Using ABC To Prototype VDM Specifications

Aaron Kans and Clive Hayton,
Software Engineering Research Center,
Dept. of EEIE, South Bank University,
103 Borough Rd, London.

Abstract

ABC is a high-level, easy-to-use imperative language, designed originally as a replacement for BASIC. Although intended primarily as a teaching language, it has many powerful attributes that make it suitable as a language for the rapid prototyping of formal specifications. This paper illustrates how ABC was used to prototype specifications written in the formal specification language of VDM.

1 Introduction

Formal methods such as VDM [1] and Z [2] are becoming increasingly prevalent in industry as well as within graduate software engineering courses. When training staff or students to write such specifications, prototyping tools can be extremely useful [3]. The ability to rapidly produce executable programs from formal specifications allows trainees to

- become familiar with the semantics of constructs in the specification language,
- capture basic syntactic errors,
- capture semantic errors arising from imprecise specifications,
- validate formal specifications against original user requirements.

VDM is an example of a model based formal method in which the notion of *state* and *state update* play an important role. Consequently such methods are often associated with the development of software written in imperative languages such as Pascal and C. These languages make poor choices for the purpose of prototyping however. One reason for this is that they lack the expressive power associated with formal specifications. This would make the prototyping exercise far from *rapid*! ABC, on the other hand, is a simple yet very powerful interactive, imperative language (programs are typically a fifth of the size of their Pascal counterparts) that makes rapid prototyping feasible.

We outline the salient features of the language and then illustrate how it may be used to rapidly prototype VDM specifications. We concentrate first on some implementation issues and then, by way of a small example, illustrate the prototyping phase. An earlier issue of SIGPLAN NOTICES [4] presented a more detailed overview of the ABC language and, although this paper is self contained, interested readers are referred to this article. We will assume that the reader has some familiarity with VDM.

2 The ABC Language

(In the following discussion >>> is the ABC prompt) Instead of writing a long piece of code, as one would in a language like C or Pascal, ABC programmers will instead define predicates in *HOW TO REPORT* statements, functions in *HOW TO RETURN* statements and commands in *HOW TO* statements. For example:

```
>>>  HOW TO REPORT  is.even (a):  REPORT a mod 2 = 0
```

defines a predicate, *is.even*, that returns true if the (formal) parameter, *a*, is exactly divisible by 2 and false otherwise,

```
>>>  HOW TO RETURN square.of (b):  RETURN  b * b
```

defines a function, *square.of*, that returns the value obtained by squaring the parameter, *b*, and

```
>>>  HOW TO ADD.TO.TOTAL (c):
         SHARE  total
         PUT  total + c  IN  total
```

defines a command (function with side affects), *ADD.TO.TOTAL*, that adds the parameter to some running total.

Note that functions and predicates are named in lower case whereas commands are named in upper case, type declarations are not required (all type checking is done in context at run time); sequencing is indicated by a new line (as apposed to say a ;) and indentation is used to group commands (as apposed to say begins and ends). Also, in these examples, we have declared a single parameter in parentheses; this is only strictly necessary if we have a list of parameters.

In our definition of the command *ADD.TO.TOTAL*, the *SHARE* clause indicates access to a global variable, *total*, and the *PUT...IN...* clause is the ABC assignment command in which the expression, *total + c*, is evaluated and put into the variable, *total*. This assignment command is the syntactic reverse of assignments found in most languages, however we may easily define an assignment command with the more familiar syntax as follows

```
>>>  HOW TO LET var BE  exp :  PUT exp IN var
```

This single *LET...BE...* command, like any other function/ predicate/ command that we define, is immediately saved as a file by the ABC system and effectively extends the command set of the language. It is in this way that we have extended the command set of ABC to include those functions, predicates and commands that are available in the specification language of VDM but are not directly available in ABC. Users of this extended version of ABC can then begin to prototype their formal specifications. We have called this extended version of the ABC language ABC+ and we now describe the extension below.

