# *Integrating the B-method into PVS*

J.M. ZHOU
*Software Engineering Institute*
East China Normal University
Shanghai, China
jmchow86@hotmail.com

Jian Guo and Fu Song
*Software Engineering Institute*
East China Normal University
Shanghai, China
jguo{,fsong}@sei.ecnu.edu.cn

***Abstract*—This paper presents a method that can translate a system described by B-method into formal specification language of PVS. In this method, a machine in B-method is expressed to be a theory in PVS, while an invariant to be a type in PVS. Some properties described by B-method are transformed into conjectures of PVS language, then these conjectures can be proved effectively by PVS prover. In the end, an example, a lift system controller, is given to illustrate an application of this method.**

***Keywords-B-method; PVS; formal method; lift-system;***

## I. INTRODUCTION

With the increasing presence of information technology, the combination of hardware and software boosts the need for engineering methods and the higher quality systems. This trend is further amplified by the more complicated systems and the safety requirement. However, full or excessive testing is often not possible because of time or budget constraints of the specific product.

It has long been recognized that formal methods[1] can help engineers to improve software quality by providing ways of finding and fixing errors as well as eliminating ambiguities. In this paper, we focus our attention on the two formal description. B-method[2] and PVS system[3]. We build a PVS in which the B-method is embedded. In this way, we can not only improve the methodology for modeling, but also implement a full-typed specification language and a powerful theorem prover[4].

Section 2 discusses some basic concepts of B-method and PVS. How to integrate the B-method into PVS is given in section 3. Section 4 shows the translation through a case of lift-system controller. Section 5 covers the related works. Conclusion is given in section 6.

## II. AN OVERVIEW OF B-METHOD AND PVS

### *A. B-method*

A general development in B-method consists of an abstract specification, followed by some refinement steps. The final refinement corresponds to an implementation. A module in B-method is called an (abstract) machine. Each machine provides services allowing an external user to access or modify its current state. Syntactically, a machine consists of several clauses which determine the static and dynamic properties of states. These clauses include SETS, VARIABLES, INVARIANT, INITIALIZATION, OPERATIONS, etc. In B-method, the clause VARIABLES defines the internal change. The clause INVARIANT constraints the domains of variables. The initial state of the machine, which must satisfy the invariant, is specified as INITIALIZATION clauses. And the services provided by the machine are defined in the clause OPERATIONS. An abstract specification can be refined to an implementation. All clauses are reserved in refinement.

### *B. PVS*

The specification language of PVS is built on higher-order logic, which supports modularity by means of parameterized theories, with a rich type-system, including the notations of subtypes and dependent types. The PVS specifications are organized as a collection of theories, and each theory is composed essentially of declarations, which are used to introduce names for types, constants, variables, axioms and formulas, and importings, which allow to import the visible names of another theories.

After a module is described as a theory, type-checking in PVS must be done so that some inconsistent type can be found as TCCs (Type-Correctness conditions). The theory is considered completely after proof of the TCCs and it's conjectures are done. The PVS prover provides a variety of commands to construct proofs of the different theorems, such as (induct "n") which is to prove the formula by induction on n; and (skolem!) which is to eliminate the FORALL quantifier. It is used interactively in a sequent-style proof presentation to display the current proof goal for the proof stepbystep. The prover maintains a proof tree for the current proof goal for the theorem being proved. The user aims at constructing a complete proof tree, in the sense that all the leaves are recognized as true.

## III. INTEGRATETHE B-METHOD INTO PVS

Framework of translation from a subset ofB-method into PVS is presented in this section, in which the semantics of design description is captured.

### *A. Structure transition from B-method into PVS*

In order to prove its properties, a system represented by B-method must be integrated into PVS. The following table shows how they correspond with each other.

978-1-4244-4994-1/09/$25.00 ©2009 IEEE

Table 1

| B | PVS |
|---|---|
| MACHINE | THEORY |
| SETS | SET |
| PROPERTIES | AXIOM |
| VARIABLES | VARIABLES |
| INVARIANT | TYPE |
| INITIALIZATION | PREDICATE |
| OPERATIONS | FUNCTIONS |

In this translation, the MACHINE in B-method corresponds to the THEORY in PVS and INVARIANT to TYPE. OPERATIONS can be translated into FUNCTIONS in PVS. The syntax of B-method is expressed by specification language of PVS.

