathtt{m} \in \mathtt{PERSON} \wedge 0 \leq \mathtt{h} \leq$

[1]This example is part of the B resources provided by the University of Teesside `http://www-scm.tees.ac.uk/bresource/welcome.html`.

```
MACHINE
    golfclub (maxmembers,maxhandicap)

CONSTRAINTS
    maxmembers : NAT &
    maxhandicap : NAT

SETS
    PERSON

VARIABLES
    members, handicaps

INVARIANT
    members <: PERSON &
    handicaps : members --> 0 .. maxhandicap &
    card(members) <= maxmembers

INITIALIZATION
    members, handicaps := {},{}

OPERATIONS

    add_member (m, h) =
        PRE
            m : PERSON &
            h : 0 .. maxhandicap &
            card(members) < maxmembers &
            m /: members
        THEN
            members := members \/ {m} ||
            handicaps := handicaps \/ {m |-> h}
        END;

    m <-- inquire_low_handicaps (h) =
        PRE
            h: NAT
        THEN
            m := { mb | mb: members & handicaps(mb) < h }
        END
END
```

Figure 1: Abstract Machine `golfclub`

maxhandicap $\wedge$ card(member) $<$ maxmembers $\wedge$ m $\notin$ members, and the postcondition is the proposition that results by applying the parallel substitution [members := members $\cup$ {m}, handicaps := handicaps $\cup$ {(m, h)}] to the invariant. The expressiveness of the two mechanisms, before-after predicates and generalized substitutions, is equivalent, but the latter gives a more algorithmic look to the specifications. Notice that a generalized substitution does not imply a constructive specification; for instance, the function inquire_low_handicaps is not constructive. There are several kinds of substitutions, and most of them are like the usual programming language control structures. However, they are only syntactic sugar for generalized substitutions.

The consistency of a machine can be checked by verifying some properties, which in B terminology are called *proof obligations*. These proof obligations 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.

Large software development is supported by using several composition mechanisms: INCLUDES, SEES, and IMPORTS. These mechanisms give different access privileges to the operations or to the local variables of an external machine. In this way, it is possible to build incrementally complex machines by using previously defined ones.

An orthogonal mechanism of composition is that of refinement, which transforms an abstract specification into an implementation. B machines can be of three kinds: abstract machines, refinements, and implementations. The initial specification of a problem is given by an abstract machine. Refinements allow data reification; for example, sets become lists, abstract sets become enumerations, and so forth. The final refinement is called implementation and it is supposed to be a program. Thus, by using the unified notation of machines, B supports the complete life cycle of software development. Validity of compositions and refinements is guaranteed by checking additional proof obligations.

Figure 2 shows a refinement of the machine golfclub (some details have been omitted). The refinement golfclub_ref inherits the parameter and constant properties of the machine golfclub. In this refinement, the set of members is represented by a finite injective sequence (members_ref), so a person can appear only once in the sequence; the handicap function is represented by a finite sequence (handicaps_ref). The sets iseq and seq are, respectively, the set of finite injective sequences and the set of finite sequences. The invariant relates the variables of the abstract machine golfclub with the variables of the refinement golfclub_ref. It states that the set members is the range of the sequence members_ref and the values contained in the sequence handicaps_ref correspond to the values given by the function handicaps. If r is a binary relation, dom(r) and ran(r) stand, respectively, for the domain and range of r, and !x.P is the universal quantification of x in P.

```
REFINEMENT
  golfclub_ref

REFINES
  golfclub

VARIABLES
  members_ref, handicaps_ref

INVARIANT
  members_ref : iseq(PERSON) &
  handicaps_ref : seq(NAT) &
  members = ran(members_ref) &
  !i . (i : dom(members_ref) =>
     handicaps_ref(i) = handicaps(members_ref(i)))

INITIALIZATION
  members_ref, handicaps_ref := {},{}

OPERATIONS

  add_member(m,h) =
    ...

  m <-- inquire_low_handicaps(h) =
    ...
END
```

Figure 2: Refinement of the Abstract Machine `golfclub`

A refinement defines at least the same operations that are defined by the machine being refined. The proof obligations for a refinement guarantee that from the user's point of view the operations provided by the refinement simulate those provided by the abstract machine. The opposite it not necessarily true.