3 ABC+: An Extension of ABC

As noted above, the *state* plays a central role in the VDM methodology. It consists of an abstract model of the system we are specifying. The model is abstract in the sense that we construct it using the *abstract* mathematical data types of sets, maps, sequences and composite objects (as opposed to the so-called *concrete* programming data types such as arrays and pointers [5]). This is in contrast to algebraic approaches to software specification, such as OBJ [6], in which there is no underlying model.

Much of the power of ABC stems from its high level data types, apart from the basic scalar types of string and number (both of unbounded length) these are

- lists (sorted collection of any one type, in which repetitions can occur)
- tables (generalised arrays with any one type of key and any one type of item)
- compounds (records without field names).

These data types are clearly very similar to those provided for in VDM. The extension of ABC to ABC+ involves defining extra functions on these types so as to completely model the data types of VDM.

3.1 Sets

Below is an example of a valid list in ABC

{ 5; 1; 5; 1; 6; 6; 8; 4; 10}

the only distinction between this list type and sets as defined in VDM is that sets can have no repetitions in them (the elements must be unique) and the elements of a set are usually separated by commas and not semicolons. We shall not be concerned about syntactic differences such as the latter. The former semantic distinction clearly does concern us. To remedy this situation we need to redefine the list constructor command of ABC, *INSERT .... IN...* (which inserts a given item into a given list), so as to only inserts those items that are not present in the original list. We do this in the obvious way as follows

```
>>>  HOW TO INCLUDE item IN set:
         IF item not.in set: INSERT item IN set
```

Note that this command includes the predefined ABC predicate, *not.in*, which tests for non-membership of an item within a list. With this constructor in place, we can now translate VDM set expressions into ABC+ expressions as follows

VDM	ABC+	DESCRIPTION
{ }	*{ }*	The empty set.
{ x1, x2,...,xn }	*{ x1; x2;...;xn }*	Set enumeration.
{ x ∈ S / P(x) }	*INITIALISE someset* *FOR x IN S:* *IF P(x):* *INCLUDE x IN someset*	Set comprehension, the set containing those elements x of the set S that satisfy the predicate P.
{ i,...j }	*{ i..j }*	Set range expression.

Note that in the above, *INITIALISE* is an ABC+ command that sets the value of the parameter to be the empty set, and the *FOR..IN..* command is the powerful ABC loop which effectively allows us to iterate over any set.

As well as representing set expressions we also need to implement the set operators as defined in VDM. The following table provides a complete list of VDM set operators and their ABC+ counterparts.

VDM	ABC+	DESCRIPTION
card S	card S	The cardinality of a set.
F S	powerset S	The set of all finite subsets of S.
∪ SS	dist.union SS	The distributed union of sets SS.
∩ SS	dist.intersect SS	The distributed intersection.
S1 ∪ S2	S1 union S2	Set union.
S1 ∩ S2	S1 intersect S2	Set intersection.
S1 — S2	S1 less S2	Set difference.
S1 ⊆ S2	S1 subset.of S2	The subset relation.
S1 ⊂ S2	S1 psubset.of S2	The proper subset relation.
e ∈ S	e in S	Set membership test.
e ∉ S	e not.in S	Set non-membership test.

It should be clear that test operators such as $\subseteq$ needed to be implemented as predicates in *HOW TO REPORT* statements, and the remaining operators as functions in *HOW TO RETURN* statements.

The last two predicates in the table above, *in* and *not.in* are predefined in ABC and the *card* function was simply a renaming of ABC's predefined cardinality operator #. The remaining operators were all easily implemented using the set constructor we defined earlier and by exploiting the power of the ABC *FOR* loop. We list the implementation of the intersection operator as an example.

```
>>>   HOW TO RETURN set1 intersect set2:
          INITIALISE  intersection
          FOR  item IN  set1:
                IF  item in set2: INCLUDE item IN intersection
          RETURN  intersection
```

This implementation follows directly from the following definition of intersection

$$S1 \cap S2 \triangleq \{ x \in S1 \quad | x \in S2 \}$$

and the translation between a set comprehension expression and an ABC+ expression.