*B. Translation*

*1) SETS to SET:*

*a) B-method:*

*SETS ::= Sets; Set_Declaration | Set_Declaration*

Set_Declaration::= Identifier | Identifier = Id_List
Id_List::= Id-List | Identifier

*b) PVS:*

SetBindings ::= SetBinding[[,] SetBindings]
SetBinding ::= {IdOp[: TypeExpr]}} | Bindings

In PVS, sets of elements of a type t are represented as predicates, i.e., functions from t to boolean. A set may be given as *[t→bool]*, *pred[t]*, or *setof[t]*, which are all type equivalent.

*c) Example of usage:*

B-method: SETS BUS

PVS: bus: TYPE = SETOF[BUS]

Here we also use the set BUS in PVS.

*2) PROPERTIES to AXIOM:*

*a) B-method:*

PROPERTIES ::= Predicate
Predicate ::= Predicate ∧ Predicate | Predicate ⟹
Predicate | ¬ Predicate | ∀ Variable ● Predicate |
[Substitution]Predicate | Expression = Expression|
Expression ∈ Set

*b) PVS:*

AXIOM: PROPERTIES

Here PROPERTIES is described as a AXIOM in the PVS. AXIOM must be proved before they are used.

*c) Example of usage:*

B-method: PROPERTIES
passengers ∈ NAT ∧ driver ∈ NAT ∧ driver ≠ empty

We could assume that a bus is composed of passengers and driver. Obviously, these two parts are all natural number. Suppose there is a bus's property which the driver can not be empty.

PVS:
AXIOM: FORALL(passengers,driver:NAT):driver/=0

PROPERTIES in B-method is represented AXIOM in PVS.

*3) INVARIANT to TYPE:*

*a) B-method:*

VARIABLE ::= Identifier | Variable, Variable
INVARIANT ::= Predicates

*b) PVS:*

TypeDecl ::= Id[,Ids[Bindings]:
{TYPE | NONEMPTY_TYPE | TYPE+}
[{= | FROM} TypeExpr [CONTAINING Expr]}

TypedIds ::= IdOps [: TypeExpr] [|Expr]

TypeId ::= IdOp [: TypeExpr] [| Expr]

In B-method, the INVARIANT uses the predicates to constraint the VARIABLES. Here type declarations in PVS are used to introduce new type names to the context, and the type could be applied through a variable in the later.

*c) Example of usage:*

B-method:
VARIABLES *ticket*
INVARIANT *ticket* ∈ STOPS ⟶ PRICE

Every *ticket* has its own price, and if we know the STOPS we could get its PRICE through *ticket*(STOPS).

PVS:
TICKET: TYPE = [STOPS ⟶ PRICE]
Ticket: VAR TICKET

In PVS, we define the TICKET type first, then use it through a ticket variable.

*4) OPERATIONS to FUNCTIONS:*

*a) B-method:*

*OPERATIONS ::= Operations*

Operations ::= Operations; Operation_Declaration

In a certain sense, OPERATIONS is an abstract concept, it is defined by the user. It's quite like the function in PVS, so we decide to use the FUNCTIONS in PVS to describe the specific content in OPERATIONS.

*b) PVS:*

FUNCTIONS ::= PVS predicates

FUNCTIONSTYPES ::= $[ t_1, \cdots, t_n \rightarrow t ] |$

FUNTION$[ t_1, \cdots, t_n \rightarrow t ]$ | ARRAY$[ t_1, \cdots, t_n \rightarrow t ]$

These are three main forms of FUNCTIONS, each $t_i$ is a type expression. But sometimes we also use the predicates to define a FUNCTION.

*c) Example of usage:*

B-method:

OPERATIONS

Buy_ticket(p,t)

PRE p$\in$ passengers $\wedge$ t$\in$ tickets $\wedge$ p$\notin$ has_ticket

THEN has_ticket := has_ticket $\cup$ p $\mapsto$ price

END

This operation can be regarded as that if someone has not bought the ticket, then he must buy his ticket, and *has_ticket* which is a set from a passenger to price will record his behavior.

