From VDM Specifications to Functional Prototypes

Paulo Borba[*], Silvio Meira[†]

Departamento de Informática

Universidade Federal de Pernambuco

P.O. Box 7851, 50739, Recife, PE, Brazil

1 Abstract

This paper describes, mostly in an informal way, a method for translating formal, model-based specifications, into functional prototypes. Such prototypes can be used to validate software system specifications in the very early stages of the design process.

We show how to translate the executable subset of the VDM language into Lazy ML, establishing the translation schemes and requirements needed for an automated translation process.

Here we are not concerned with implicit specifications, but indicate, in places, what has to be done in order for such specifications to be automatically translated into a functional program.

A small example is presented and, being itself part of a translation system from VDM to LML, gives the reader a good idea of the power of the approach.

Keywords:

Formal Methods, Formal Specification, Rapid Prototyping, Software Validation, Animation Techniques, Functional Languages.

2 Introduction

This article is concerned with the animation of model-based formal specifications through functional prototypes. In particular, we consider how VDM specifications can be transformed into

[*]Supported by a CNPq research scholarship, Brazil. Email: Paulo.Borba@prg.ox.ac.uk. Currently at PRG–Oxford University, 11 Keble Road, Oxford England, OX1 3QD. FAX: +44 865 273839. Tel: +44 865 273846.

[†]Research funded by CNPq grant 300133/85-CC. Email: Silvio@di.ufpe.br.

Lazy ML (LML) prototypes; neither of the two languages is necessary here, as any other pair of model-based specification and typed functional languages could have been used, with minor (mainly syntactical) modifications.

The increasing use of formal techniques in software development needs not to be stressed here. But it should be clear to all that this will only be possible if the proper computing environments and tools are available to support the mechanical parts of more mathematical software development processes.

In a simplified, analysis-design-implementation model of software construction, the prototyping phase corresponds to the implementation. Some aspects concerned with implementation, such as efficiency and the definition of sophisticated interfaces for the user of the system are not taken into account when constructing a prototype.

The generation of prototype code from a specification, in this case, is semi-automatic. The pre-defined types of VDM are implemented as types of the target functional language and used as the basis for the construction of the prototype.

Pre- and post-conditions are automatically translated and when operations are executed, preconditions are tested first. After the execution, post-conditions can also be tested, thus verifying the consistency of the specification with respect to the implementation, for particular cases. For some applications, it may be useful to make condition checking optional, in order to reduce execution time and to explore the behaviour of the system when the conditions are not valid.

Translation of post-conditions and *associated functions* into runnable code may involve some transformations which must be carried out by the designer in order to reduce the level of abstraction and to allow automatic coding. These transformations, however, are easily carried out.

The verification of the semantic equivalence between different levels of abstraction is not imposed by the environment. It is up to the designer to verify the consistency of transformations but the system may assure consistency by testing pre- and post-conditions, as well as type invariants, during execution.

A generic interface is used to test the prototype and there is no need to write code for input and output, which is automatically provided by the environment according to parameter and result types. The internal state of an application can be checked at any time and many versions of the state may be stored to allow different kinds of test to be carried out simultaneously.

3 VDM and Functional Languages

VDM was originally developed in the IBM Vienna Research Laboratories, starting back in 1973 and it has evolved to be one of the most well established methods for rigorous software development. From sometime in the past to this date, the method and its formal specification language have shared the same name; a complete account of the language is given in [Jon90].

VDM is based in first-order logic and set theory. The method suggests the development of software through successive refinements. From a very abstract initial specification, others are derived until, after data and operation refinements, the final implementation is constructed. The consistency, that is to say, the semantic equivalence between the levels of refinement must be rigorously verified.

The type system is based on set theory. Types are sets of values. In addition to basic types, such as Integer and Boolean, VDM offers other type constructs and related operations. These types, which include sets, mappings and sequences, are used for data structuring.

A mathematical model is constructed for the software, defining the state of the application and operations over it. Auxiliary functions may also be defined.

An operation is defined by specifying its pre- and post-conditions. Furthermore, operations must indicate which components of the state they read and/or write. Functions do not affect the state.

There is no data encapsulation and a model defines a concrete object upon which the operations actuate. There are no modularization mechanisms and reusability is difficult. One can only reuse functions and pre- and post-conditions in a specification.

VDM is one of the most used formal specification language these days, both in industry and research, and one of the few for which there is a BSI standard [BSI89] and specialized conferences; a number of tools to support the development of specifications also exists, notably the *mural* [Moo90] proof assistant.

One very interesting connection to be explored is the relationship between model-based specification and functional languages. The latter have been once put forward as specification languages in their own right, given the abstractness intrinsic to well-written, properly typed, lazy, higher-order programs. Turner makes that point in [Tur84] and a good example of a language in which such style for specifications can be exercised is shown in [Tur85]. A number of researchers still consider that being the case, as it is pointed out in [Mer87].

Although one might not accept that functional programs are "executable specifications", no one can doubt that they are indeed executable. That does away with one of the primordial characteristics of abstract, not necessarily executable, specifications [HJ89], where one does not have to deal with the intricacies of the concrete object and algorithms needed to make a piece of "code" executable by a computer.

