Supporting Co-Use of VDM and B by Translation

Juan Bicarregui[1], Matthew Bishop[2], Theodosis Dimitrakos[1], Kevin Lano[2], Tom Maibaum[2], Brian Matthews[1], and Brian Ritchie[1]

[1] Rutherford Appleton Laboratory, Chilton, Didcot, Oxon OX11 0QX
{J.C.Bicarregui, T.Dimitrakos, B.M.Matthews, B.Ritchie}@rl.ac.uk
[2] Computer Science Dept., King's College London, Strand, London WC2R 2LS
{bishopm, kcl}@dcs.kcl.ac.uk, tom@maibaum.org

Abstract. VDM and B are two mature formal methods currently in use by industry and supported by commercial tools. Though the methods are foundationally similar, the coverage of their supporting tools differs significantly. The integration and co-use of the two methods has been considered in a number of previous papers, and it has been demonstrated that both methods can be profitably applied at different points in the development life-cycle, with initial abstract specifications in VDM being translated into design specifications in B prior to refinement into code. In this paper we describe a partial translation from VDM to B, which may allow automated support for this step in the process. We also describe possible future extensions to the translation.

1 Introduction

VDM [Jon90] and B [Abr96] are model-oriented formal methods for the development of sequential systems; both are currently supported by mature commercial toolkits. Both are based on first-order predicate calculus and set theory, although their underlying logics are different, and both have proof rules for formal verification and validation. Both are equipped with a formal semantics.

Despite these similarities, the supporting tools for VDM [Gro99b] and B [B-C00] differ considerably in their capabilities for analysis, proof, animation and code generation. Because users from both communities could benefit if these complementary functionalities could be made to cooperate, there has already been considerable research on the feasibility of integrating the two methodologies. Their notations were compared in the B User Trials [BR93,BM95]; integration of the two was investigated in the MaFMeth project [BDW95,BDW96,BDMW97], in which a manual translation from VDM to B was used in order to apply formal methods to the whole life-cycle of a commercial product. It was discovered that a large number of errors were introduced by the informal manual translation between the two languages, and so in the Spectrum project [BMRA98,MRB98] formal (partial) translations between VDM and B were introduced, although these were still carried out by hand. Work on integrating VDM and B is currently continuing [BDL+99] in the VDM+B project[1].

We now briefly describe the similarities and differences between VDM and B, before outlining a partial translation and the issues which remain to be considered in supporting it.

1.1 VDM

VDM has a rich set of data type constructors, augmented with invariant predicates, and a large expression and statement language. Functions and state-transforming operations can be defined

[1] http://www.itd.clrc.ac.uk/Activity/VDMB

either implicitly, using pre- and post-conditions, or explicitly. The strong type system supports static detection of many well-formedness errors.

There is an ISO standard [LHB+96] for VDM-SL which includes a denotational semantics [LP95] based on the logic of partial functions [JM94]. There is also a proof theory [Jon90,BFL+94], which supports the validation of specifications through the discharge of proof obligations. The proof theory has not been validated with respect to the semantics. Ongoing work on supporting proof in VDM via the PROSPER toolkit [DCN+00] will provide another axiomatic semantics which is known to differ from the denotational semantics.

The standard for VDM-SL does not include any support for structuring, although several alternative approaches are outlined in an "informative annex" to the standard, and one of these alternatives is implemented by the IFAD VDM-SL Toolbox [Gro99a,Gro99b]. Structuring in VDM is discussed in another paper to be presented at this workshop.

1.2 B

The B Method is based on *Abstract Machine Notation* (AMN), which uses generalized substitutions to represent state transformations. The language contains a number of powerful structuring mechanisms based on abstract machines, allowing for modularisation of code by means of data encapsulation. However, the expression language of B is much less rich than that of VDM.

The underlying semantics of B is grounded in weakest preconditions over untyped set theory and classical logic. There is a proof theory [Abr96,Lan96], and the available tools allow for proof of refinement from abstract specification through to imperative code. Again, the proof theory has not been completely formally verified with respect to the semantics.

1.3 Objectives of the Translation

There are four principal issues to be considered in translating VDM to B:

Automatability The cost of moving from VDM to B should be minimal, and hence the translation should be as automatic as possible. We do not propose that the translation should be fully automatic, since experiments have shown that user intervention is necessary in order to generate a good specification. Nevertheless, it is envisaged that the process be guided by a machine which could also automatically translate parts of the VDM specification.