3.2 Maps

ABC's table data type adequately models the map type of VDM. For example, we may have a mapping in VDM from a Name type to a Starsign type as follows

$$\{ Aaron \rightarrow Virgo,\ \ Clive \rightarrow Taurus,\ \ Roger \rightarrow Virgo \ \}$$

in ABC this could be represented as the following table

```
{["Aaron"] :" Virgo";   ["Clive"] : "Taurus";   ["Roger"] : "Virgo" }
```

Apart from the syntactic differences, note that types that are defined as being base types in VDM, in this example Name and Starsign, we need to implement either as numbers or as strings (the scalar types of ABC) in ABC+.

We thus have the following translation from map expressions to ABC+ expressions

VDM	ABC+	DESCRIPTION
$\{\}$	`{}`	The empty map.
$\{d1 \rightarrow r1, \ldots, dn \rightarrow rn\}$	`{[d1]:r1;...;[dn]:rn}`	Map enumeration.

$\{ d \rightarrow f(d) \mid d \in D \wedge P(d)\}$	`INITIALISE somemap`
	`FOR d IN D:`
	`IF P(d):`
	`PUT f(d) IN somemap[d]`
		Map comprehension.

While ABC table types adequately model VDM map types, their associated operators are not as rich as the map operators provided for in VDM. We therefore need to implement these map operators in ABC+. The following table provides the list of VDM map operators and their ABC+ counterparts.

VDM	ABC+	DESCRIPTION
m(d)	*m[d]*	Map application.
dom m	*dom m*	The domain of a map.
rng m	*rng m*	The range of a map.
m^{-1}	*inverse m*	Map inverse.
$s \lhd m$	*s domr m*	Domain restriction.
$s \lhd\!\!\!- m$	*s doms m*	Domain subtraction.
$m \rhd t$	*m rngr t*	Range restriction.
$m1 \dagger m2$	*m1 ovwr m2*	Map overwrite.

3.3 Sequences

Sequences are collection of (possibly repeated) items in which ordering is important. ABC does not directly provide users with a sequence type, which is often seen as basic in functional languages such as Lisp (in the functional literature sequences are referred to as lists). However, sequences themselves can be adequately modelled as a map from the natural numbers to the items of the sequence. For example, the sequence

$$[\ \ John, Rajiv, Carol, John\ \ ]$$

can be modelled as a map with domain $\{1, 2, 3, 4\}$, and range $\{John, Rajiv, Carol\}$ such that

$$\{\ 1 \rightarrow John,\ 2 \rightarrow Rajiv,\ 3 \rightarrow Carol,\ 4 \rightarrow John\ \}$$

We therefore choose to represent sequence expressions as equivalent map expressions as follows

VDM	ABC+	DESCRIPTION
[ ]	*{ }*	The empty sequence.
[e1, e2,..., en]	*{[1]: e1; [2]:e2;..;[n]:en}*	Sequence enumeration.
l (i,...j)	*l sub {i..j}*	Subsequence expression. The ith through jth element of sequence l.

Although sequences can be viewed as maps with restricted domains, the advantage in recognising sequences as a special case is that operators which are natural for sequences can be defined. We need to implement these operators in ABC+.

VDM	ABC+	DESCRIPTION
l [e]	*l [e]*	Sequence application.
hd l	*hd l*	The head of sequence l.
tl l	*tl l*	The tail of sequence l.
len l	*len l*	The length of sequence l.
l1 ins l2	*l1 ins l2*	A predicate reporting on whether or not l1 is a subsequence of l2.
inds l	*inds l*	The set of indices of a sequence.
elems l	*elems l*	The set of elements of a sequence.
l1 ^ l2	*l1 conc l2*	Sequence l1 concatenated onto sequence l2.
dconc ll	*dconc ll*	The distributed concatenation of sequences ll.