The view taken in this work is that the pair formed by model based specification languages and functional programming languages can offer a good deal to programmers whose interest is to have a correct prototype, built from a high-level, logical (not necessarily executable) specification of the piece of software they are constructing. However, we do not discuss the details of deriving a concrete specification from a more abstract, as considered in [Mor90] and [BMPP89]. Instead, we show how to derive functional prototypes from a class of concrete VDM specifications. Neither VDM nor LML are ideal languages to conduct such a project. But they are real and used throughout the world; this is why we chose them.

We assume the reader has a working knowledge of VDM (or model-based formal specifications in general) and LML (or functional programming) in order to properly understand this article. Nevertheless, concepts from both VDM and LML, when they depart from the common knowledge, are explained when used.

4 Translation Schemes

We can translate structures of the specification language into equivalent ones in the prototyping language. The translation schemes are mainly syntactic and can be implemented to make the translation process automatic. The process was formalised (by means of a VDM specification of a system to automatically translate a VDM specification into LML) and can be proved correct, but we will present the schemes in an informal and intuitive way, since a complete formalization is out of the scope of this paper.

The basic idea to prove the correctness of the translation process is to verify whether the properties about the VDM operations associated to its mathematical objects (see [Jon90]) are satisfied by the corresponding operations implemented in LML (see Figure 1). In this way, proving that VDM operations in the described form are implemented by LML functions is straightforward. For operations over sequences, the proof of correctness is presented in [Bor91].

Although we use concepts and notations associated with the specification language VDM

(Vienna Development Method) [Jon90] and the functional language LML [AJ88], the derivation process is sufficiently general to be applied to other model based specification and functional languages, as described in [JS90] and [BM89].

Initially, we show how types and expressions are translated. Then, we discuss how VDM functions and operations can be modeled by LML functions. We only take in consideration *explicitly defined* operations (see [Jon90]), which can be directly translated into LML since they specify not only *what* the operation does, but also *how* it is done. *Implicit* operations should be first refined into *explicit* ones from which the prototype can be generated. The class of operations considered for translation is precisely defined in Section 4.3, where the syntactical form of the operations in this class is described.

4.1 Types

Type translation is simplified by implementing the usual mathematical objects (sets, sequences and maps) used in model based specification languages as abstract data types in the functional language. This is an extension of the functional language to increase the level of abstraction of data types and to decrease the semantic gap between the two languages. In this way, type translation is direct and there is no need to perform data type refinement.

For example, VDM sets are implemented by the LML module shown in Fig. 1, where `delete` is used to remove an element from a set. Note that a specific equality operator has to be supplied, otherwise it would not be possible to model, for example, sets of sets.

4.1.1 Basic Types

Basic types of the specification language are translated into the corresponding built-in LML types. Type constructors for sets, sequences and maps are represented by equivalent ones implemented in the functional language. We will represent VDM objects in *italics* and associated LML objects in `teletype`. The association between type constructors in the languages is established by Table 1, where X and Y are type expressions. `Set`, `Seq` and `Maps` are, respectively, the type constructors for sets, sequences and maps defined in LML. The types $\mathbf{N}_1$ and $\mathbf{N}$ are subtypes of $\mathbf{Z}$, and are defined as types with invariant

```
module

export emptyset, member, geneset, union, setequal;

rec

    type Set *a = set (List *a) (*a -> (*a -> Bool))

--  emptyset: (*a->*a->Bool)->(Set *a)

and emptyset eq = set [] eq

--  member: *a->(Set *a)->Bool

and member e (set [] eq)    = false

||  member e (set (h.t) eq) =

      (eq e h) orelse member e (set t eq)

--  geneset: *a->(Set *a)->(Set *a)

and geneset e (st as (set l eq)) =

      if member e st then st

      else set (e.l) eq

--  union: (Set *a)->(Set *a)->(Set *a)

and local notmem st e = notmember e st

    in

      union (st as (set l1 eq1)) (set l2 eq2) =

        set (l1 @ (filter (notmem st) l2)) eq1

    end

--  setequal: (Set *a)->(Set *a)->Bool

and setequal (set (h.t) eq) st =

      if member h st then

        setequal (set t eq) (delete st h)

      else false

||  setequal (set [] eq) (set [] eq1) = true

||  setequal (set [] eq) (set l  eq1) = false

end
```

Figure 1: VDM Sets Implemented in LML.

VDM	LML
$\mathbf{Z}$	`Int`
$\mathbf{B}$	`Bool`
Char	`Char`
Token	`List Char`
$\mathsf{set\ of}\ X$	`Set X`
$\mathsf{seq\ of}\ X$	`Seq X`
$\mathsf{map}\ X\ \mathsf{to}\ Y$	`Maps X Y`

Table 1: Type Constructors.

$\mathrm{N} = \mathbf{Z}$

where

$inv\text{-}\mathrm{N}(i) \quad \triangleq \quad i \geq 0$

$\mathrm{N}_1 = \mathbf{Z}$

where

$inv\text{-}\mathrm{N}_1(i) \quad \triangleq \quad i \geq 1$

which we will see how to properly translate later in the paper (see 4.1.6).