Correctness The semantics of the source specification should be preserved in the target specification. Of course, the semantics and underlying logics of the two languages are different, although we believe that the differences are sufficiently small to be ignored in the first attempt at a translation. This will naturally affect the formal correctness of our translation.

Conservativeness This is similar to the previous point: the translation should keep the same model in the source and target languages. That is, the data models should be similar, and the operations at similar levels of granularity, in both specifications. The different structuring conventions of the two languages, and the comparative richness of the expression language of VDM as compared to B, will both have effects on the conservativeness of our translation.

Utility The results of the translation should be useful! In particular, the target specification should be comprehensible to a human user, and amenable to manipulation using the standard tools for B.

1.4 Approach to the Translation

A partial translation from VDM to B was outlined in [BMRA98,MRB98]. The translation which we present below is still partial, and omits many details. We begin by setting out some overall guidelines which will motivate the rules of the translation:

- Each record type which is used as a type, or part of a type, of a state component variable in VDM is represented by an object manager machine.
- Other types, such as those used only in inputs and outputs of operations, are translated by a property-oriented (i.e. algebraic) specification.
- Constants and function definitions may be translated as separate stateless machines. However, function definitions may have to be translated into (possibly stateless) operations if they manipulate types which have been translated into object manager machines. In general, we translate VDM functions into stateless operations wherever possible.
- Machines may need to be combined to assert invariants involving several objects.
- The state model and operations in VDM are translated into a top-level state machine in B.

As in [MRB98], we introduce three possible translations for record types: a property-oriented approach, a simple object machine and an object manager. The property-oriented approach, which amounts to an algebraic specification of the type, is used by all record types which are not part of the state model. It has the disadvantage of producing a machine with a very large and unwieldy properties clause containing complex expressions in set theory and first-order logic. The support tools for B find such expressions hard to deal with, and the resulting specification is rather unnatural for B.

The object-machine and object-manager approaches, which apply only to record types (and aggregates of record types) which are part of the state model, allow a more natural structured specification to be developed. However, types defined in this way cannot be used in (for example) arguments to functions. Choosing the appropriate translation for a record type which occurs in the state model will require a preprocessing step, which we discuss later.

2 Definition of the Translation

The translation functions are written in the form

TranslationFunction[VDM_Term]$\rightarrow AMN\_Term$

This is not sufficiently general to capture all of the process, since certain rules have non-local actions, which are currently given informally. A more general formulation would have the form

TranslationFunction[VDM_Term]$AMN\_Env \rightarrow AMN\_Term \times AMN\_Env$

where AMN_Env is the environment — the current collection of AMN machines, including refinements and implementations. This more general formulation has not yet been fully developed.

There are a number of these basic translation functions, which translate types, type definitions, state definitions, expressions, functions, operations, etc. Not all of these functions have yet been formally defined. In particular, the Expression function (which translates VDM expressions into AMN) is not defined in this paper, although it is invoked by both TypeDef and StateDef.

For the sake of clarity, we have allowed the use of subscripts and single-letter identifiers in the AMN translations, although these are not strictly permissible in B.

2.1 Type Definitions

Basic Types

The VDM basic types are

```
basic type = 'nat' | 'nat1' | 'int' | 'rat' | 'real' | 'char' | 'bool' | 'token'
```

These are translated to the nearest available B types where possible. The function Type defines the translation:

$\mathsf{Type}[\ nat\ ] \rightarrow \mathbb{N}$
$\mathsf{Type}[\ nat1\ ] \rightarrow \mathbb{N}_1$
$\mathsf{Type}[\ char\ ] \rightarrow STRING$

There are of course semantic differences between these types in VDM and B — in particular, sets take the role of types in B, and all sets in B are finite. These differences are not yet accounted for in the translation.

The VDM types `bool` and `int` require the declaration of library machines which must be included into the specification, and which cannot be captured in the notation at present:

$\mathsf{Type}[\ bool\ ] \rightarrow$ **INCLUDE Bool_TYPE**
$\mathsf{Type}[\ int\ ] \rightarrow$ **INCLUDE Int_TYPE**

The types `rat` and `real` should be handled similarly, once appropriate library machines are defined.

Type Definitions: Synonyms

Synonyms such as $Time = \mathbb{N}$ are translated as constants, using an auxiliary function *typeMachinesFromType(typ)*, which finds other defined types contained in *typ* and returns the names of the machines they are named in.

