Synthesising structure from flat specifications

BM Matthews, B Ritchie and JC Bicarregui

Published version information

Citation: BM Matthews, B Ritchie and JC Bicarregui. 'Synthesising structure from flat specifications.' Lecture Notes in Computer Science, vol. 1393 (1998): 148-161. Is in proceedings of: 2nd International B Conference (B'98: Recent Advances in the Development and Use of the B Method), Montpellier, France, 22-24 Apr 1998.

DOI: 10.1007/BFb0053359

Synthesising structure from flat specifications

Brian Matthews, Brian Ritchie, and Juan Bicarregui

Rutherford Appleton Laboratory, Chilton, Didcot, OXON, OX11 0QX, U.K.
{bmm,br,jcb}@inf.rl.ac.uk

Abstract. Within the design process, a high-level specification is subject to two conflicting tensions. It is used as a vehicle for validating the requirements, and also as a first step of the refinement process. Whilst the structuring mechanisms available in the B method are well-suited for the latter purpose, the rich type constructions of VDM are useful for the former.
In this paper we propose a method which synthesises a structured B design from a flat VDM specification by analysing how type definitions are used within the VDM state in order to generate a corresponding B machine hierarchy.

1 Introduction

Within the design process, a high-level specification is subject to two conflicting tensions. It is used as a vehicle for validating the requirements, and also as a first step of the refinement process. Whilst the structuring mechanisms available in the B method [1] are well-suited for the latter purpose, the rich type constructions of VDM [7] are useful for the former. Indeed, previous work [2] has shown that although VDM and B are equivalent in theory, in practice, VDM is used for requirements analysis, high level design and validation whereas B places more emphasis on refinement, low level design, and code generation.

Thus the kind of structuring used in the B Method, which is intended to allow compositional development from the specification, can be seen as implementation detail which can obscure the abstract behaviour.

The SPECTRUM project has investigated the benefits of combining VDM and B in the development using VDM for abstract specification and validation (as well as generation of abstract test suites) and B for development of that specification (refinement, verification and code generation). This combination requires a translation between the VDM and B notations during the development.

Typically at the early stages in the development the VDM specification has a data model employing a single module including a single state which is a monolithic value of a complex type (a composite record value whose elements themselves may consist of records, sets, maps or sequences). VDM's language of types and expressions supports the forms of data abstraction which are useful in comprehension and validation.

The required B specification will comprise a hierarchy of abstract machines, each of which contains state values of relatively simple types. This decomposition of the state may support subsequent refinement, developing an understanding of how the design can be achieved.

In other words, when translating a VDM specification to B, complexity in VDM's expression language should be replaced by complexity in AMN's state language, in order to obtain the best from both notations.

In this paper we propose a method which synthesises a structured B design from a flat VDM specification by analysing how type definitions are used within the VDM state in order to generate a corresponding B machine hierarchy.

1.1 Background

VDM and B were first used together in the MAFMETH project [3]. There, a high-level VDM specification was hand-translated into B. MAFMETH showed that using the two methods in this manner gave benefits over "traditional" development approaches. However, translation by hand was error prone: most of the few design errors were introduced at this stage.

The EC project SPECTRUM[1] has been further investigating the interoperability of VDM and B. The project has developed a design lifecycle whereby VDM is used in the early stages of design for high level design, and validation against requirements through prototype generation, and a move to B is performed for the later stages of development towards code, while referring back to the VDM as a test case oracle. Thus an important requirement of the project is an automated translation of VDM into B.

Z to B translation has been carried out in [9] and elsewhere (e.g. [4]). [9] proposed a style similar to "algebraic" specification for translating Z's free type definitions. Though feasible, the resultant B specifications were "unnatural" (in terms of B style), and were difficult to work with in practice. Nevertheless, this style formed a starting point for SPECTRUM, giving a *property-oriented* style of specification, with extensive use of **CONSTANTS** and **PROPERTIES** clauses, and few state variables.

An object-based style of B specification is developed in [8] wherein each object class is realised by a machine that "manages" a state that effectively comprises a set of currently-existing instances of that class. Individual objects in the set are "known" to the rest of the system via their "handles" (or object identifiers) that are created and maintained by the object manager machine. Whilst this *object-manager* approach is more natural within B and one which is more readily analysable by B tools, it can at times be overly complex, and it was not readily apparent how this could be derived from a VDM specification.

Here we propose a hybrid of these two approaches, based on a top-down analysis of the VDM specification. Where a type is used as part of the state, we follow the object based approach; a machine is created to manage the values of that type. Where a type is used only in a declarative way, say as the parameter or result of an operation, then the "algebraic" approach is followed. This translation was tested within SPECTRUM through the case study of translating a simple train control specification from Dassault Electronique ("the Metro specification").