4.1.2 Type Abbreviation

Type abbreviations in VDM are translated to macro definitions in LML, since this is the only available way to associate a name to an already defined type. Thus, the set of declarations

$X = \mathsf{set\ of}\ \mathrm{N}$

$Y = \mathsf{seq\ of}\ (\mathsf{set\ of}\ \mathbf{B})$

$Z = \mathsf{map}\ \mathrm{N}\ \mathsf{to}\ (\mathsf{seq\ of}\ \mathrm{N})$

is directly translated to the sequence of LML declarations

```
#define X (Set Int)

#define Y (Seq (Set Bool))

#define Z (Maps Int (Seq Int))
```

using type constructors, as discussed in the previous subsection, with `#define` defining a macro. With those definitions, for instance, any occurrency of `Y` is replaced by `(Seq (Set Bool))`.

In a language with a richer type environment, such as SML [HMM86], abbreviations are translated to equivalent constructs, defining new names for a type.

4.1.3 Composite Objects

A composite type (like records in Pascal) is translated into an algebraic data type with only one constructor and a tuple type (corresponding to the composite type components) as the constructor's argument. Composite types are implemented as algebraic data types to assure that type names are part of the type. Then, VDM composite objects such as

$$
\begin{array}{lll}
OC :: & a & : A \\
& \ldots & \\
& z & : Z
\end{array}
$$

are represented by

```
type OC = OC_con  (A # ... # Z)
```

where A, …, Z are type expressions, a, …, z are field identifiers and # is LML's type constructor for tuples. In particular, composite objects like

$$OC' :: a : A$$

are translated to

```
type OC´ = OC´_con A.
```

All operators available in the specification language for composite types may be automatically generated in the prototyping language. For instance, selectors can be implemented by

```
    a (OC_con (a1, b1,..., z1)) = a1
and z (OC_con (a1, b1,..., z1)) = z1.
```

VDM's *make* functions are replaced by type constructors. So, expressions like

$$mk\text{-}OC(e_a, \ldots, e_z),$$

denoting an element of type OC will be represented as `OC_con (ea, ..., ez)`, where $e_a, \ldots, e_z$ are expressions.

The equality operator for a composite object has to be defined when the type expression is composed by at least one type whose elements cannot be compared by the standard equality operator of LML, represented by `=`. Otherwise, `=` itself can be used as the equality operator for the composite object. The elements of the abstract data types implemented in LML (sets, maps and sequences) cannot be compared by the standard equality and have specific equality operators, represented by `Set_eq`, `Map_eq` and `Seq_eq`. Also, any type defined in terms of these type constructors cannot be compared by the standard equality operator and need a specific one.

For types defined like OC, the equality operator is implemented by

```
OC_eq (OC_con (a1, ..., z1)) (OC_con (a2, ..., z2)) =
   (A_eq a1 a2) andalso
   ...
               andalso
   (Z_eq z1 z2)
```

where `A_eq`, ..., `Z_eq` are the equality operators associated to the types of the components of the composite object.

With this implementation, we can naturally translate function definitions that use pattern-matching over elements of a composite object. For instance, the VDM function specification

$$April? : Data \to \mathbf{B}$$

$$April?(mk\text{-}Data(d, m, y)) \quad \triangleq \quad m = 4$$

can be trivially translated to

```
April? (Data_con (d,m,y)) = (m = 4)
```

assuming that $Data$ is implemented by

```
type Data = Data_con (Day # Month # Year)
```

where `Day`, `Month` and `Year` have a standard interpretation.

In a similar way to composite types, product types can be also implemented by tuples in LML, so we omit the details here (see [Bor91]).

4.1.4 Union Types

Union types —types constructed by the union of other types— are directly translated into algebraic data types with constructors (tags) corresponding to each component of the union type. A union type definition like

$A = B \mid C \mid \ldots \mid Z$

is represented by

```
type A = A_B B + ... + A_Z Z.
```

Considering the VDM type definitions above, an element of type B can be used as an element of type A and, for $e: A$, if is-$B(e)$ (this predicate is **true** if e is a member of type B) we can use e as an element of type B. This is not so in LML without explicit use of constructor and projection functions, so we need to define those. The is- function is defined by the equations below

```
    A_is "B" (A_B x) = true
    ...
||  A_is "Z" (A_Z x) = true
||  A_is _    _      = false
```

where the first parameter is a type name and the other is an expression of type `A`.

The projection functions are implemented in the following way:

```
    A_proj_B (A_B x) = x
and A_proj_Z (A_Z x) = x.
```

The equality operator for a union type should be defined when the union is formed by at least one type whose elements cannot be compared by the LML equality operator. Otherwise, the standard operator can be used to compare elements of the union type. Anyway, the operator for the type A, defined above, can be implemented as

```
   A_eq (A_B x) (A_B y) = B_eq x y
|| ...
|| A_eq (A_Z x) (A_Z y) = Z_eq x y
|| A_eq _       _       = false
```

where `B_eq`, ..., `Z_eq` are the equality operators associated to the types `B`, ..., `Z`.