$\mathsf{TypeDef}[\ name = typ$
$inv\ p == e$
$] \rightarrow$
MACHINE *name*‘**_type**’
SEES *typeMachinesFromType(typ)*
CONSTANTS
name
PROPERTIES
name = $\mathsf{Type}[\ typ\ ]$
$\wedge$
$\mathsf{Invariant}[\ p == e\ ]$
END

For example, $Time = \mathbb{N}$ would be translated into:

MACHINE Time_type
CONSTANTS *Time*
PROPERTIES $Time = \mathbb{N}$
END

Type Definitions: Enumerations

Enumerated types in VDM are of the following form:

```
type definition = identifier , '=' , name , { | name }
```

where each `name` is a new token. They are translated into set constants, with the tokens as elements of the set. In VDM, enumerated constants are also types, and so are also translated as sets. If

the VDM specification does not make use of the enumerated constants as types, then the above translation is sufficient; otherwise, the enumerated type is translated into B is as follows:

TypeDef[$name = tok_1 \mid \ldots \mid tok_n$
]$\rightarrow$
MACHINE $name$**'_type'**
SETS
tok_1*'_set'* = { tok_1 },
...
tok_n*'_set'* = { tok_n },
$name$*'_set'*
PROPERTIES
$name$'_set' = tok_1*'_set'* $\cup \ldots \cup$ tok_n*'_set'* $\wedge$
$tok_1 \neq tok_2 \wedge$
...
$tok_{n-1} \neq tok_n \wedge$
END

For example, the enumerated type *Switch* = On | Off is translated to:

MACHINE Switch_type
SETS
On_set = { *On* }, *Off_set* = { *Off* },
Switch_set
PROPERTIES
Switch_set = *On_set* $\cup$ *Off_set* $\wedge$
On $\neq$ *Off*
END

Type Definitions: Record Types

The VDM concrete syntax for record types is as follows:

```
type definition = identifier , '::' , field list , [ invariant ]
field list = { field }
field = [ identifier , ':'] , type
```

This may be translated into a property-oriented (algebraic) specification by defining both the type identifier *name* and the fields $l_1, \ldots, l_n$ as constants in B. For each field l_i we add a projection function from the base type; this simulates the field selection feature of VDM in B. The properties clauses describe the algebraic theory of these projections and their interaction with the *'mk_'name* constant. We introduce a new set *name'0'*, and use the constant *'inv_'name* is to define *name* as the subset of *name'0'* over which the invariant holds. It would be more natural to define both *name'0'* and *name* as sets, but one set cannot be defined in terms of another in B.

The translation function is as follows:

TypeDef[$name :: l_1 : t_1, \ldots, l_n : t_n$
$inv\ mk\_name(x_1, \ldots, x_n) == e$
$] \rightarrow$
MACHINE $name$**'_type'**
SEES
t_1'_type', ..., t_n'_type', $BOOL\_Type$
SETS
$name$'0'
CONSTANTS
$name$,
'mk_'$name$,
'inv_'$name$,
$l_1, \ldots, l_n$
PROPERTIES
$name \subseteq name$'0' $\wedge$
'mk_'$name$: $t_1, \ldots, t_n \rightarrow name$'0' $\wedge$
$l_1 : name$'0' $\rightarrow t_1 \wedge$
...
$l_n : name$'0' $\rightarrow t_n \wedge$
$(\forall(vt_1, \ldots, vt_n).(vt_1 : t_1, \ldots, vt_n : t_n \Rightarrow$
$(l_1(\text{'mk\_'}name(vt_1, \ldots, vt_n)) = vt_1))) \wedge$
...
$(\forall(vt_1, \ldots, vt_n).(vt_1 : t_1, \ldots, vt_n : t_n \Rightarrow$
$(l_n(\text{'mk\_'}name(vt_1, \ldots, vt_n)) = vt_n))) \wedge$
$(\forall(nn).(nn : name \Rightarrow (\text{'mk\_'}name(l_1(nn), \ldots, l_n(nn)) = nn))) \wedge$
'inv_'$name$: $name$'0' $\rightarrow$ BOOL $\wedge$
$(\forall(x_1, \ldots, x_n).(x_1 : t_1, \ldots, x_n : t_n \Rightarrow$
$((\text{'mk\_'}name(x_1, \ldots, x_n) = \texttt{TRUE}) \Leftrightarrow (\mathsf{Expression}[e])))) \wedge$
$(\forall(nn).(nn : name$'0' $\Rightarrow ((\text{'inv\_'}name(nn) = \texttt{TRUE}) \Leftrightarrow (nn : name))))$
END