3 Abstract Machine Notation in PVS

PVS is a general verification system, that is, it consists of a specification language integrated with a formal proof checker (or theorem prover). The specification language of PVS is based on a higher-order logic enhanced with a rich type system (dependent types and subtypes are supported). The type system is so expressive that type checking is undecidable. Thus, the type checker generates proof obligations that must be discharged by the user. This task can be achieved by using the several decision procedures and proof automation tools provided by the system.

PVS specifications are structured in theories. A theory can be parameterized and it consists of constant declarations (such as uninterpreted types, variables, abstract constants,...), constant definitions (such as types and subtypes, functions, predicates,...), and logical objects (such as axioms, theorems, conjectures,...). A PVS theory can use other theories by means of an importing mechanism.

PVS does not impose a particular style of specification, and the specification language supports most of them, such as axiomatic, declarative, and algorithmic. For example, the PVS theory in Figure 3 is a very naïve axiomatic specification of the golf club problem.

The theory is parameterized by `maxhandicap`. We declare types to represent persons, valid handicaps, and the golf club as follows: `PERSON` is a uninterpreted and nonempty type (this is the meaning of `TYPE+` in PVS), `HANDICAPS` is a subtype of the type `nat`, and `GolfClub` is a functional type from `PERSON` to `HANDICAPS`. The functions `add_member` and `inquire_low_handicaps` are abstract, but their behavior is defined by the axioms `inv_add_member` and `inv_inquire_low_handicaps`. The `WITH` construct is a function updating mechanism provided by PVS.

The theory `golfclub_th` does not take into account the constraint about the capacity of the golf club. Also, since functions in PVS are total, a golf club function is defined for any person, even for those who are not members of the club. To cope with these requirements, we have to refine the specification.

In the Tutorial Introduction to PVS [11] there are several examples showing how to refine specifications in PVS. In [20], Rushby, Owre and Shankar describe the "predicate subtyping" feature of PVS as a general mechanism to write specifications. We go one step further, based on the "predicate subtyping" capability of PVS, we enhance the PVS specification language with the abstract machine notation of the B-method.

The specification of Figure 4 represents a golf club machine in our enhanced version of PVS. It looks very similar to the one previously defined in Figure 1,

```
golfclub_th [maxhandicap:nat] : THEORY
BEGIN

  PERSON : TYPE+

  HANDICAPS : TYPE = subrange(0,maxhandicap)

  GolfClub : TYPE = [PERSON -> HANDICAPS]

  add_member : [GolfClub, PERSON, HANDICAPS
                -> GolfClub]

  inv_add_member : AXIOM
    FORALL (g: GolfClub, m: PERSON, h: HANDICAPS) :
      add_member(g,m,h) = g WITH [(m) := h]

  inquire_low_handicaps : [GolfClub, HANDICAPS
                           -> setof[PERSON]]

  inv_inquire_low_handicaps : AXIOM
    FORALL (g: GolfClub, h:HANDICAPS) :
      FORALL (m: PERSON) :
        (g(m) < h <=> member(m,inquire_low_handicaps(g,h))

END golfclub_th
```

Figure 3: Golf Club Specification in PVS

```
golfclub_pvs [maxmembers:nat,maxhandicap:nat]: MACHINE
BEGIN
  PVS
     `IMPORTING finite_sets, Empty'

  TYPES
     PERSON
     HANDICAPS = `subrange(0,maxhandicap)'

  VARIABLES
     members : `finite_set[PERSON]'
     handicaps : `[(members) -> HANDICAPS]'

  INVARIANT
     `card(members) <= maxmembers'

  INITIALIZATION
     members := `emptyset[PERSON]' ||
     handicaps := `emptyfunc[PERSON,HANDICAPS]'

  OPERATIONS

     add_member (mm:PERSON, hh:HANDICAPS) =
       PRE
         `card(members) < maxmembers AND
          NOT member(mm,members)'
       THEN
         members := `add(mm,members)' ||
         handicaps := `handicaps WITH [(mm) |-> hh]'
       END

     m <- inquire_low_handicaps(hh:HANDICAPS):`finite_set[PERSON]' =
       m := `{x:PERSON | members(x) AND handicaps(x) < hh}'

END golfclub_pvs
```

Figure 4: Golf Club Machine in an Enhanced PVS

modulo some syntactical conventions that we have changed to be compatible with the PVS syntax. Expressions delimited by quotation marks are regular expressions in PVS.