Other variations of union types like optional and enumeration types [Jon90] can be implemented in a very similar way to the implementation of union types (see [Bor91]). For example, an enumeration type definition like

$BasicType = \{\textsc{Nat}, \textsc{Real}, \textsc{Bool}\}$

stating that *BasicType* is formed by the elements NAT, REAL, and BOOL, can be considered as the union of three types, each one having only one element — one of the elements of *BasicType*. In this way, we can implement the definition above as

```
rec type BasicType = BasTpNAT NAT + BasTpREAL REAL + BasTpBOOL BOOL
and      NAT = NATcon_
and      REAL = REALcon_
and      BOOL = BOOLcon_
```

with BasicType being a defined type where NAT, REAL and BOOL are algebraic types containing only one element.

4.1.5 Given Sets

Given sets —types without constructive representation— are usually implemented as basic types of the functional language. For example, the implementation of a given set definition like

GS , is not yet defined

is chosen from a number of possible constructive representations, one of which could be

```
#define GS (List Char)
```

where we use list of characters to implement GS.

4.1.6 Type Invariants

Type definitions with invariant such as

$A = \mathbf{N}$

where

$inv\text{-}A(i) \quad \triangleq \quad i \geq 1 \wedge i \leq 5$

defining type A as the set $\{1, \ldots, 5\}$, are translated to the base type

```
#define A Int
```

where Int is LML's rough equivalent of $\mathbf{N}$ (natural numbers).

Usually, the type systems of most functional languages do not support type invariants. In the prototype, invariant checks are done at run time. A type invariant (the part of the definition following `where`) is translated to a function, as shown in 4.3.1, and any operation that receives as parameter an element of a type with invariant first applies the invariant (function) to check if the received parameter satisfies the invariant. Operations whose result types have invariants should give results satisfying the invariants —in VDM, this generates a *proof obligation* (see [Jon90]). But if this obligation is not discharged, which can be common in the prototyping stage, the result can be checked at run time by using the translated invariant, but one still have to discharge the obligation later.

4.2 Expressions

Operations over sets, sequences and maps (e.g. set union, map domain) are directly translated into the associated ones implemented in the functional language as operations of the abstract data type. A restricted form of comprehension over those objects is implemented by a higher order operator. For example, Table 2 shows how some set expressions are translated to LML. Consider that X and Y are type expressions; e, e_1, ..., e_n: X; `eq` is the equality operator for elements of X, and `eq´` for elements of Y; i, j: $\mathbf{N}$; R, S: $\mathsf{set\ of}\ X$; T: $\mathsf{set\ of}\ (\mathsf{set\ of}\ X)$; $p: X \to \mathbf{B}$, $f: X \to Y$. The `choice` function returns one element of a non-empty set. The equality operator for the elements of a set is provided as an argument to the functions that create sets: `emptyset`, `singleton` and `makeset`. Auxiliary constructions such as "if-then-else", "let-in" and "case" also have counterparts in the functional language.

The logical operators (e.g. and, not, implies) of the specification language are easily translated into pre-defined operators in the functional language, as shown in Table 3, where e and f are boolean expressions, X is a type expression, S: $\mathsf{set\ of}\ X$ and $p: X \to \mathbf{B}$. However, due to the difference[1] between VDM and LML logical operators, the translation of $\wedge$, $\vee$, and $\Rightarrow$ requires the discharge of a proof obligation.

The translation of $e \wedge f$ to `e andalso f` requires that $e \neq \perp \vee f \neq \mathsf{false}$. To translate $e \vee f$ to `e orelse f`, and $e \Rightarrow f$ to `if e then f else true` we need to verify whether $e \neq \perp \vee f \neq \mathsf{true}$ is valid. For example, the expression

$$s(i) > 0 \wedge i \in \mathsf{dom}\, s,$$

where $s \in \mathsf{seq\ of}\ \mathbf{Z}$ and $i \in \mathbf{Z}$ (both are different of $\perp$), cannot be translated since when $i \notin \mathsf{dom}\, s$ we have that $s(i) = \perp$ and $(s(i) > 0) = \perp$. Furthermore, we cannot prove that $(i \in \mathsf{dom}\, s) =$

[1] The binary logical operators in the functional language are strict in the first argument, differently from the operators in the specification language. For example, $\perp$ `andalso false =` $\perp$, but $\perp \wedge \mathsf{false} = \mathsf{false}$, where `andalso` is LML conjunction operator.

VDM	LML
$\{\}$	`emptyset eq`
$\{e\}$	`singleton e eq`
$\{e_1, e_2, \ldots, e_n\}$	`makeset [e1,e2,...,en] eq`
$\{e \mid e \in S \bullet p(e)\}$	`setcomp S p`
$\{f(e) \mid e \in S \bullet p(e)\}$	`setcompf f S p eq´`
$\{i, \ldots, j\}$	`i setint j`
$e \in S$	`e member S`
$R \cup S$	`R union S`
$R - S$	`R difset S`
$R = S$	`R setequal S`
card S	`card S`
$R \subset S$	`R psubset S`
let $e \in S$ in e_1	`let e = choice S in e1`
$\bigcup T$	`distunion T eq`

Table 2: Set Operators.

VDM	LML
$e \wedge f$	`e andalso f`
$e \vee f$	`e orelse f`
$\neg e$	`not e`
$e \;\Rightarrow\; f$	`if e then f else true`
$\forall x \in S \cdot p(x)$	`forall S p`
$\exists x \in S \cdot p(x)$	`exists S p`

Table 3: Logical Operators.

true, for any s and i. On the other hand, the expression

$$i \in \mathsf{dom}\, s \wedge s(i) > 0$$

can be translated to

```
(i member (domseq s)) andalso ((s aplseq i) > 0)
```

because $(i \in \mathsf{dom}\, s) \neq \bot$ when $s \neq \bot$ and $i \neq \bot$.