3.4 Composite Objects

The compound type of ABC closely models the composite object type of VDM. For example, one may define a composite object type, *Student*, in VDM as follows

Student :: name : Name
mark: Mark

This type has two fields, *name* and *mark*, that correspond to the name and mark of a given student. To create such an object in VDM we must use the generator function *mk*, for example

mk_Student (Susy, 78)

ABC automatically provides a generator function, so the corresponding ABC compound object would be

(" Susy", 78)

Associated with each composite object in VDM is the so-called *generic* omitted object, *nil*. Such an omitted object can not be part of the ABC+ system, it is therefore incumbent upon *users* of the system to define an omitted object for each composite object type in their state. For example, a user may define the omitted *Student* object as follows

>>> HOW TO RETURN nil.student: RETURN (" ", -99)

Where the empty object is associated with an object with (some representation of) null fields. Note that users must postfix the name of the function *nil* with the name of the given object, this is because there may be several composite object types in the state and hence several different omitted object definitions.

As we have already mentioned, ABC possesses latent types, i.e there are no type declarations. This is very useful for our purposes since this reduces the programming burden on users and hence allows for the rapid production of prototypes. To be able to effectively manipulate composite objects, however, it is necessary to declare the fields of the given objects. We have therefore provided an operation, *DEFINE*, in which users may name composite objects and list their associated field names. This command places the information provided by the users in a table (map). This table can then be accessed by the ABC+ operators *s* and *mu* which model the VDM composite object operators as follows

VDM	ABC+	DESCRIPTION
$s\_fn(o)$	*s("fn",o,id)*	The selector function returns field fn of object o, associated with identifier id.
$\mu(o, fn \rightarrow t)$	*mu(o, "fn", t, id)*	The mu function returns object o, associated with identifier id, with the field fn modified to t.

We need to pass the object id as a parameter in each case because several different objects could conceivably have the same field names.

3.5 Pretty Display Procedures

The major extension of ABC to ABC+ involves the implementation of the abstract data types of VDM. In doing so we have deliberately ignored any slight syntactic differences that arise between the ABC+ representation of these types. We have, however, provided pretty display commands, *DISPLAY SET/ DISPLAY MAP/ DISPLAY SEQUENCE*, so that users can display the data types in their VDM

syntax.

Before asking trainees to write formal specification it is often useful to ask them to evaluate simple type expressions. This allows them to become familiar with the semantics of the operators associated with each type. ABC can assist in this process since it is an interactive language and, hence, can evaluate expressions immediately. In the following examples **bold** type indicates system output.

```
>>>    DISPLAY SET ( { } union {5..10} )
       { 5, 6, 7, 8, 9, 10 }

>>>    LET grademap BE {["Mark"]: 55; ["Liz"]:88; ["Tony"]:55}
>>>    DISPLAY MAP (grademap ovwr {["Mark"]: 95}
       { Liz → 88, Mark → 95, Tony → 55 }
```

and so on. Note that ABC keeps, and therefore displays, all data types as sorted. Of course the main purpose of our extending ABC to ABC+ is so that users may be able to effectively prototype their VDM specifications. It is this process we now look at.

4 From VDM to ABC+

We will illustrate the actual prototyping phase by way of an example. While fairly small, this example illustrates the important steps that users must go through in order to prototype their VDM specifications. The requirements of the specification are as follows

"A residents car park is to be specified in which permit holders only may park. Traffic needs to be recorded and permits must be allowed to be allocated and cancelled. Initially the car park is to be empty and no permits are to have been issued."