Note that the invariant returns an object-level boolean value, and so expressions containing the invariant must equate it with TRUE in order to raise it to the logical level. An alternative approach would be to make the invariant a B definition (i.e. a syntax-level substitution), but such definitions are not exported with a machine and must be repeated in every machine in which they are used. However, this would not be an obstacle to mechanical translation, and might result in fewer or more tractable proof obligations.

Other type definitions, such as recursive types, are not yet handled by the translation.

2.2 The State Model

A VDM state has the concrete syntax:

```
state definition = 'state' , identifier , 'of' ,
                             field list ,
                   [ invariant ] ,
                   [ initialisation ] ,
                   'end'

invariant = 'inv' , invariant initial function

initialisation = 'init' , invariant initial function
```

In the translation to B, we translate this into two machines, one of which uses the TypeDef function defined earlier to generate the type definitions. In the other machine, variables are introduced for each state component, and the invariant is translated directly from the state invariant, with the addition of the substitution from formal variables to state variables. The initialisation is given by assigning the state variables to any value of the state type which satisfies the translation of the invariant. The operations section is left blank at this stage; operations are translated separately.

The first machine, handling the type definitions, is produced by applying the TypeDef translation function described above:

TypeDef[$name :: l_1 : t_1, \ldots, l_n : t_n$
$inv\ mk\_name(x_1, \ldots, x_n) == p$]

The second machine is given by the new StateDef translation function:

StateDef[state $name$ of
inv $mk\_name(x_1, \ldots, x_n) == p$
init $n == q$
end
] $\rightarrow$
MACHINE $name$
SEES
$name$'_type', t_1'_type', ..., t_n'_type'
VARIABLES
$vl_1, \ldots, vl_n$
INVARIANT
$vl_1 : t_1 \wedge \ldots vl_n : t_n \wedge$
Expression$[p][x_1 := vl_1, \ldots, x_n := vl_n]$
INITIALISATION
ANY n
WHERE
$n : name \wedge$ Expression$[q]$
THEN
$vl_1 := l_1(n)$ || /*" function l_1 is from machine t_1'_type' "*/
...
$vl_n := l_n(n)$
END
OPERATIONS
...
END

Record Types in the State Model

For record types which are referred to by the state, we may in some cases avoid the use of the property-oriented translation which was given above.

We shall describe the alternative translations in these cases by means of examples, rather than presenting them in full generality. Suppose first that R is a record type of the following form:

$R :: r_1 : R_1,$
$\quad r_2 : R_2$

We may translate this into the following simple state machine:

MACHINE R_obj
VARIABLES r_1, r_2
INVARIANT $r_1 : R_1 \wedge r_2 : R_2 \ \ldots$
END

This may be included into a top-level state machine, using a renaming — for example, if the state contains $r : R$, the translated machine should contain **INCLUDES r.R_obj**. If R_1 or R_2 are themselves record types, they can be defined in terms of machines which are then included by *R_obj*.

If the state contains an aggregate type (set, map, sequence, etc.) of records, then it can be handled using an object manager. For example, the type *R-set* may be translated into the following manager machine:

MACHINE R_mgr
SETS *R_Ids*
VARIABLES $rids, r_1, r_2$
INVARIANT $rids \subseteq R\_Ids \wedge r_1 : rids \rightarrow R_1 \ \wedge \ r_2 : rids \rightarrow R_2 \ \wedge \ \ldots$
END

This machine is also included into the top-level state machine, but without renaming — we have only one *R*-manager in the system. If the VDM state contains a variable *r:R-set*, then the top-level B state machine should be as follows:

MACHINE S
INCLUDES R_mgr
VARIABLES $r, \ldots$
INVARIANT $r \subseteq rids \wedge \ldots$
$\ldots$
END

Again, if R_1 or R_2 are themselves record types, they may be broken down into smaller machines. There may be a problem if a manager machine such as *R_mgr* is referred to by more than one record type, since the rules of composition in B do not allow a machine to be included in more than one other machine. This problem can be resolved by splitting *R_mgr* into two machines: a simple machine declaring the abstract set *R_Ids*, and a manager machine. The former is then accessed using the SEES construct, and the latter is included with renaming as needed. This allows the same set of object identifiers to be used by different object managers.