In the remainder of this paper we discuss this translation in more detail. Section 2 gives an overview of the approach. Sections 3 and 4 give some of the technical details of the automated translation, and Section 5 discusses some extensions for further language features. Section 6 gives an example, and the paper is summed up in Section 7.

2 An Analytic Approach

The VDM-SL and B-AMN notations have broadly similar semantics[2] and address similar problem areas (specification of sequential state-based systems). However, the two notations place different emphases on state structure, and on type and value expressivity. VDM provides an expressive type definition language, and a similarly rich language of expressions and functions, but (relative to AMN) its state model is flat: in effect, the state model of a VDM-SL specification typically contains at most a small number of variables, though each variable can have a complex value (of a complex type), and operations are generally viewed as acting on the state as a whole. On the other

[1] EC ESPRIT project 23173, SPECTRUM, is a collaboration between: Rutherford Appleton Laboratory, GEC Marconi Avionics, Dassault Electronique, B-Core UK Ltd, Institute of Applied Computer Science (IFAD), Space Software Italia and Commissariat à l'Energie Atomique. For information about this project contact Juan Bicarregui.

hand, whilst AMN provides powerful constructs for state modularisation, its type and expression notations are impoverished relative to VDM-SL. The state of an AMN specification consists of a relatively large number of state variables, usually of simple types, spread across a number of machines, each of which has "local" operations.

Thus, in translating from VDM-SL to AMN, two problems must be addressed: firstly, how to represent complex types and expressions within a weaker expression syntax, and secondly, how to "infer" a structured state model from an unstructured model.

We propose a new approach to translation from VDM to B which attempts to use the most appropriate style of B specification for the various parts of the VDM specification, resulting in a specification with a more distributed state. A top-down analysis is undertaken to work out which approach is best for each part of the specification. We first consider the *state* in the VDM specification. Consider the state of a representative VDM specification:

state S of
  $n : \mathbb{N}$
  $a : A$
end

where A is a user defined record type:

$A :: a1 : A1$
  $a2 : A2$

In the property oriented approach, since the record type A is translated as an algebraic specification, it is given as the following stateless machine (truncated for brevity):

MACHINE ***A_Type***

SETS ***A***

CONSTANTS

  mk_A, *inv_A*, *a1*, *a2*

PROPERTIES

  $a1 : A \rightarrow A1 \wedge$
  $a2 : A \rightarrow A2 \wedge$
  $mk\_A : A1 \times A2 \rightarrow A \wedge$
  $inv\_A : A \rightarrow BOOL \wedge$
  $(\forall\, xx,yy).(\, xx \in A1 \wedge yy \in A1 \Rightarrow$
    $(a1(mk\_A(xx,yy)) = xx))$
  ...

END

And a top level state machine of the form:

MACHINE ***S***

SEES ***A_Type***

VARIABLES ***n*, *a***

[2] Whilst we recognise that VDM has a denotational semantics over a logic for partial functions, and B has a weakest precondition semantics over classical logic, and thus have some differences in interpretation, we believe that the semantics are sufficiently similar for interoperability to be meaningful and useful. A further part of our work intends to investigate any semantic differences.

INVARIANT

$n \in \mathbb{N} \wedge a \in A$

END

This is a literal translation of the VDM. the *algebraic* VDM record type A is translated into a stateless "property-oriented" machine, which declares the type as a set, a new *mk_A* constant as the constructor function, the fields as projection functions, and a new *inv_A* constant as the invariant function. The behaviour of these constants is defined using properties, in effect giving an algebraic specification of the type. The properties clause soon becomes very large, with complex first-order logic and set theory expressions. Because of the relative weakness of this expression language, these become hard to read, and the support tools for B find such expressions hard to deal with.

The property oriented approach has led to a different "granularity" of the state than would be natural in a B specification. A more "natural" B approach would be to split the record A into its fields, and give a state variable for each, generating a *simple state* specification as follows:

MACHINE ***A_Obj***

VARIABLES

a1, *a2*

INVARIANT

$a1 \in A1 \wedge a2 \in A2$

...

END

And a top level state machine which includes this to represent the "inheritance" of the datatype:

MACHINE ***S***

INCLUDES ***A_Obj***

VARIABLES ***n***

INVARIANT

$n \in \mathbb{N}$

...