The logical quantifiers (e.g. for all, exists) are translated into functional versions when the base set is finite. If this is the case, the expression $\forall x \in S \cdot p(x)$ can be translated to `forall S p`, when the base set is a function of the parameters and input state of the operation. Otherwise, semantic preserving transformation rules should be used to eliminate them, as will be discussed in the next section.

4.3 Functions and Operations

In this section we describe a method for translating functions and operations. Only explicitly defined operations are translated. More abstract specifications should be refined into specifications written in a explicit style. For this end, we can proceed in a completely *ad hoc* way, where we find a more concrete specification (explicitly defined) and prove it correct with respect to the more abstract (implicitly defined), or we can use a complete and consistent set of semantic preserving rules to aid the process, in such a way that we derive a correct implementation from the specification. This can be done by means of refinement or transformation rules, as described in [Mor90] and [BMPP89].

In fact, for our aims, the effort needed to do such a refinement should be smaller than usually needed in the examples of [Mor90], for example, since we are only interested in a prototype rather than in an efficient implementation and the programming language that we use is semantically closer to the specification language.

4.3.1 Functions

Explicitly defined functions can be easily translated, directly by translation of the expression body. In this way, the VDM function specification

$$f : A \times \ldots \times X \to Z$$

$$f(a, \ldots, x) \quad \triangle \quad a \ldots x$$

with parameters a: A, ..., x: X giving result e: Z becomes

```
f (a,...,x) = e
```

in the prototype, where e is an expression. There are no major problems when translating VDM's $a \ldots x$ into the corresponding LML expression and this can be done automatically.

Functions defined in a explicit style having the post condition explicitly establishing the function result in terms of the arguments, i.e., functions specified in the following syntactic form

$f \quad (a: A, \ldots, x: X)\ z: Z$

pre $a \ldots x$

post $z = g(a, \ldots, x)$

are implemented by definitions like

```
f (a,...,x) =
 if pre-f(a,...,x)
 then g(a,...,x)
 else error-value
```

where `pre-f(a,...,x)` is the LML expression corresponding to the VDM operation pre-condition. If this condition is not satisfied, an error value is returned, represented here as `error-value` that can be a distinguished value of `Z`.

We use the simplest way to indicate that the pre-condition was violated. However, in some cases it may be more useful if the system could signal the error and try to execute the operation (function) with the arguments which do not satisfy the pre-condition, in order to analyse the behaviour of the specification, since perhaps the pre-condition may be too strong. Another option is to place the user in a debugger to investigate the reasons which caused the error. These facilities are not available in the current implementation, but could be added easily.

If a function specification is not in the described syntactic form, the translation cannot be automatic and the programmer has to transform the specification into an equivalent one in the required syntactic form.

Another way to implement the f function above uses conditional patterns, as guards to the equation defining the function. This is shown in the following definition:

```
f (a,...,x) & (pre-f(a,...,x)) = g(a,...,x).
```

In this way, `f` is only defined for argument values that satisfy the condition `pre-f(a,...,x)`.

The pre-condition is always directly translated. The post-condition can only be directly translated if it is in the special form shown above. Despite the restriction required to translate a post-condition, in a number of cases only very simple rules are needed to transform the condition to the required form. For instance, predicates like

$$\text{let } x_e = e' \text{ in}$$
$$z = g(a, ..., x, x_e)$$

which are not in the required form for the translation, are transformed to equivalents in the form

$$z = \text{let } x_e = e' \text{ in}$$
$$\quad g(a, ..., x, x_e)$$

which can be translated. Another example are predicates in the form

$$p(a, ..., x) \;\Rightarrow\; z = g_1(a, ..., x) \wedge$$
$$\neg p(a, ..., x) \;\Rightarrow\; z = g_2(a, ..., x)$$

which are first transformed into

$$z = \text{if } p(a, ..., x)$$
$$\quad \text{then } g_1(a, ..., x)$$
$$\quad \text{else } g_2(a, ..., x).$$

and then translated into LML. This is not automatically done by the current implementation, but it could be extended to deal with these simple rules.

4.3.2 Operations

A VDM operation is modeled by a function that receives the operation's arguments and the state before it is performed, and gives as result the operation's result and the state after its use. A function that implements an operation checks if the pre-condition of the operation is satisfied and when this does not happen an error result is returned (the same alternatives for error handling, as described in Section 4.3.1 for functions, are valid for operations as well). An operation of the syntactic form

$$OP \;\; (p{:}\, Tp)\; r{:}\, Tr$$
$$\text{ext rd } v \;:\; Tv$$
$$\quad \text{wr } t \;:\; Tt$$

$\mathsf{pre}\ p \ldots v \ldots t$

$\mathsf{post}\ t = p \ldots \overleftarrow{v} \ldots \overleftarrow{t} \ldots r$