From these requirements the following state definition is drawn up which involves modelling both the group of cars that have permits and the group of cars parked in the car park as sets (we stick to the notation of [1]).

```
CarparkSys ::    parked:  Car_set
                 permits: Car_set
```

The fact that only permit holders may park in the car park is a *real world constraint* and as such needs to be recorded in the *state invariant*

$$inv\ (mk\_Carparksys(parked,permits)\quad \triangleq parked \subseteq permits$$

4.1 Entering a VDM State Definition Into ABC+

A VDM state definition consists of a type declaration of variables in the state, a possible state invariant and possible global functions. Since ABC has latent types we enter a VDM state definition into ABC+ by

- defining any composite objects we may have in our state through the *DEFINE* operation,
- defining the state invariant (which is simply a global predicate) in a *HOW TO REPORT* statement , where the relevant state variables are listed in a *SHARE* clause,
- defining global functions in *HOW TO REPORT* statements.

In our example we have only a state invariant to consider, which we then implement as follows

```
>>>  HOW TO REPORT inv:
         SHARE parked, permits
         REPORT parked subset.of permits
```

(Of course, tests in VDM may include the logical connectives as well as atomic assertions. ABC not only supports the standard connectives *AND, OR* and *NOT* but also the existential quantifier, *SOME x IN set HAS predicate(x)* , and the universal quantifier, *EACH x IN set HAS predicate(x).*)

Having declared the invariant the user is then ready to enter her operation specifications.

4.3 Entering VDM Operation Specifications Into ABC+

We look at in detail the translation of one particular operation, *NEWPERMIT* (that allocates a permit to a given car) from this example a general method of translation should be clear. Its specification is given as

```
NEWPERMIT (c : Car)
ext  wr  permits: Car-set
pre  c ∉ permits
post permits = permits' ∪ {c}
errs "CAR HAS PERMIT" : c ∈ permits
```

This says that the operation, *NEWPERMIT*, has one input parameter, *c*, and has write access to the state variable *permits*. ***Pre*** specifies the pre condition of the operation, that the car should not already have a permit, and ***post*** specifies the post condition of the operation, that the new value of permits should be equal to the old value of permits (here represented as *permits'*) union with the singleton set *{c}*. ***Errs*** specifies any error messages we may wish to output and the conditions under which we output them. Translating this specification into ABC+ gives

```
>>>  HOW TO NEWPERMIT (c):
         SHARE permits
         WR permits USING old.permits
         SELECT
              pre: POST
              ELSE: WRITE"CAR HAS PERMIT"
         END
     pre: REPORT c not.in permits
     POST: LET permits BE old.permits union {c}
```

This translation process consists of the following steps.

We translate *NEWPERMIT*'s header into ABC+ by

- preceding the operation name with the words *HOW TO* (since we implement operations as ABC commands)
- listing parameters without their type information.

Ext indicates which state variables the operation needs access to, the types of these variables, and the type of access to these variables (either ***rd*** for read_access or ***wr*** for write_access). In ABC+ we

- list which variables we need access to in a *SHARE* clause,
- indicate the type of access in either a *RD* or a *WR..USING..* clause.

RD and *WR..USING..* are two more ABC+ commands (the latter defines a hooked variable i.e the value of a variable before execution of the operation, where as no such variable is defined in the former).

SELECT is the ABC n-branched conditional command thus

```
SELECT:
        pre: POST
        ELSE: WRITE"Error message"
```

can be read as "*if the test pre is true then execute the command POST, otherwise output an error message*". The test *pre* and the command *POST* are still to be refined and we do so at the end of the operation, these refinements are seen as local to the command *NEWPERMIT* and so we may use the same names *pre* and *POST* in other operations without causing confusion (note that local refinements are not preceded with the words *HOW TO*). Of course VDM post conditions are tests which tell us what must be true after execution of the operation where as *POST* is a command which needs to satisfy a post condition (we return to this point later). Instead of refining *pre* and *POST* at the end of the operation it is of course possible to write their definitions directly into the *SELECT* clause.

Finally the *END* clause is an ABC+ command that instructs the system to check that the state invariant has not been violated upon execution of this operation.