Keeping the above machines in mind, we now return to considering the state model. We begin by preprocessing the record types which occur in the state model, by dividing them into two disjoint sets *Simple* and *Manager*. Record types which are referenced directly are added to *Simple*, while those which are part of aggregates (sets, maps, sequences) are added to *Manager*. We then impose the conditions that any type in *Manager* must be removed from *Simple*, and any record types which occur in a field of a type in *Manager* must themselves be added to *Manager*.

There are then three cases to consider: a *Simple* record which contains a reference to another *Simple* record. a *Simple* record which refers to a *Manager* record, and a *Manager* record which refers to a *Manager* record. (The preprocessing rules out the other case, of a *Manager* record referring to a *Simple* record.)

In the first case, suppose *name*$\in$*Simple* where *name*:: $l_1 : T, \ l_2 : R$, and suppose further that T is not a record type, and R is a record type in *Simple*.

TypeDef[*name*:: $l_1 : T,\ l_2 : R$
]$\rightarrow$
MACHINE *name*‘**_obj**’
INCLUDES l_2.*R‘_obj’*
VARIABLES l_1
INVARIANT $l_1 \in T$
...
END

where *R_obj* is a simple state machine as above.

In the second case, suppose *name*$\in$*Simple* where *name*:: $l_1 : T,\ l_2 : R$, and further that T is not a record type, and R is a record type in *Manager*.

TypeDef[*name*:: $l_1 : T,\ l_2 : R$
]$\rightarrow$
MACHINE *name*‘**_obj**’
INCLUDES *R‘_mgr’*
VARIABLES l_1, l_2
INVARIANT $l_1 \in T \wedge l_2 \in rids$
...
END

where *R_mgr* is a manager machine as above.

In the last case, suppose *name*$\in$*Manager* where *name*:: $l_1 : T,\ l_2 : R$, and further that T is not a record type, and R is a record type in *Manager*.

TypeDef[*name*:: $l_1 : T,\ l_2 : R$
]$\rightarrow$
MACHINE *name*‘**_mgr**’
INCLUDES *R‘_mgr’*
SETS *name‘_Ids’*
VARIABLES *name‘ids’*, l_1, l_2
INVARIANT
name‘ids’ $\subseteq$ *name‘_Ids’* $\wedge$
$l_1 \in$ *name‘ids’* $\rightarrow T \wedge$
$l_2 \in$ *name‘ids’* $\rightarrow$ *rids* $\wedge$
...
END

where *R_mgr* is a manager machine as above. It may also be necessary to give property-oriented translations as well as state-based translations for some record types — in particular, types which are used in the input or output of functions.

2.3 Operations

Here we consider explicit operations, which are defined in VDM as follows:

```
explicit operation definition =
         identifier , ‘:’ , operation type ,
         identifier , parameters , ‘==’ , statement ,
                 [ ‘pre’ , expression ] ,
                 [ ‘post’ , expression ]
```

The formal translation rules for operations have not yet been completed, and so instead we provide an example of a translation and comments on the issues involved. The following operation was taken from a case study:

ControlCycle: Buttons × *TrainVelocity* ⇒ *Action*
ControlCycle(b,sp) ≙
  let *mk_(tt1, tt2)* = *TrainTransition(train,doors.ds,b)*,
    mk_(dt1, dt2) = *DoorTransition(doors,train.ts,b,clock)*
  in (*clock* := *clock* + 1;
    train := *mk_Train(tt1, sp)*;
    doors := *dt1*;
    return *mk_Action(tt2,dt2)*);

There are a number of issues to consider in translating this to AMN. As this operation returns a value, the B version requires a name, which must be provided by the translator. The B version requires a precondition even if the optional precondition of the VDM operation is absent; this is used to give the types of the arguments, which are implicit in VDM but must be made explicit in B.

The VDM `let` construct is translated into an ANY substitution in B, and typing information which was implicit in the VDM specification must be made explicit in the B specification. The pair patterns in the `let` clause have been translated to explicit pairs (maplets).

The VDM operation uses sequencing, the closest counterpart of which in B is also sequencing (of generalised substitutions); however, sequencing is not available in B machines, but only in implementations. In this case, parallel substitution suffices, but only because there is no potential inference between the substitutions.

