A System for Translating Executable VDM Specifications into Lazy ML

Paulo Borba[1] Silvio Meira[2]

Departamento de Informática

Universidade Federal de Pernambuco

P.O. Box 7851, 50732-970, Recife, PE, Brazil

E-mail: {phmb,srlm}@di.ufpe.br

[1]Research funded by CNPq grant 301021/95-3.

[2]Research funded by CNPq grant 300133/85.

Abstract

This article describes the process of developing a system for translating VDM executable specifications into Lazy ML prototypes. The system was specified in VDM and a Lazy ML prototype implementation was derived from its specification. This article concentrates on discussing the lessons learned in each stage of the development process, evaluating the adequacy of the used methodology.

Keywords: Formal Methods, Rapid Prototyping, Functional Languages, Specification Languages, Compilers.

Introduction

This article describes the development process of a system for translating VDM [1] executable specifications into Lazy ML (LML) prototypes [2, 3]. This system provides machine support for software development methodologies based on VDM [4].

Prototypes may be used as animation tools for validating software system specifications in the very early stages of the development process. This validation is essential for formal development of software in practice. As observed in Reference 4, those prototypes can usually be directly translated from specifications, after a superficial refinement process. Nevertheless, this translation involves a lot of mechanical and tedious tasks such as type inference and checking, overloading resolving, etc. Indeed, those tasks are best realized by software, mainly when considering industrial strength specifications. The system described in this article automates the translation, speeding up the derivation of prototypes from specifications, and making the derivation more reliable.

The pair formed by model based specification languages and functional programming languages offer a good deal to programmers whose interest is to have a correct prototype, built from a high-level, logical specification of the piece of software they are constructing [4].

At first, it might seem that functional languages are not appropriate for prototyping state-based specifications; however, states can be quite easily modelled by arguments and results of functions. Also, high order, strongly typed, lazy, purely functional languages are appealing for formal derivation of prototypes from specifications because they have a simple formal semantics and they motivate a declarative programming style. This set of properties leads to the ease of reasoning about the translation process, to referential transparency, polymorphism and pattern matching, which reduce the semantic gap between the specification and the prototyping language; so that it is easier and quicker to derive prototypes.

In fact, the translation system (translator) described here was developed using the methodology for prototype derivation introduced in Reference 4, and supported by the system itself; that is, the system was specified in VDM and a LML prototype implementation was derived from this specification. Part of this prototype was bootstrapped.

VDM and LML were not chosen because they are ideal for a project such as this. They were chosen because the two languages were well established and used worldwide. Instead of VDM, other languages could have been used: Z [5], which provides some modularization mechanisms, or the object-oriented extensions of VDM [6] and Z [7] which offer support for data encapsulation and modularity *in the large*. Instead of LML, Haskell [8] could have been used; that would probably facilitate the translation process since Haskell supports extra features, such as overloading, not supported by LML.

This article describes the whole development path of this system, including the lessons learned in each stage and evaluating the adequacy of the used methodology. In order to properly understand the article, the reader should have a working knowledge of LML or functional programming, and a basic knowledge of VDM or model-based formal specifications in general. Nevertheless, concepts from both languages are explained when they depart from common knowledge.

Requirements

The application described in this article translates an executable subset of the VDM language into LML. Essentially, the requirements for this application were introduced in Reference 4, by giving translation schemes for mapping VDM constructions into the associated LML code. The application automatically applies those translation schemes to a specification, in order to obtain a corresponding prototype. This sections gives a general overview of those schemes.

The types of the specification language are implemented by types in the functional language; these are used as the basis for the construction of prototypes. For instance, natural numbers and other basic types of VDM are translated into the corresponding built-in LML types, whereas composite objects such as

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

similar to records in Pascal, are represented in LML 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. Also, VDM union types—types constructed by the union of other types—are directly translated into LML algebraic data types, with constructors (tags) for each component of the union type.

Type translation is simplified by implementing the usual mathematical objects from model based specification languages as abstract data types (ADTs) in the functional language. In this way, VDM type constructors for sets, sequences and maps are represented by equivalent ones implemented in LML. Thus type translation is direct; there is no need to perform data type refinement.

The translation of expressions is basically specified by indicating how the operations over data types and logical operators are translated.

Operations over sets, sequences and maps, such as set union and map domain, are translated into the associated functional operations implemented in the corresponding ADTs. The same is true for the operations associated to other type constructors. For instance, the operators available for VDM composite types can be automatically generated in LML: selectors are implemented by

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

and VDM's *make* functions are replaced by type constructors in such a way that expressions like

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

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

The logical operators of VDM—and, not, implies, etc.—are easily translated into LML pre-defined boolean operators. The logical quantifiers such as "for all" and "exists" are translated into predicates in the functional language, if the associated base set is finite. Otherwise, some sort of refinement is necessary before the translation.

Only VDM functions and operations defined in an explicit style can be automatically translated. Whereas pre-conditions can always be automatically translated, post-conditions may require some refinement, which must be carried out by the user, in order to reduce the level of abstraction and to allow automatic coding. However, the necessary refinement may sometimes be easily carried out, since the interest is on obtaining a prototype, rather than an efficient implementation. In fact, in a number of cases, only very simple refinement or transformation rules are needed to transform the post-condition to the required explicit style.