There is a major difference between the specifications in Figures 1 and 4: PVS machines are based on the higher-order logic and type theory of PVS. In particular, variables and parameters must be explicitly typed. In type theory functions are not interpreted as binary relations but as computational objects. Moreover, functions in PVS are total; however, partial functions can be represented by using subtypes and dependent types.

In PVS machines we use a clause `TYPES` rather than the original clause `SETS` of B. Consider the machine `golfclub_pvs`. The expression `maxmembers:nat` does not represent a membership relation, but a typing judgment where `nat` is a type. Moreover, from our point of view, `PERSON` and `members` represent different things. The first one is an abstract entity, used to classify some objects, while the second one is a real container. Thus, `PERSON` is declared as an uninterpreted type, while `members` is declared as a set of elements of `PERSON`. PVS machines enforce a clear distinction between typing and logical requirements.

4 Semantics and Validity of PVS Machines

Internally, a PVS machine is translated to a PVS theory, where the parameters and types of the machine become parameters and types of the theory. The state of a PVS machine is encoded in the functional style of PVS as follows. The variables of the machine define a record type, called the *general type*. Each field of the record corresponds to a variable of the machine. The invariant of the machine is expressed in the PVS theory as a subtype of the general type. In this way, the mutual dependence between the variables given by the constraints is handled by the dependent type mechanism of PVS.

The general type defined for `golfclub_pvs` is

```
golfclub_pvs_Type : TYPE = [#
  members:finite_set[PERSON],
  handicaps:[(members) -> HANDICAPS]
#]
```

Record fields in PVS are declared between `[#` and `#]` brackets. Instances of a record type are given between `(#` and `#)` parentheses and, in the same way as for functions, records can be overridden by using the `WITH` construct.

Notice that handicaps are defined only for members. This restriction is handled by the type of the field `handicaps`, that is, `[(members) -> HANDICAPS]`. We recall, that if e represents a set expression in PVS, (e) is a notation for the type of the elements in e.

The invariant of the machine is handled by the following type:

```
golfclub_pvs : TYPE = { x : golfclub_pvs_Type |
  card(members(x)) <= maxmembers }
```

Thus, only the records satisfying the restriction about the cardinality of the value contained in `members` are admissible. This restriction is enforced by the type checker.

A machine operation is translated by a function in PVS. This function receives an additional parameter with the record state satisfying the invariant and pre-condition, and returns a record state satisfying the invariant and post-condition. If the operation returns a value, the function in PVS returns in addition to new record state, the output value. For example, the operation `add_member` is interpreted as a PVS function with the following signature:

```
add_member(a_club:golfclub_pvs)
          (m:PERSON, h:HANDICAPS |
           card(members(a_club)) < maxmembers AND
           NOT member(m,members(a_club)))
          : golfclub_pvs
```

and the operation `inquire_low_handicaps` is interpreted as a PVS function with the following signature:

```
inquire_low_handicaps(a_club:golfclub_pvs)(hh:HANDICAPS) :
    [golfclub_pvs, finite_set[PERSON]]
```

Once again, notice how the dependent type mechanism has been used to constrain the parameters of the function according to the specification of the operation.

Generalized substitutions are interpreted as PVS expressions dealing with record field overriding, function updating, and set operations. However, we do not support the following substitutions:

- `WHILE P DO S INVARIANT I VARIANT V`. The semantics of this substitution is that of loops in imperative programming languages. The proposition `I` must hold in each iteration, and the value `V` is the termination argument. Since the specification language of PVS follows the functional paradigm, we think that the mechanism of recursive functions provided by the language is better adapted to our PVS machine notation.

- `SELECT P THEN S`. If `P` is a predicate and `S` is a generalized substitution, the informal semantics is that `S` will be applied only if `P` holds. This substitution cannot be handled in PVS because it needs explicit control of the execution, which is not provided by the specification language of PVS. However, in most cases, it can be simulated by the usual conditional mechanisms of PVS.