END

If the data types *A1, A2* are themselves record types they can be broken down further into similar machines. Thus we build a hierarchy of machines which preserve the structure of the VDM specification, but have a finer state granularity. This specification is much clearer and easy to work with, exploiting as it does the strength of B and B tools in manipulating machine state and generalised substitutions.

If the state has an aggregate type of records, such as set or map, then the appropriate B specification is different. For example, if the state is of the form:

```
state S of
   n : ℕ
   a : A-set
end
```

with A as before, then an "object manager" approach is more appropriate:

MACHINE ***A_Mgr***

SETS ***A_Ids***

VARIABLES

$aids$, $a1$, $a2$

INVARIANT

$aids \subseteq A\_Ids \wedge$
$a1 \in aids \rightarrow A1 \wedge$
$a2 \in aids \rightarrow A2$

...

END

Typically, an object manager machine will also include some basic operations for inspecting and manipulating the variables; for example equality should be defined on the value of the attributes rather than on the identifier. In this paper, these are omitted for clarity.

A top level state machine which includes this to represent the "inheritance" of the datatype within the VDM record:

MACHINE ***S***

INCLUDES ***A_Mgr***

VARIABLES ***n*, *a***

INVARIANT

$n \in \mathbb{N} \wedge a \subseteq aids$

...

END

Where the attribute being modelled is a sequence or map of records, the variable a here would be a sequence or map of *aids*. Again, if either of *A1, A2* are themselves record types, then they should themselves be made into objects. However, since they are accessed from an object manager machine, they should be implemented as object managers themselves.

In this fashion we can analyse the specification and give appropriate definitions for each record type. This is formalised in the following sections.

3 Preprocessing the VDM Specification

In this top-down analysis, the VDM spec is preprocessed to decide how to best translate each record type. Two sets are declared, *Simple* and *Manager* which represent those types which should be represented as a simple object machine, or as an object manager machine respectively. The analysis is a simple recursive procedure. First find the state, for example:

```
state $Example$ of
   $a : T$          // where T contains no record type
   $b : R$          // where R is a record type
   $c : S$-set      // where S is a record type
end
```

T is any type which does not contain a record type, and it can be basic, user defined or an aggregate type (set, map or sequence) which do not have record types as members.

Any record types which are referenced directly, such as R, are added to the set *Simple*, and any record which is part of an aggregate, such as S, are added to the set *Manager*.

This process is repeated for each record type $R \in$ *Simple*, and $S \in$ *Manager*, with the additional conditions:

1. if $S \in$ *Manager*, then $S \notin$ *Simple*;
2. if record R occurs in a field of $S \in$ *Manager*, then $R \in$ *Manager*.

Thus if a record type S is added to *Manager*, it must be removed from *Simple*, and any records referenced from S must also be added to *Manager*.

4 Inferring a Structured Model

When the preprocessing is complete, then machines can be generated. All record types in *Simple* are translated as simple structured object machines, while types which are in *Manager* are translated as object manager machines. Any record types not in either of these sets are translated as property oriented stateless machines.

Thus there are three cases to consider of records to translate: a *Simple* record with a reference to a *Simple* record within it; a *Simple* record with a reference to a *Manager* record; and a *Manager* record with a reference to a *Manager* record. The final case of *Simple* within *Manager* will not occur.

In the first case, if for example, P $\in$ *Simple* such that:

$$\begin{array}{ll} P :: r1 : T & \text{// non record type} \\ \quad\; r2 : R & \text{// record type such that R} \in \textit{Simple} \end{array}$$

then we would generate the simple state machine:

MACHINE ***P_Obj***

INCLUDE ***r2.R_Obj***

VARIABLES

r1

INVARIANT

r1 $\in$ *T*

END

where *R_Obj* is a simple state machine as in Section 2 above. Note that the machine *R_Obj* is included into this machine with a renaming, using the convenient name *r2*. This machine may well be used elsewhere in the specification, and each including machine needs a unique copy.

For the second case, if P $\in$ *Simple* such that:

$$\begin{array}{ll} P :: r1 : T & \text{// non record type} \\ \quad\; r2 : S & \text{// record type such that S} \in \textit{Manager} \end{array}$$

then generate the simple state machine:

MACHINE ***P_Obj***

INCLUDE ***S_Mgr***

VARIABLES

$r1, r2$

INVARIANT

$r1 \in T \wedge r2 \in sids$

...

END

where *S_Mgr* is an object manager machine for the type S, as given in section 2.