The translated operation is as follows:

action ⟵ *ControlCycle(bb, sp)* ≙
  PRE
    bb : Buttons ∧ *sp : TrainVelocity*
  THEN
    ANY *tsr, tar, dr, dar*
    WHERE
      tsr : TrainState ∧ *tar : TrainAction* ∧
      dr : Door ∧ *dar : DoorAction* ∧
      (tsr ↦ *tar) = trainTransition(train, ds(doors), bb)* ∧
      (dr ↦ *dar) = doorTransition(doors, ts(train), bb, clock)*
    THEN
      clock := *clock* + *1* ||
      train := *mkTrain(tsr, sp)* ||
      doors := *dr* ||
      action := *mkAction(tar, dar)*
    END
  END ;

Where parallel substitution is not acceptable because there is potential interference between the substitutions, it may be possible to translate using another ANY substitution to introduce temporary variables. For example, the following fragment of a VDM specification:

swp()
ext wr $xx : \mathbb{N}$
wr $yy : \mathbb{N}$
pre true
post $xx = \overleftarrow{yy} \ \wedge \ yy = \overleftarrow{xx}$

could be implemented as a fragment of a B machine:

OPERATIONS
swp =
ANY *aa,bb*
WHERE
$aa : \mathbb{N} \wedge$
$bb : \mathbb{N} \wedge$
$aa = yy \wedge$
$bb = xx$
THEN
$xx := aa \ \|$
$yy := bb$
END ;

2.4 Functions

In general, it is the function definitions in VDM which present the greatest difficulties in translation, since many VDM-SL expression constructs have no direct equivalents in the AMN language[2].

Our approach is to translate VDM functions into B operations as much as is practically possible. This is particularly necessary in the case where a function manipulates a type which has been translated into an object manager. Some analysis of functions is thus required in order to determine which part of the state they are applied to, and which machine to enter them into. Function definitions (and constants) may also be translated into separate stateless machines in some cases.

Certain functions are reasonably straightforward to translate. For example, if a function has the following signature:

record_fun1 (rin : Record , $a_1 : t_1, \ldots, a_n : t_n$) *rout : Record*
pre ...
post *rout* = *mk_Record*($P_1(rin, a_1, \ldots, a_n), \ldots, P_m(rin, a_1, \ldots, a_n)$);

where *Record* is a record type for which an object manager machine *Record_obj* has been defined, then we translate this function as an operation which is added to the *Record_obj* machine, as follows:

[2] For example, let us briefly consider the VDM-SL `if-then-else` and `cases` expressions. B has no expression-level `if-then-else` construct, although it does have one in the generalised substitutions syntax. We could translate `if A then B else C`, assuming that `B` and `C` are of boolean type, as `(A=>B) & ((not A) => C)`. If `B` and `C` are not boolean, then we pull the conditional further out before translating, so that `f(x) = if A then 3 else 7` becomes `if A then (f(x)=3) else (f(x)=7)`. The `cases` expression could then be translated similarly, although the difference in semantics between VDM and B means that we must ensure that the cases are disjoint in B, adding yet another complication to the translation.

$record\_fun1(a_1, \ldots, a_n) \mathrel{\widehat{=}}$

PRE $a_1 \in t_1 \wedge \ldots \wedge a_n \in t_n$

$\ldots$

THEN $r_1 := P_1(a_1, \ldots, a_n);$

$\ldots$

$r_m := P_m(a_1, \ldots, a_n);$

END ;

In this operation, $r_1, \ldots, r_m$ are the variables of the record machine. The expressions $P_1, \ldots, P_m$ may involve the variables of the machine, and may also themselves be operations (with consequent changes in syntax), expecially if the variables are themselves record types and thus provided by an included machine.

3 Extending the Translation

It will be clear from the above that the translation is defined only for a subset of the VDM-SL language, and that this subset does not contain some of the essential features of the language. Accordingly, a major priority for further work on this project is to extend the translation to cover more of VDM-SL. Since manual translation has been shown to be a major source of errors, we also intend to produce a tool for automatically (with guidance from the user) translating VDM specifications into B.

VDM and B have the same expressive power in theory, and so any VDM specification should in theory have an equivalent B specification. However, we do not anticipate that our translation will ever encompass the whole VDM-SL language. Some VDM-SL constructs are sufficiently difficult to translate into B that it is doubtful whether a machine translator could produce a useful equivalent specification, particularly given our requirement that the translated specfication be usable both for the toolkit and the user. In this situation, we advocate either refinement in VDM (with the B style in mind) prior to translation, or else manual translation.