Before we can begin execution of our operations we must initialise our state, our requirements tell us that both there are to be no cars in the car park, and no permits issued thus we have

```
>>>  LET parked, permits BE { }, { }
```

Note that ABC supports the concept of multiple assignments. We can now attempt to issue a permit to a car. Since Car is a base type in our original spec we must represent this either as a string or a number, we choose a number.

```
>>>  NEWPERMIT(1)
```

We have received no type-error messages as a result of executing this operation, so we can assume our specification is *syntactically* correct. We can now check our state variables to see whether or not the intended *semantics* of the operation are respected.

```
>>>  DISPLAY SET permits
     {1}
>>>  DISPLAY SET  parked
     { }
```

This is as expected. We can now test to see what happens if we try to issue another permit to the same car.

```
>>>  NEWPERMIT(1)
     CAR HAS PERMIT
```

The error message we expected has been raised. Furthermore, since the system automatically reports on any violation of the state invariant, we know that execution of this operation has respected the state invariant. We can then implement and execute the remaining operations and test properties such as

- *"If I give car 1 a permit, then I enter car 1 into the car park, a list of cars in the car park should include car 1"*,
- *"If I cancel car 1's permit, then attempt to enter car 1 into the carp park, an error should be raised"*,

and so on, to examine whether or not the results obtained are consistent with original user requirements.

5 Final Remarks

We have shown how by extending ABC to ABC+ we arrive at a useful tool for the rapid prototyping of formal specifications. We have concentrated on VDM specifications but users of other model based formal methods such as Z could also benefit from exploiting the power of ABC to prototype their specifications. We have used the tool in our postgraduate formal methods course and have had a positive response from our students.

For the prototyping exercise to be plausible, of course, we require operation post conditions to be explicit enough to reveal their computational content, i.e it should be possible to rapidly arrive at a set of commands that satisfy these post conditions. This is true not only of any imperative prototyping tool, but also those tools built around functional languages [7]. Since we intend this tool to be used primarily in the educational stages of formal methods training (where specifications are at the level of complexity contained in most text books), we have found that by and large post conditions are explicit enough for our purposes.

We have described the ABC language but we have also indicated that ABC is a complete system. For instance, all file handling is carried out within the system so there is no need to teach users separate file handling commands. Thus, for example, at the end of a session the system automatically saves all variables and all code as defined by the user. Also, users of the system are always working in a syntax directed editor. Consequently they will find that they have to do very little typing; the system provides command templates, free matching closed brackets and speech marks, free indentation and so on. This makes the prototyping stage even more rapid. For a definitive discussion of the system and the language readers are referred to [8].

There exist ABC implementations for the IBM PC or compatibles, the Apple Macintosh and for Unix machines, and these implementations are freely available from many bulletin boards and servers. We hope to have included enough information in this paper to indicate how users can extend ABC to suit their particular needs, but for those readers who wish to obtain a copy of our ABC+ tool (which runs on a PC), send a disk to the first author at the South Bank University address given on the title page.

ACKNOWLEDGEMENTS

The authors would like to thank David Mole for his advice throughout the writing of this paper. The first author is supported by a grant from the Science and Engineering Research Council.

REFERENCES

1 **Jones C B,** Systematic Software Development Using VDM (2nd edition), Prentice-Hall (1990)

2 **Spivey J M,** The Z Notation, Prentice-Hall (1990)

3 **Hekmatpour S, Ince D,** Software Prototyping, Formal Methods and VDM, Addison-Wesley (1988)

4 **Pemberton S** 'A short introduction to the ABC language' SIGPLAN NOTICES Vol 26 No 2 (February 1991) pp 11-16

5 **Ince D** ' Arrays and pointers considered harmful' SIGPLAN NOTICES Vol 27 No 1 (January 1992) pp 99-104

6 **Gougen J, Meseguer J** 'Rapid prototyping in the OBJ executable specification language' Squires (1982) pp 75-84

7 **Alexandra H, Jones V,** Software Design and Prototyping Using Me Too, Prentice-Hall (1990)

8 **Geurts L, Meertens L, Pemberton S,** The ABC Programmer's Handbook, Prentice-Hall (1989)