The main advantage of this interpretation of PVS machines as PVS theories is that the consistency of a machine corresponds to the type correctness of a theory. Therefore, the proof obligations to be checked are just the proof obligations generated by the type system, and so it is possible to use all the automation tools and tactics provided by the PVS proof checker. In particular, the proof obligations guarantee that the initial state satisfies the invariant and that the invariant is preserved by the operations. A similar approach has been explored in the case of the embedding of VDM-SL specifications in PVS [3,14], and in a reworking of a Z specification in PVS [21].

Proof obligations are also generated about the existence of parameters, sets, and constants satisfying the machine constraints. Certain kinds of compositions are supported by using the importing mechanism of PVS.

Finally, we show how to embed the refinement of machines in PVS. This is done in such a way that the soundness of the refinement is also handled by the type system.

Consider the refinement of Figure 5 (some details have been omitted). As in the case of B refinements, PVS refinements inherit the parameter and constant properties of the abstract machines. Since functions in PVS are total, to constrain the domains of the functional variables `members_ref` and `handicaps_ref`, we have to declare explicitly a variable `num_mem` containing the number of members in the club.

A major difference between `golfclub_ref` and `golfclub_pvs_ref` is that we have two invariants for the refinement. The one introduced by the keyword `INVARIANT` constrains the local variables of the machine, and the one introduced by the keyword `REFINE_INVARIANT` relates the variables of the machine `golfclub_pvs` with the local variables. In this way, `golfclub_pvs_ref` is a machine itself. Actually, this is not the case in B machines, since the invariant of a refinement may contain non-local variables.

Let us show with the golf club example how the validity of the refinement can be handled by typing constraints. First, the refinement relation between the two machines is interpreted by a record type containing all the pairs of abstract machines and refinement machines satisfying the refinement invariant. Let us call `INV_REF` the refinement invariant, so that we consider the following type

```
golfclub_Ref : TYPE = { [#
    ref:golfclub_pvs_ref,
    abs:golfclub_pvs
  #] |  INV_REF(ref,abs) }
```

Now consider, for example, the `add_member` operation. As we have shown, this operation has been interpreted as functions in their respective PVS theories. The idea now is to apply the two functions to the same arguments in two states (one for each machine) satisfying the refinement invariant, and to check that the refinement invariant is preserved by the operations. Let us call `PRE_ABS` the pre-condition of `add_member` in `golfclub_pvs` and `PRE_REF` the pre-condition of `add_member` in `golfclub_ref_pvs`. The refinement relation between the two `add_member` functions is interpreted by the following PVS function:

```
add_member_ref(a_club_ref:golfclub_Ref)
              (m:PERSON, h:HANDICAPS |
               PRE_ABS(abs(a_club_ref))) :
              golfclub_Ref
```

```
golfclub_pvs_ref [maxmembers:nat, maxhandicap:nat]: REFINEMENT OF golfclub_pvs
BEGIN
  TYPES
    MEMBERS = ‘subrange(0,maxmembers)’

  VARIABLES
    num_mem : MEMBERS
    ref_members : ‘[subrange(1,num_mem) -> PERSON]’
    ref_handicaps: ‘[subrange(1,num_mem)-> HANDICAPS]’

  INVARIANT
    ‘injective?[subrange(1,num_mem),PERSON](ref_members) AND
     num_mem <= maxmembers’

  REFINE_INVARIANT
    ‘members =
       {p:PERSON | EXISTS (i:subrange(1,num_mem)) : ref_members(i)=p} AND
     FORALL (i:subrange(1,num_mem)):
             ref_handicaps(i) = handicaps(ref_members(i))’

  INITIALIZATION
  BEGIN
    num_mem := 0 ||
    ref_members := ‘emptyfunc[subrange(1,0),PERSON]’ ||
    ref_handicaps := ‘emptyfunc[subrange(1,0),HANDICAPS]’
  END

  OPERATIONS
     add_member (mm:PERSON, hh:HANDICAPS) =
       PRE
        ‘num_mem < maxmembers AND
         NOT EXISTS (i:subrange(1,num_mem)) : ref_members(i)=mm’
       THEN
         num_mem:=‘num_mem+1’ ||
         ref_members(‘num_mem+1’) :=  mm ||
         ref_handicaps(‘num_mem+1’) := hh
       END

     m <- inquire_low_handicaps(h:HANDICAPS): finite_set[PERSON] =
        m:=‘{x:PERSON | members(self)(x) AND
                        handicaps(self)(x) < (hh:HANDICAPS)}’

END golfclub_pvs_ref
```