We intend to implement the translation tool in Standard ML of New Jersey[3] For VDM files in the standard VDM-SL and IFAD ASCII syntax, we have already developed a parser using the ML-Lex (lexical analyser generator) and ML-Yacc (parser generator) programs which are distributed as part of the SML/NJ package.

4 Proof Support

The correctness of the translation remains an issue; ideally, this would require verifying the translation rules in some common model. Given the different semantics and underlying logics of the languages, it is not at all clear how this would be done.

Another approach to correctness could be to show that proofs about VDM specifications can be transformed to proofs about the corresponding B specifications. This would at least allow the user to verify that the target specification preserves those properties that were of particular interest in the source specification.

To this end, we propose to (partially) embed both VDM and B into the HOL higher-order theorem-proving system [GTM93]. This would provide a common theorem-proving environment for both languages. There are several possible approaches to embedding the logic of partial functions into

[3] Available from `http://cm.bell-labs.com/cm/cs/what/smlnj/`.

higher-order logic [JM94,KK97,Bur98]. An embedding of VDM into HOL is already under development at IFAD, using the PROSPER toolkit [DCN+00]. Previous work on proof support for VDM can be found in [JJLM91].

There are several papers describing embeddings of formal specification languages into automated theorem-provers for higher-order logic, among them [Cha98] (B into Isabelle/HOL), [ABM97] (VDM into PVS), [GP98] (Z into PVS and HOL) and [BFM99,Muñ99] (B into Coq/PVS).

If partial embeddings from VDM and B into HOL can be produced, then we hope it will be possible (with interaction from the user) to translate HOL proofs about VDM specifications into HOL proofs about the associated B specifications. This would provide some evidence for the correctness of the translation from VDM into B.

References

[ABM97] S. Agerholm, J. Bicarregui, and S. Maharaj. On the verification of VDM specification and refinement with PVS. In J. Bicarregui, editor, *Proof in VDM: Case Studies*, FACIT. Springer-Verlag, 1997.

[Abr96] J.-R. Abrial. *The B-Book: Assigning Programs to Meanings.* C.U.P., 1996.

[B-C00] B-Core (UK) Ltd. The B-Toolkit user manual, 2000. Available online from `http://www.b-core.com/`.

[BDL+99] J.C. Bicarregui, Th. Dimitrakos, K. Lano, T. Maibaum, B.M. Matthews, and B. Ritchie. The VDM+B project: Objectives and progress. In John Fitzgerald and Peter Gorm Larsen, editors, *VDM in Practice*, pages 29–45, September 1999.

[BDMW97] J.C. Bicarregui, J. Dick, B.M. Matthews, and E. Woods. Making the most of formal specification through animation, testing and proof. *Science of Computer Programming*, 29(1–2):55–80, June 1997.

[BDW95] Juan Bicarregui, Jeremy Dick, and Eoin Woods. Supporting the length of formal development: From diagrams to VDM to B to C. In H. Habrias, editor, *7th International Conference on: Putting into practice methods and tools for information system design*, Nantes (France), October 1995. IUT de Nantes. ISBN 2-906082-19-8.

[BDW96] Juan Bicarregui, Jeremy Dick, and Eoin Woods. Quantitative analysis of an application of formal methods. In Marie-Claude Gaudel and Jim Woodcock, editors, *FME'96: Industrial Benefit and Advances in Formal Methods*, pages 60–74. Springer-Verlag, March 1996.

[BFL+94] Juan Bicarregui, John Fitzgerald, Peter Lindsay, Richard Moore, and Brian Ritchie. *Proof in VDM: A Practitioner's Guide.* FACIT. Springer-Verlag, 1994. ISBN 3-540-19813-X.

[BFM99] Jean-Paul Bodeveix, Mamoun Filali, and César Muñoz. A formalization of the B method in Coq and PVS. In *FM'99 – B Users Group Meeting – Applying B in an industrial context : Tools, Lessons and Techniques*, pages 32–48. Springer-Verlag, 1999.

[BM95] Juan Bicarregui and Brian Matthews. Formal methods in practice: a comparison of two support systems for proof. In Bartosek et al., editor, *SOFSEM'95 Theory and Practice of Informatics.* Springer-Verlag, 1995. LNCS 1012.