PVS:

Buy_ticket(p:passengers, ticket): has_ticket=

LAMBDA(p1:passengers(has_ticket))

IF p=p1

THEN has_ticket

ELSE add(p,has_ticket) AND pay(p,t)

ENDIF

In PVS, *has_ticket* is a set which record the passengers who have bought their tickets. The buy_ticket operation can judge whether this passenger buys a ticket, that illustrates the passenger has bought his ticket, otherwise he must buy his ticket first before he is added to *has_ticket*.

## IV. CASE STUDY

Here a simple lift-system is given to explain all above properties. Suppose there are n lifts in a m-floor building. Every lift has a group of buttons, each of which corresponds to one floor. The lift runs to the floors which relevant buttons are pressed and light, then the buttons' light turns off after the lift visits the relevant floors. Every floor has two buttons except the ground and the top floor. One of these two buttons is up-direction which means that someone requests the lift to the upper-floor, and the other is down-direction, expressing to the lower-floor.

### *A. Lift system in B-method*

There are two sets in this lift-system, one LIFT denotes lifts, and the other DIRECTION is the enumerated set which element reflects the way of a moving lift. Two constants: top and ground are also defined as the top and ground floors.

VARIABLES and INVARIANT:

- The variable *moving* represents the set of moving lifts;
- For each lift *l*, if *l* is not moving, The corresponding floor can be got from *floor(l)*.
- *dir(l)* is the direction of the lift *l*. When it is moving, *dir(l)* is the running direction.When it is not moving, *dir(l)* is the intended direction or keeps the final state.
- The variable *in* is a binary relation from FLOOR to DIRECTION. When (f,d) $\in$ *in,* it illustrates that some people want to travel from floor *f* in direction *d.*
- The variable *out* is a binary relation from LIFT to FLOOR. If (f, d)$\in$ *out*, then some persons want to leave this lift *l* at the floor *f.*
- The invariant *moving* $\trianglelefteq$ (out $\cap$ floor)= $\varnothing$ means *moving* doesn't appear this situation that someone requires the lift *l* to stop at the floor *f* when the current lift *l* is stopping at this floor *f.*
- In order to satisfy this condition that person in the floor *f* doesn't demand the same direction lift when a lift stops this floor, the INVARIANT in $\cap$ ran(moving $\trianglelefteq$ (floor $\otimes$ dir))= $\varnothing$ is written.

Operations:

The lift should has its own operations like: people could press buttons to request a floor inside a lift, and also could request a lift on the floor, so we define the *Request_Floor(l,f)* which means that someone wants to go to the floor *f* in the lift *l*; and we also define the *Request_Lift(f,d)* which means that someone wants to go up or down through the lift at the floor *f.*

### *B. Abstract Machine in PVS*

In our intergradation, the lift machine in B-method is translated intoa theoryin PVS. A SETS *LIFT* is changed to a SET in PVS. In order to verify some properties, here we refine this SET. The LIFT in B-method wants to describe two properties: Buttons and how to change the buttons states. So in PVS, a recordtype is written which contains two relationships: *Buttons* and *Control*. *Buttons* wants to expresses all the buttons in the lift and on the floor, and *Control* includes information of floors, lift's moving, and the direction of *lifts*.We define this recordtype as following:

Lift =[#btns: Buttons,ctr:Control#]

In B-method, the INVARIANT (ground $\mapsto$ dn) $\notin$ *in* and (top $\mapsto$ up) $\notin$ *in* constraints that the up-button couldn't press on the top and the down-button couldn't press on the ground. In PVS, two new types: *Up_floor* and *Down_floor* are described the direction buttons on the every floor explicitly. The type *Up_floor* is defined in every floor except the top floor, and *Down_floor*, in every floor excepts the ground floor. The variable moving in B-method is translated into a type Movement which enumerates all the states of a lift's change: running and halted.