Figure 5: Golf Club Refinement in an Enhanced PVS

Table 1: Encoding of the Abstract Machine Notation in PVS

Abstract Machine Notation	PVS
Machine	Theory
Variables states	Record dependent type
Invariant, pre-, post-conditions	Subtype restriction
Generalized substitution	PVS language
Operations	Functions
Loops	Recursive functions
Proof obligations	TCCs

```
(# abs := golfclub_pvs.add_member
                    (abs(a_club_ref))(m,h),
   ref := golfclub_ref_pvs.add_member
                    (ref(a_club_ref))(m,h)
 #)
```

Notice that the above function uses the pre-condition of the abstract machine and not that of the refinement. A type constraint (TCC — Type Correctness Condition — in PVS terminology) generated by the system is

$$(\texttt{PRE\_ABS} \wedge \texttt{INV\_REF}) \rightarrow \texttt{PRE\_REF}.$$

This means that the pre-conditions of refinement operations can be weaker than those of the abstract machine.

In the case of `inquire_low_handicaps` the interpretation forces the values returned by the two functions to be equal.

In Table 1, we summarize the interpretation of abstract machines in PVS.

5 The PBS System

Our embedding of the abstract machine notation has been fully implemented in a front-end tool called PBS. PBS is implemented in the functional programming language Ocaml [19], at it uses the lexer and parser libraries, ocamllex and ocamlyacc, provided by the language.

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*. Figure 6 illustrates the general architecture of the PBS system. As the diagram suggests, a specification in PBS can import any PVS theory. Therefore, the user is not constrained to a limited set of data types.

We have rewritten several examples of abstract machines, in particular from [2, 13, 16]. Table 2 summarizes some of these developments.

Client, Product and Invoice are part of an invoice system developed in [2]. This example provides the basic functionality of a data processing system. Dur-

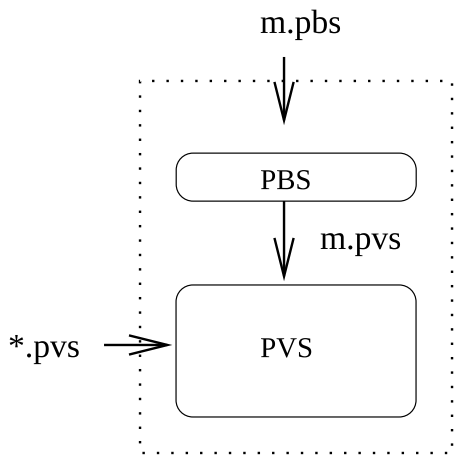

Figure 6: The PBS Tool

Table 2: Metrics of Some Examples

Machine	PBS (in lines)	PVS theory (in lines)	TCCs	Auto proved
Golf club	57	78	12	41%
Golf club refinement	82	148	26	69%
Client	56	83	9	77%
Product	66	92	14	64%
Invoice	125	166	33	72%

ing the development, the type checker of PVS allowed us to find some minor errors in the specification given in [2].

In [8], Büchi describes a banking application implemented in two commercial tools supporting the B-method: Atelier B from Steria and the B-Toolkit from B-Core. Bank is the largest machine of that system, and we have rewritten it in PBS. In Table 3, we compare our metrics for this example with those given by Büchi.

The difference between the size of the files is due to the fact that variables and parameters in PBS are explicitly typed, so the invariant and the pre-conditions of the operations in PBS machines become shorter. Proof obligations of both machines do not correspond one-to-one. We recall that proof obligations in PBS machines are generated by the type checker of PVS. The type checker can solve trivial type conditions, and in some cases to subsume one type condition in another one.

Table 3: Comparison Between B and PBS Machines

Machine	File length (in lines)	Proof obligations	Auto proved
Bank in PBS	232	33	66%
Bank in B	362	49	95%

The type checker of PVS was able to discharge automatically 66% of the TCCs generated by the Bank machine, while the B tool used by Büchi solved 95% of them. However, we remark that in our case, proof obligations were discharged by the built-in default tactic of PVS that is applied to TCCs. Most of the obligations that were not discharged by this tactic are easily proved by simple combinations of other standard tactics provided by the PVS theorem prover; only a couple of proof obligations required an interactively guided proof. We intend to develop a proof tactic that is customized for the kind of proof obligations generated by the PBS embedding.