$\quad \wedge\, r = p \ldots \overleftarrow{v} \ldots \overleftarrow{t} \ldots t$

with parameter p and result r, having write access to state component t and read only access to component v, where hooked variables represent state components before the performance of an operation, and non-hooked ones represent components after it, is translated to the function

```
OP (p,v,t) =
   if pre-OP(p,v,t)
   then (t´,r´)
   else (err-t,err-r)
```

where `t´` and `r´` are the expressions resulting from the translation of the expressions associated with t e r in the operation's post-condition; `err-t` and `err-r` are error values belonging to the types `Tt` and `Tr`. The pre-condition is represented by `pre-OP(p,v,t)`. As discussed in the last subsection, the form of the post-condition is important, because some predicates cannot be translated directly. For instance, it is not possible to translate the post-condition $\overleftarrow{t} = [r]^\frown t$, with $^\frown$ standing for sequence concatenation, but the translation of its equivalent $r = \mathsf{hd}\ \overleftarrow{t} \wedge t = \mathsf{tl}\ \overleftarrow{t}$ is trivial.

As it was pointed out before, only operations written in a explicit style (identified by the syntactic form of the operation above) —expressing the result and the state *after* as functions of the operation arguments and the state *before* the operation— can be automatically translated. All other ones should be superficially refined by finding a simple algorithm that implements it.

5 An Example: Translation of Type Declarations

As a simple example, we will show how the (partial) specification of an operation that translates VDM type declarations into LML type declarations is translated into LML. This is part of a translation system from VDM to LML, developed using the approach being described.

For simplicity, we consider that type declarations create new types in terms of basic or (already) user-defined types. Usually, in functional languages a type name can only be used after its definition, while in VDM the order is irrelevant. So, to translate VDM type declarations we need to put them in order. For example, a declaration set like

$C = \mathrm{BOOL}$

$B = D$

$E = C$

$D = C$

should be translated to

```
#define C Bool
#define D C
#define B D
#define E C
```

or

```
#define C Bool
#define E C
#define D C
#define B D
```

where in both cases a type name is only used after its definition.

A type declaration is composed by a type name and a type expression, and is modeled by

$$\begin{aligned} \mathit{TypeDec} :: \mathit{tpname} &: \mathit{TypeName} \\ \mathit{type} &: \mathit{Type} \end{aligned}$$

where

$\mathit{TypeName}$, is not yet defined.

Type expressions are either basic (NAT, REAL and BOOL) or declared types (type names).

$$\mathit{Type} = \mathit{BasicType} \mid \mathit{DecType}$$

$$\mathit{BasicType} = \{\mathrm{NAT}, \mathrm{REAL}, \mathrm{BOOL}\}$$

$$\mathit{DecType} :: \mathit{tpname} : \mathit{TypeName}$$

Following the translation schemes presented in section 4, we can translate the declarations above to:

```
#define Typename (List Char)
rec type TypeDec = TpDec (TypeName # Type)
and      Type = BasTp BasicType + DecTp  DecType
and      BasicType = BasTpNAT NAT + BasTpREAL REAL + BasTpBOOL BOOL
and      NAT = NATcon_
and      REAL = REALcon_
and      BOOL = BOOLcon_
and      DecType = DTp TypeName
```

where a list of characters is used as the implementation of *TypeName*. The associated operators can be defined as described in section 4.

VDM type declarations are represented by

$TypeDecs = \textsf{set of } TypeDec$

which to be semantically valid (*W*ell-*F*ormed) should satisfy the function

$$\begin{array}{l} WF\_TypeDecs(s) \quad \triangleq \\ \quad (\forall d, e \in s \cdot d \neq e \;\Rightarrow\; tpname(d) \neq tpname(e)) \\ \quad \wedge\ noCycles(s) \\ \quad \wedge\ \forall e \in s \cdot \\ \qquad \textsf{is-}DecType(type(e)) \;\Rightarrow \\ \qquad \exists d \in s \cdot tpname(type(e)) = tpname(d) \end{array}$$

which is true when the following properties are valid:

- there are no two type declarations defining the same type, as in

  $A = B$

  $A = \textrm{N}\small{\textrm{AT}};$

- there are no cyclic declarations, that is, types defined in terms of themselves. This is established by the function *noCycles* whose definition we omit here for brevity. In this way, declarations like

  $B = A$

  $A = B$

  are not allowed;

- all used type names are defined. So, we cannot have declarations like

$$A = B$$

$$C = \textsc{Real}$$

since B is not declared.

As the order of declaration is important in LML, we model LML type declarations by

$$LMLTypeDecs = \mathsf{seq\ of}\ TypeDec.$$

A sequence of declarations is well-formed if the set formed by its elements does not have cycles, and all named types are defined before, in the order of the elements of the sequence, the first point where they are used. This restriction is formalised by the following function

$$
\begin{array}{l}
WF\_LMLTypeDecs(s) \quad \triangleq \\
\quad noCycles(\mathsf{elems}\, s) \wedge \\
\quad \forall i \in \{1, \ldots, \mathsf{len}\, s\} \cdot \\
\quad\quad \mathsf{is}\text{-}DecType(type(s(i))) \;\Rightarrow \\
\quad\quad \exists j \in \{1, \ldots, i-1\} \cdot tpname(type(s(i))) = tpname(s(j)).
\end{array}
$$