The INVARIANT *out* $\in$ *LIFT* $\leftrightarrow$ *FLOOR* gives the demand of each floor in B-method. The variable [*floor* $\rightarrow$ *boolean*] in PVS describes whether the relevant floor is requested in a lift. The OPERATIONS in lift machine can be

rewritten as the functions which describe the lift's capability. *Request_floor* in B-method can be described a function in PVS. If someone press any floor-button in the lift, the floor-button should be light and the lift should reaction immediately and the service(f) be true. *Request_lift* is not explained here.

### C. *Verification of some properties*

Here we verify the proof: the lift runs to the floors which relevant buttons are pressed and light. In PVS, we should prove these two conjectures: first, if any floor-button is pressed, the button should be light; second, the lift should move to the corresponding floor.

```
response_push_floor: conjecture
    FORALL(f: Floor):
        btns(l)=push_floor(f,btns(init_lift))
        =>(floor_btns(btns(l)))(f)=light
```

The *push_floor* is an operation which the floor-button is pressed.

```
response_floor: conjecture
    FORALL(f : Floor):
        (floor_btns(btns(init_lift)))(f)=light
        => FORALL(f_1: Floor): service(f_1)=true
```

We give the prove steps of the response push floor as follows:

We give the prove steps of the response push floor as follows:

```
|----------
{1} FORALL(f : Floor):
    btns(l) = push_floor(f, btns(init_lift)) ⟹
        (floor_btns(btns(l))) = light
rule : (induct "f")
Inducting on f on formula 1
```

We use the command induct *"f"* to prove it, and it will yields 6 subgoals. The subgoals are easy to prove, we use the command *grind* or *ground* to prove them.We don't show them.

## V. RELATED WORKS

Application of the B-method are available in literatures. Chartier formalizes the Abstract Machine Notation of B embedded in Isabelle/HOL[9]. One of his works implement a tool like proof obligation generators. And C.H. Pratten gives in[10] a tool about B that generates a PVS representation of abstract machine proof obligations. Cesar Munoz proposes a embedded Abstract Machine Notation of the B-method in PVS[11], which has been implemented in the front-end called PBS. Our work integrate the B-method into PVS and prove some properties of B-method in the PVS. Through this way, the verification becomes more convenient to understand and use it.

## ACKNOWLEDGMENT

This work is supported by National High Technology Research and Development Program of China (No.2007AA010302).

## REFERENCES

[1] Gargantini, A., Riccobene, E.: ASM-based Testing: coverage criteria and automatic tests generation. In: Moreno-D'az, R., Quesada-Arencibia, A. (eds.) Formal Methods and Tools for Computer Science (Proceedings of Eurocast 2001), Canary slands, Spain, Universidad de Las Palmas de Gran Canaria, February 2001, pp. 262C265 (2001)

[2] J.-R. Abrial. The B-Book -Assigning programs to meaning. Cambridge Unviersity Press, 1996.

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

[4] Myla Archer, Constance Heitmeyer, and Steve Sims. TAME: A PVS interface to simplify proofs for automata models. In User Interfaces for Theorem Provers, Eindhoven, The Netherlands, July 1998. Informal proceedings available at http://www.win.tue.nl/cs/ipa/uitp/proceedings.

[5] Jonathan Bowen. Formal Specification and Documentation using Z: A Case Study Approach, International Thomson Publishing, 1996.

[6] S. Owre, N. Shankar, J. M. Rushby, and D. W. J. Stringer-Calvert. PVS System Guide. Computer Science Laboratory, SRI International, MenloPark, CA,September 1999.

[7] Sam Owre and Natarajan Shankar. Abstract datatypes in PVS. Technical Report SRI-CSL-93-9R,Computer Science Laboratory, SRI International, 1993.Extensively revised June 1997; Alsoavailable as NASA Contractor Report CR-97-206264.

[8] Edmund M. Clarke Jr. et al. Model Checking. The MIT Press, 1999.

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

[10] C.H. Pratten. An introduction to proving AMN specifications with PVS and the AMN-PROOF tool. In HenriHABRIAS, editor, Proc. Z Twenty Years on-What is its Future, pages 149-165. IRIN-IUT de Nantes, October 1995.

[11] C. Munoz. PBS:Supportfor the B-methodin PVS. Submitted as CSL-SRI report, 1998.