[BMRA98] J.C. Bicarregui, B.M. Matthews, B. Ritchie, and S. Agerholm. Investigating the integration of two formal methods. *Formal Aspects of Computing*, 10(6), 1998.

[BR93] Juan Bicarregui and Brian Ritchie. Invariants, frames and postconditions: a comparison of the VDM and B notations. In J.C.P. Woodcock and P.G. Larsen, editors, *FME'93: Industrial-Strength Formal Methods*, pages 162–182. Formal Methods Europe, Springer-Verlag, April 1993. Lecture Notes in Computer Science 670.

[Bur98] L. Burdy. A treatment of partiality: Its application to the B method. In *CADE-15 Workshop on Mechanization of Partial Functions.* `http://www.cs.bham.ac.uk/~mmk/cade98-partiality/index.html`, Lindau, Germany, 1998.

[Cha98] L. Chartier. Formalisation of B in Isabelle/HOL. In D. Bert, editor, *Proceedings of 2nd International B Conference*, volume 1393 of *LNCS*, pages 66–83. Springer-Verlag, 1998.

[DCN+00] Louise A. Dennis, Graham Collins, Michael Norrish, Richard Boulton, Konrad Slind, Graham Robinson, Mike Gordon, and Tom Melham. The PROSPER toolkit. In *Tools and Algorithms for the Construction and Analysis of Systems: 6th International Conference, TACAS 2000*, volume 1785 of *LNCS*, pages 78–92, Berlin, Germany, March/April 2000. Springer-Verlag.

[GP98] A.M. Gravell and C.H. Pratten. Embedding a formal notation: Experiences of automating the embedding of Z in the higher order logic of PVS and HOL. Technical Report DSSE-TR-98-6, Department of Electronics and Computer Science, University of Southampton, 1998.

[Gro99a] The VDM Tool Group. The IFAD VDM-SL language. Technical report, IFAD, 1999. Available as a PDF file from `http://www.ifad.dk/`.

[Gro99b] The VDM Tool Group. The IFAD VDM-SL Toolbox user manual. Technical report, IFAD, 1999. Available as a PDF file from `http://www.ifad.dk/`.

[GTM93] M.J. Gordon and editors T.F. Melham. *Introduction to HOL: A Theorem-Proving Environment for Higher-Order Logic*. Cambridge University Press, 1993.

[JJLM91] Cliff Jones, Kevin Jones, Peter Lindsay, and Richard Moore, editors. *mural: A Formal Development Support System*. Springer-Verlag, 1991. ISBN 3-540-19651-X.

[JM94] Cliff B. Jones and Kees Middelburg. A typed logic of partial functions reconstructed classically. *Acta Informatica*, 31(5):399–430, 1994.

[Jon90] Cliff B. Jones. *Systematic Software Development Using VDM*. Prentice-Hall International, Englewood Cliffs, New Jersey, second edition, 1990. ISBN 0-13-880733-7.

[KK97] Manfred Kerber and Michael Kohlhase. Mechanising partiality without re-implementation. In G. Brewka, C. Habel, and B. Nebel, editors, *Proceedings of the 21st Annual German Conference on Artificial Intelligence (KI'97)*, volume 1303 of *LNAI*, pages 123–134. Springer-Verlag, 1997.

[Lan96] K. Lano. *The B Language and Method: A Guide to Practical Formal Development*. Springer-Verlag, 1996.

[LHB+96] P. G. Larsen, B. S. Hansen, H. Brunn, N. Plat, H. Toetenel, D. J. Andrews, J. Dawes, G. Parkin, et al. Information technology — programming languages, their environments and system software interfaces — Vienna Development Method — specification language — part 1: Base language, December 1996.

[LP95] Peter Gorm Larsen and Wiesław Pawłowski. The formal semantics of ISO VDM-SL. *Computer Standards and Interfaces*, 17(5–6):585–602, September 1995.

[MRB98] B. Matthews, B. Ritchie, and J. Bicarregui. Synthesising structure from flat specifications. In D. Bert, editor, *Proceedings of 2nd International B Conference*, volume 1393 of *LNCS*. Springer-Verlag, 1998.

[Muñ99] C. Muñoz. PBS: Support for the B-method in PVS. Technical Report SRI-CSL-99-01, SRI, 1999.