PBS and some of the examples that we have developed are electronically available at: `http://www.csl.sri.com/~munoz/src/pbs.tar.gz`.

6 Related Work and Summary

The embedding of a state-oriented method such as B, VDM, or Z into a general specification language integrated with a proof checker such as HOL or PVS can be made in different ways and for different purposes.

In the context of HOL, Gordon proposes in [15] a shallow embedding of the Hoare logic in HOL, and in [7] Bowen and Gordon propose a shallow embedding of Z in HOL. The motivation is to provide mechanical support for the notation by using the HOL system. More recently, Chartier has formalized the abstract machine notation of B in Isabelle/HOL [9] 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 [18] a tool that generates a PVS representation of abstract machine proof obligations. This translation conforms to the standard semantics of B given in [2]. The goal of Pratten's work is to assist the validation process of B specifications by means of the general theorem prover of PVS. In [3], Agerholm presents a method for translating VDM-SL specifications to PVS. This shallow embedding transforms VDM-SL constructs in similar PVS constructs, and then the semantics of the VDM-SL specifications is not preserved. Maharaj and Bicarregui propose in [14] an extension of this approach to support refinements. In the case of Z, Stringer-Calvert, Stepney, and Wand present in [21] a re-reworking of a Z specification using the higher-order logic and type system of PVS. The interpretation of invariants, pre-conditions, and post-conditions that we describe here is very close to that proposed by these researchers, and in particular in Agerholm's work. However, our approach to validate refinements is different from that proposed in [14]. In that work, the proof obligations about the soundness of the refinement are not handled by the type system (as they are in our case), but they are seen as conjectures in the theory.

A shallow embedding of the abstract machine notation into a set constraint language has been studied by Tellez-Arenas in [22]. 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.

The use of type restriction to express invariants or pre- and post-conditions in PVS is not original [11,20]. However, as far as we know, our interpretation of machine refinements by using type restrictions has not been previously explored. Thus, we give a uniform interpretation in type theory of a wide subset of B constructs.

In this paper, we have proposed a shallow embedding of the abstract machine notation in PVS. Our goal is not only to assist the checking of proof obligations, but also to improve the notation with a higher-order logic. In this way, we do not translate B machines in PVS, but we provide a native notation for abstract machines in PVS. PVS abstract machines integrate the structural aspect of the B-method with the specification language of PVS. This approach provides an interesting variant on shallow embedding.

In a report of cases studies developed in B [4], the following uncomfortable aspects of the B-method are pointed out:

- Typing conditions can be handled using the set theory provided by B. However, mathematical objects as variables or functions are not explicitly typed. Although type inference is possible, in some cases this "free-typing" style obscures the specifications.

- Post-conditions are written as generalized substitutions. Although they have the same expressiveness as before-after predicates, they encourage the writing of algorithmic specifications. For some complex operations a before-after predicate could be more convenient. The same conclusion was drawn by Bicarregui and Ritchie in [5].

- Support for data types is limited. In particular, record types are absent in the B notation.

- Proof obligations usually deal with type conditions that could be easily solved by a type checker.

- B imposes a very rigid discipline. Two typical examples are that sequential substitutions are not allowed in abstract machines (but they are allowed in refinements and implementations), and parameters are restricted to be scalars or uninterpreted sets. In some of the case studies, such restrictions seem to be very strong.

Our claim is that our embedding, thanks to the higher-order logic of PVS, overcomes some of these deficiencies. We have implemented a front-end tool called PBS. The results obtained with the PBS tool are satisfactory according to our expectations: trivial type conditions are discharged automatically by the type checker of PVS and most of the proof obligations can be solved by the decision procedures provided by the theorem prover. To conclude, we summarize the main features of PBS machines:

- PBS machines are based on a rich type theory, thus all the mathematical objects are explicitly typed.
- PBS machines enforce a clear distinction between abstract sets (represented by types) and containers (represented by sets).
- The language for expressions in PBS is the PVS specification language.