The function that realises the type translation from VDM to LML is specified by

$$
\begin{array}{l}
Vdm\_Lml \;\; (td\colon TypeDecs)\ std\colon LMLTypeDecs \\
\mathsf{pre}\ \ WF\_TypeDecs(td) \\
\mathsf{post}\ \mathsf{elems}\, std = td \\
\quad \wedge\ WF\_LMLTypeDecs(std) \\
\quad \wedge\, \forall i, j \in \{1, \ldots, \mathsf{len}\, std\} \cdot \\
\quad\quad\quad i \neq j \;\Rightarrow\; tpname(std(i)) \neq tpname(std(j))
\end{array}
$$

that receives well-formed VDM type declarations and gives associated well-formed LML type declarations. The function is very abstract and does not give an algorithm to compute the result; it only states that the elements of the sequence given as result are the same of the argument, and that the result does not have duplicated elements.

In this case the translation cannot be automatic since the operation is not directly executable. Then, to derive a prototype we need to find a simple algorithm that implements the above specification. A simple and executable, but inefficient, implementation is given by the following explicit function definition

$Vdm\_Lml1 : TypeDecs \rightarrow LMLTypeDecs$

$Vdm\_Lml1(td) \quad \triangleq$

$\quad$ if $WF\_TypeDecs(td)$

$\quad$ then if $td \neq \{\,\}$

$\quad\quad$ then let $s = \{e \mid e \in td \bullet$

$\quad\quad\quad\quad \forall f \in td \cdot \text{is-}DecType(type(f)) \;\Rightarrow$

$\quad\quad\quad\quad tpname(type(f)) \neq tpname(e)\}$ in

$\quad\quad\quad$ let $e \in s$ in

$\quad\quad\quad$ $Vdm\_Lml1(td - \{e\}) \frown [e]$

$\quad\quad$ else $[\,]$

$\quad$ else $[\,]$

The algorithm chooses, from the set received as argument, declarations that are not used by any others in the set and puts those in the end of the sequence given as result. Such elements always exist in a declaration set, otherwise there would be a cycle. The set without such elements is still a valid declaration set. The process is recursively repeated and an empty declaration set corresponds to an empty declaration sequence.

To prove the correctness of the refinement process, we should discharge the following proof obligation:

$$td \in TypeDecs \vdash WF\_TypeDecs(td) \Rightarrow \text{post-}Vdm\_Lml(td,\, Vdm\_Lml1(td))$$

The predicate post-$Vdm\_Lml(td,\, Vdm\_Lml1(td))$ is an abbreviation for the post-condition of Vdm_Lml, where std is replaced by $Vdm\_Lml1(td)$. The proof obligation establishes that Vdm_Lml1 is an implementation of Vdm_Lml and can be easily discharged. The proof is simple, although tedious and it is omitted here.

Now we can proceed to get the LML prototype. Types, expressions and functions are translated as described in section 4. As described in 4.2, consider the following: set difference $(-)$ is represented by `setdif`, sequence concatenation is `concat`, the choice expression let $e \in S$ in becomes `let e = choice S in`, set comprehension in the form $\{e \mid e \in S \bullet p(e)\}$ is translated to `setcomp S p`, the set $\{i, \ldots, j\}$ is represented by `i setint j`, the empty set $(\{\})$ is `emptyset eq`, where `eq` is the equality operator for set elements, $\{e\}$ becomes `singleton e eq`, the empty sequence $([\,])$ is `emtptyseq eq`, and $[e, f, g]$ is translated to `makeseq [e,f,g] eq`. Furthermore,

$A \Rightarrow B$ is implemented by `if A then B else true`. The corresponding LML function is defined as

```
Vdm_Lml1 td =
 if (WF_TypeDecs td) then
   if not(td TypeDecs_eq (emptyset TypeDecs_eq)) then
     let rec
       g = (\e. forall td (\f.
             if Type_is_ "DecType" (type f) then
                tpname´(Type_proj_DecType(type f)) <>
                tpname(e)
             else true))
     and s = setcomp td g
     and e = choice s
     in Vdm_Lml1(td difset (singleton e TypeDecs_eq))
        concat (makeseq [e] LMLTypeDecs_eq)
   else emptyseq LMLTypeDecs_eq
 else emptyseq LMLTypeDecs_eq
```

where lambda abstraction has the syntax `\e.exp`, and `TypeDecs_eq` and `LMLTypeDecs_eq` are the equality operators for *TypeDecs* and *LMLTypeDecs*. `tpname` and `tpname´` are, respectively, the selectors for *TypeDec* and *DecType*.

This is a sketch of how the prototypes are effectively computed from their VDM specifications. Naturally, the refinement process could be continued up to the point where an efficient implementation is achieved. However, depending on the tests to be done with the prototype, the implementation above may be acceptable.

6 Conclusions

The system described here is a prototyping tool of limited use; its shortcomings made it simple to understand and design, although its implementation is a sophisticated piece of code: more than 15k lines of LML, part of it bootstrapped using the system itself.

It is not a match for algorithm design systems as KIDS[Smi90] but it was not built for that purpose. Generating functional prototypes from VDM specifications using our approach is a quite efficient means of testing model-based specifications and can be further developed using functional calculi such as [Mer87] in order to generate efficient programs.