Note that there is no renaming carried out here. There is only one object manager machine in the system, and all references should be to that machine. However, there is an issue here: the rules of composition in B allow a machine to be included in only one other machine. Thus if the manager record is referred to from more than one record type, this condition may be broken. The resolution of this problem would be to break down the *S_Mgr* machine into two, with a simple machine declaring the abstract set *S_Ids*, representing the object identifiers, and a manager machine. The manager machine is then included with renaming, and the set is accessed via the SEES construct. This allows the same set of object identifiers to be used across different object managers.

For the third case, if P $\in$ *Manager* such that:

```
P :: r1 : T        // non record type
     r2 : S        // record type such that S ∈ Manager
```

then generate the object manager machine:

MACHINE ***P_Mgr***

INCLUDE ***S_Mgr***

SETS ***P_Id***

VARIABLES

$pids, r1, r2$

INVARIANT

$pids \subseteq P\_Id \wedge$

$r1 \in pids \rightarrow T \wedge$

$r2 \in pids \rightarrow sids$

...

END

Other types and record types, which are not accessed via the state model are treated by a property oriented translation.

It may also be necessary to give property oriented translations as well as state based ones for certain records; they are used as input/output to functions for example. The analysis of the specification can be extended to cover this eventuality.

5 Handling further language constructs

The translation presented so far concentrates on the different data models of VDM and B. To present a full translation, other aspects of the languages need to be considered, especially the type and expression language, and the operation and function language. These are not the main subject of this paper, which is considering the data model, so we shall only consider them briefly.

The richness of the VDM-SL expressions and types makes direct translation to AMN difficult. Many expression constructs are not easily expressed in AMN's properties notation; and result types of operations and functions may be of compound types (e.g. tuples) which are awkward to represent and work with in AMN. On the other hand, there are reasonably obvious translations from the VDM expression syntax to AMN's generalised substitutions. Though the latter are state transformers and not functional expressions, this suggests a translation approach that wherever possible "re-interprets" VDM functions as AMN operations which do not change the state.

A VDM specification which makes heavy use of functions and expressions translates to a heavily property-oriented AMN specification which is ungainly to work with in practice, generating difficult proof obligations both for self-consistency and in subsequent refinement and implementation.

Part of the principle of our approach is to translate VDM functions into B operations as much as is practically possible. Some analysis of functions is thus required to determine which part of the state they are applied to, and which machine to enter them into. Some of these are fairly straightforward. If a function has a signature of the form:

$record\_fun1\,(rin : Record, a_1 : t_1 \ldots a_n : t_n)\ rout : Record$

pre ...

post $rout = \mathsf{mk}\text{-}Record\ (P_1(rin, a_1 \ldots a_n), \ldots, P_m(rin, a_1 \ldots a_n))$;

where *Record* is a record type in the set *Simple*, then it is reasonable to translate this function as an operation in machine *Record_obj*, with the declaration of the form:

```
record_fun1( a_1, ..., a_n )   ≙
    PRE
        $a\_1 \in t\_1 \wedge \ldots\ a\_n \in t\_n$
        ...
    THEN
        $r\_1 := P\_1(a\_1 \ldots a\_n)$
        ...
        $r\_m := P\_m(a\_1 \ldots a\_n)$
    END ;
```

where *r_1, ... r_m* are the variables of the record machine. The expressions *P_1, ..., P_m* which give the transformation of variables may also involve the variables of the machine, and also may themselves be operations (with consequent changes in syntax), especially if the variables are themselves record types, and thus provided by an included machine.

6 A Worked Example.

To illustrate the different approaches, we consider the following small example of a VDM-SL specification. This has a state *Metro*, which has a train component. The train itself is a record with two components, *motion*, representing the status of the train's motion, and the current speed of the train. The invariant on the train states that when the train is stopped, its speed is zero. We also provide a function for braking.

state

```
state $Metro$ of
   $train : Train$
end
```

types

$$Train :: motion : MOTIONSTATUS \\ \qquad\qquad speed : \mathbb{N}$$

inv mk-$Train\,(mm, ss) \triangleq$
  $(mm = \text{STOPPED} \;\Rightarrow\; ss = 0)$

$brake\;(tin : Train)\; tout : Train$
pre $tin.motion \in \{\text{ACCELERATING}, \text{STEADY}\}$
post $tout =$ mk-$Train\,(\text{DECELERATING}, tin.speed)$;

Under the property oriented translation, this becomes two machines, one with the state model:

MACHINE ***Metro_Type***

SEES

Train_Type

VARIABLES

train

INVARIANT

$train \in Train$

END

The other machine is a stateless property-oriented specification, with the invariant represented as a complex first-order logic formula in the properties clause, and the *brake* represented as a stateless operation:

MACHINE ***Train_Type***

SETS ***Train0***

CONSTANTS

Train, *mk_Train*, *inv_Train*, *motion*, *speed*, *init_Train*

PROPERTIES

$Train \subseteq Train0 \;\wedge$
$mk\_Train \in MOTIONSTATUS \times \mathbb{N} \rightarrow Train0 \;\wedge$
$inv\_Train \in Train0 \rightarrow BOOL \;\wedge$
$motion \in Train0 \rightarrow MOTIONSTATUS \;\wedge$
$speed \in Train0 \rightarrow \mathbb{N} \;\wedge$
$(\forall\, mm,ss).(\; mm \in MOTIONSTATUS \wedge ss \in \mathbb{N} \;\Rightarrow$
$\quad (inv\_Train(mk\_Train(mm,ss)) \;\Leftrightarrow\; ((mm = stopped) \;\Rightarrow\; (ss = 0)))) \;\wedge$
$\ldots$

```
OPERATIONS

    $tout \longleftarrow brake(\ tin\ )\quad \widehat{=}$
        PRE
            $tin \in Train \wedge$
            $motion\ (\ tin\ ) \in \{\ accelerating\ ,\ steady\ \}$
        THEN
            $tout\ :=\ mk\_Train\ (\ decelerating\ ,\ speed\ (\ tin\ )\ )$
        END
END
```

In this latter machine, all functions are represented as stateless operations, which permits a more expressive syntax than translating them as properties. Nevertheless, this is an awkward machine to manipulate in the B-Method, and while perfectly valid in the language, is not a "natural" approach in B.

However, in the top-down method, the train record type is translated into a machine with state variables.

```
MACHINE    Train_Obj

VARIABLES

    $motion\ ,\ speed$

INVARIANT

    $motion \in MOTIONSTATUS \wedge$
    $speed \in \mathbb{N} \wedge$
    $(motion = stopped) \Rightarrow (\mathrm{speed} = 0)$

OPERATIONS

    $brake\quad \widehat{=}$
        PRE
            $motion \in \{\ accelerating\ ,\ steady\ \}$
        THEN
            $motion\ :=\ decelerating$
        END

END
```

This machine then INCLUDEd into the top-level Metro specification, together with a renaming. This is a much more "natural" B machine, using more state variables of simple types, which are close to machine types. This machine can be easily included into a continuing B development, and resulting proof obligations more easily expressed and discharged.

7 Conclusions

We have demonstrated the feasibility of synthesizing structured B specifications from VDM. However, the translation is not yet complete. Further analysis is still required to provide an account of the translation of VDM functions and operations, and of VDM's full type and expression language. This is the subject of ongoing research.

The translation presented here automates the extraction of *design information* from the VDM-SL specification, by deriving a finer-grained state model. As the design process requires intelligent insight, the resultant design may not be in the form the user desires, and therefore an element of user judgement is still required to determine which elements of this design are appropriate. However, as the design elements have been automatically generated, the useful elements have been gained at no cost.

Acknowledgements

We would like to thank our partners on the SPECTRUM project and the Commission of the European Union for their support.

References

1. J-R. Abrial, *The B-Book: Assigning Programs to Meaning*, Cambridge University Press, 1996.
2. J.C. Bicarregui, and B. Ritchie, *Invariants, frames and preconditions: a comparison of the VDM and B notations*, in Proceedings of Formal Methods Europe'93, Lecture Notes in Computer Science, Vol. 670, ed. J. Woodcock and P G Larsen, Springer-Verlag 1993.
3. J.C. Bicarregui, A.J.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) p.55-80 Elsevier-Science, (June 1997)
4. R.F. Docherty, *Translation from Z to AMN*, proceedings 7th International Conference on Putting into Practice Methods and Tools for Information System Design, ed. H. Habrias, ISBN 2-906082-19-8, 1995.
5. J. Draper (ed), *Industrial benefits of the SPECTRUM approach*, SPECTRUM Project External Deliverable 1.3, 1997.
6. ISO, *"ISO/IEC 13817-1 Information Technology - Programming Languages, their environments and system software interfaces - Vienna Development Method - Specification Language. Part 1:Base Language"*, 1996.
7. C.B. Jones, *Systematic Software Development Using VDM*, 2nd Edition, Prentice-Hall, 1990.
8. K. Lano, *The B Language and Method: a guide to practical formal development*, Springer-Verlag 1996.
9. B. Ritchie, J.C. Bicarregui, and H. Haughton, *Experiences in Using the Abstract Machine Notation in a GKS Case Study*, Proceeding of Formal Methods Europe '94, Naftalin, Denvir, Bertran (Eds), LNCS 873, Springer-Verlag, 1994.