Acknowledgments

The author thanks J. Rushby, B. Dutertre and M. Filali for their useful comments about this work. In [12], J. Dick describes a complete BNF for the abstract machine notation grammar. Dick's work was an excellent guide in defining the BNF for the grammar of our PBS machine notation.

References

[1] 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 *Lecture Notes in Computer Science*, pages 398–405, Berlin, Germany, October 1991. Springer.

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

[3] S. Agerholm. Translating specifications in VDM-SL to PVS. *Lecture Notes in Computer Science*, 1125:1–16, 1996.

[4] J.C. Bicarregui, D.L. Clutterbuck, G. Finnie, H. Haughton, K. Lano, H. Lesan, D.W.R.M. Marsh, B.M. Matthews, M.R. Moulding, A.R. Newton, B. Ritchie, T.G. A. Rushton, and P.N. Scharbach. Formal methods into practice: Case studies in the application of the B method. *IEE Proc. Software Engineering*, 144(2):119–133, 1997.

[5] J.C. Bicarregui and B. Ritchie. Invariants, frames and postconditions: A comparison of the VDM and B notations. *IEEE Transactions on Software Engineering*, 21(2):79–89, February 1995.

[6] 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.

[7] J.P. Bowen and M.J.C. Gordon. Z and HOL. In Jonathan P. Bowen and J. Anthony Hall, editors, *Z User Workshop, Cambridge 1994*, pages 141–167, London, June 1994. Workshops in Computing, Springer-Verlag.

[8] M. Büchi. The B bank: A complete case study. In *Proc. ICFEM98*, pages 190–199. IEEE Press, December 1998.

[9] P. Chartier. Formalisation of B in isabelle/HOL. *Lecture Notes in Computer Science*, 1393:66–??, 1998.

[10] E.M. Clarke, J.M. Wing, et al. Formal methods: state of the art and future directions. *ACM Computing Surveys*, 28(4):626–643, December 1996.

[11] Judy Crow, Sam Owre, John Rushby, Natarajan Shankar, and Mandayam Srivas. A tutorial introduction to PVS. Presented at WIFT '95: Workshop on Industrial-Strength Formal Specification Techniques, Boca Raton, Florida, April 1995. Available, with specification files, at `http://www.csl.sri.com/wift-tutorial.html`.

[12] J. Dick. Abstract machine notation - BNF definition of the ASCII syntax. Available at `http://www-scm.tees.ac.uk/bresource/welcome.html`, 1995.

[13] K. Lano and H. Haughton. *Specification in B*. Imperial College Press, 1996.

[14] S. Maharaj and J. Bicarregui. On the verification of VDM specification and refinement with PVS. In *12th IEEE International Conference on Automated Software Engineering: ASE '97*, pages 280–289, Incline Village, NV, November 1997. IEEE Computer Society.

[15] M.J.C. Gordon. Mechanizing programming logics in higher-order logic. In G.M. Birtwistle and P.A. Subrahmanyam, editors, *Current Trends in Hardware Verification and Automatic Theorem Proving, Proc. Workshop on Hardware Verification*, pages 387–439, Banff, Canada, 1988. Springer-Verlag, Berlin.

[16] University of Teesside. B-resource. Available at `http://www-scm.tees.ac.uk/bresource/welcome.html`.

[17] S. Owre, J. M. Rushby, and N. Shankar. PVS: A prototype verification system. In Deepak Kapur, editor, *11th International Conference on Automated Deduction (CADE)*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752, Saratoga, NY, June 1992. Springer-Verlag.

[18] 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.

[19] D. Rémy and J. Vouillon. Objective ML: A simple object-oriented extension of ML. In *Proc. 24th ACM Conference on Principles of Programming Languages*, pages 40–53, January 1997.

[20] J. Rushby, S. Owre, and N. Shankar. Subtypes for specifications: Predicate subtyping in PVS. *IEEE Transactions on Software Engineering*, 24(9):709–720, September 1998.

[21] D.W.J. Stringer-Calvert, S. Stepney, and I. Wand. Using PVS to prove a Z refinement: A case study. In *Formal Methods Europe FME '97*, volume 1313 of *Lecture Notes in Computer Science*, pages 573–588, Graz, Austria, September 1997. Springer-Verlag.

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