Related work on the development of prototypes from model based specifications is described in [JS90, DKC90, SL87, WE92]. However, what makes our approach unique among those is the complete formalization of the translation process, given by a specification in VDM of a system for performing the automatic translation from VDM to LML. The cited works either describe the translation of specific case studies or have only an informal description of the translation rules. Also, our approach was extensively tested and validated in the design and development of the prototype system for performing the translation from VDM to LML, partly described in this article, not only by simple case studies.

A laboratory system such as the one we developed is certainly not as real as the software designed and implemented in industry; but it was a large and complex project indeed. There seems to be no reason to stop the same approach being used in rigorous prototype development of industrial software systems.

The work in [DKC90] shows a different approach for prototyping, using Prolog's trial-and-error mechanism, requiring the use of a program transformation system in order to obtain useful prototypes. This does not happen with our approach since operations are directly translated into functions, rather than relations. Another problem with the use of Prolog as a prototyping language is the use of non-axiomatic Prolog control features, what makes the proof of correctness of the translation process much more difficult, as reported in [WE92]. On the other hand, we use a purely functional prototyping language.

A potential source of problems in the development is the presence of state in VDM and its absence in the underlying functional model. But this can be modeled by continuations [SRI91, Per90], in a way that the prototype will be a real representation of the specification.

In order to use the described approach for real safety-critical systems, the translation process should be completely formalized and proved correct. In this way, we could assure that a generated prototype is a model of the theory defined by a VDM specification.

Proving the correctness of the critical parts of the translation process and using the (partially complete) system to bootstrap itself gave us confidence in the approach: functional prototypes of model-based specifications can be useful for testing specifications, and in cases where efficiency is not a problem, to be used as systems themselves.

7 Acknowledgements

The authors wish to thank the anonymous referees for the comments and suggestions which have improved this paper.

References

[AJ88] L. Augustsson and T. Johnsson. *Lazy ML User's Manual.* Department of Computer Science, Chalmers University of Technology, Göteborg, draft edition, January 1988.

[BM89] P. Borba and S. Meira. VDM executable notation in Standard ML (in portuguese). In *Proceedings of the IX Congress of the Brazilian Computing Society*, Uberlândia - MG, July 1989.

[BMPP89] F. Bauer, B. Möller, H. Partsch, and P. Pepper. Formal program construction by transformations — computer-aided, intuition-guided programming. *IEEE Transactions on Software Engineering*, 15(2), February 1989.

[Bor91] P. Borba. *From Formal Specifications to Functional Prototypes (In Portuguese).* MSc thesis, Universidade Federal de Pernambuco, Departamento de Informática, Brazil, July 1991.

[BSI89] BSI. VDM Specification Language—Proto-Standard. Draft, British Standards Institution IST/5/50, June, 1989.

[DKC90] A. Dick, P. Krause and J. Cozens. Computer Aided Transformation of Z into Prolog. In John E. Nicholls, *Z User Workshop*, Oxford 1989, Springer-Verlag, Workshops in Computing, pp. 71-85, 1990.

[HJ89] Ian Hayes and Cliff Jones. Specifications are not (necessarily) executable. University of Manchester, Computer Science Department, UMCS-89-12-1. December, 1989.

[HMM86] R. Harper, D. MacQueen and R. Milner. Standard ML. Edinburgh University, ECS-LFCS-86-2, 1986.

[Jon90] C. B. Jones. *Systematic Software Development Using VDM.* 2nd. ed., Prentice Hall, 1990.

[JS90] M. Johnson and P. Sanders. From Z Specifications to Functional Implementations. In John E. Nicholls, *Z User Workshop*, Oxford 1989, Springer-Verlag, Workshops in Computing, pp. 86-112, 1990.

[Mer87] L. Merteens. Algorithmics – towards programming as a mathematical activity. *Mathematics and Computer Science*. CWI Monographs. North-Holland, 1987.

[Moo90] R. Moore. Mural: A Formal Development Support Environment. In *SafetyNet'90*, December 1990.

[Mor90] Carroll Morgan. *Programming from Specifications*. Prentice Hall, 1990.

[Per90] N. Perry. The Implementation of Practical Functional Programming Languages. PhD Thesis, Imperial College of Science, Technology & Medicine, University of London, 1990.

[SL87] S. Stepney and S. Lord. Formal Specification of an Access Control System, *Software – Practice and Experience*, 17(9), September 1987.

[Smi90] D. R. Smith. KIDS: A Semiautomatic Program Development System. *IEEE Transactions on Software Engineering*, 16(9), September 1990.

[SRI91] V. Swarup, U. S. Reddy and E. Ireland. Assignments for Applicative Languages. Functional Programming Languages & Computer Architecture, 5th ACM Conference, Cambridge, Ma, USA, 1991, LNCS 523.

[Tur84] D. Turner. Functional programs as executable specifications. In *Phil. Trans. Royal Soc. of London (312), 1522*. ACM, 1984.

[Tur85] D. Turner. Miranda: A non strict functional language with polymorphic types. *Proceedings of the 1985 Conference on Functional Programming Languages and Computer Architecture*, 1985. LNCS 201.

[WE92] M. West and B. Eaglestone. Software development: two approaches to animation of Z specifications using Prolog, *Software Engineering Journal*, 7(4), July 1992.