In order to automatically translate a function, its post-condition must establish, explicitly, the function result in terms of its arguments. If this is the case, the VDM function is translated to an equivalent LML function. More precisely, only functions specified in the syntactic form below

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

pre $a \ldots x$

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

can be automatically translated; they are implemented by definitions of the form

```
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 pre-condition of the VDM operation. If this condition is not satisfied, an error value is returned, represented here as `error-value`; this value should be an existing distinguished value of type `Z`. The system does not create error values automatically.

Similarly to functions, only VDM operations having the following specific form are translated: the result and the state *after* are expressed as functions of the operation arguments and the state *before* the operation is executed. Such an operation is translated to an LML function taking the operation's arguments and the state before it is performed, and yielding the operation's result and the state after its use. Furthermore, a function that implements an operation checks if the pre-condition of the operation is satisfied; when this does not happen, an error result is returned.

In order to illustrate the translation of operations, a VDM operation in the syntactic form

$OP \quad (p \colon Tp)\ r \colon Tr$

ext rd $v \ : \ Tv$

  wr $t \ : \ Tt$

pre $p \ldots v \ldots t$

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

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

with parameter p and result r, having write access to state component t and read only access to component v, where hooked variables in the post-conditions represent state components before the execution of an operation, and non-hooked ones represent components after it, is translated to the LML 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 and r in the operation's post-condition; `err-t` and `err-r` are error values belonging to the types `Tt` and `Tr`. The translated pre-condition is represented by `pre-OP(p,v,t)`.

As discussed above, the explicit form of the post-condition is important, because some predicates cannot be directly translated. In order to be automatically translated, an operation must be in the exact syntactic form shown above. 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

$$r = \mathbf{hd}\,\overleftarrow{t} \wedge t = \mathbf{tl}\,\overleftarrow{t}$$

is trivial.

An extra requirement is that the translation system should accept specifications in VDM concrete syntax [9], and yield prototypes in LML syntax.

Many other features could be supported by such a system. In particular, it could apply simple VDM transformation rules to some operations defined in an implicit style, in order to automatically obtain a corresponding operation in an explicit style. Sometimes, this systematic application is possible. However, it is not considered by the translator presented here.

High Level Design

The high level design of the translation system described here follows some well known recipes from compiler construction techniques [10], which seem adequate in this context. In this way, the system can be understood in terms of the following main functions:

- Parser (P);
- Type translation (TT);
- Translation of expressions (ToE);
- Translation of functions and operations (ToFO);
- Pretty-printer (PP).

Except for the first and last, which are usually part of any translation system, the other functions correspond to the main syntactic constructs of VDM and provide the reason for structuring the system in this way.

Based on the discussion in the last section, the input to P, the parser, should be a specification in VDM concrete syntax and the output from PP, the pretty-printer, should be the corresponding LML code. Also, the parser generates the internal representation of a VDM specification, whereas the pretty-printer accepts the internal representation of an LML program.

Functions TT, ToE and ToFO take the internal representation of VDM specifications and yield the corresponding internal representation of LML code. They should be implemented according to the translation schemes discussed in the last section. In order to do that, the system has to perform some specific tasks.

First, the function for type translation has to generate a mapping (table) associating each VDM type to its corresponding translated type and operations in LML. Among other functions, this information is used by the ToE function, since it has to know what are the translated functions related to a given data type.

VDM adopts structural equivalence of types; that is, two type expressions are equivalent if they are the same except for renaming. For example, the types X and Y defined as

$X = \textbf{set of N}$

$Y = \textbf{set of } Z$

$Z = \mathbf{N}$

are structurally equivalent, since Z is just an abbreviation for $\mathbf{N}$. So, in order to decide if two type expressions are equivalent, it is necessary to have some information relating VDM type names such as X to associated type expressions such as **set of N**. This information is also stored in a mapping. It can be directly generated from the specification and it is necessary for performing many tasks, including the type checking of VDM expressions.

For simulating the semantics of VDM's union types in LML, sometimes it is necessary to use conversion functions from the translation of a union type to one of its subtypes or vice-versa, where a subtype is any type corresponding to the translation of one of the components of the union. These functions might be necessary if an expression of a subtype is used where an expression of the union type is expected, or vice-versa. These conversion functions are the constructors of the LML algebraic type corresponding to the union type, and their associated projections. For automatically inserting those functions into the translated code, the system must keep some information about the type hierarchy associated to a specification. Basically, the system keeps a mapping from each VDM type to its related types—subtypes and supertypes—together with the corresponding conversion functions in LML.

VDM specifications may introduce overloaded identifiers; that is, the same identifier may be used to name operations (functions) of different types. As LML does not support overloading, the LML functions corresponding to overloaded VDM operations should have different names. So, in order to translate expressions containing overloaded operations, the system needs a mapping indicating the LML identifier associated to each function resulting from the translation of a given VDM operation, which is uniquely identified by its name and type.

Lastly, because of overloading, the system needs to know the type of a VDM expression before it can be translated; only in this way, the system can uniquely identify the operations being used in this expression. So, the system should have functions for inferring and checking the type of VDM expressions; this functionality basically requires the implementation of a full polymorphic type inference system, such as the ones available in the implementation of functional languages.

Considering the tasks described above, the following is a more accurate description of the main functions of the translation system:

- Parser (P);
- Type translation, generating the "translated types mapping" (TT);
- Generation of the "structural equivalence mapping" (gSEM);
- Generation of the "union types mapping" (gUTM);
- Generation of the "overloading mapping" (gOM);
- Type inference and checking (TIC);
- Translation of expressions (ToE);
- Translation of functions and operations (ToFO);
- Pretty-printer (PP).

Figure 1 shows the system architecture, indicating the interconnection of the functions above and the corresponding data flow. For clarity, dotted lines are also used to indicate data flow. The internal representation of VDM specifications, hereafter VDM tree, is used by most functions. Based on this representation, gSEM generates a mapping for gUTM, gOM, TIC and ToE. In a similar way, TT yields a mapping, which is used by gUTM, gOM and ToE, and some LML code which is later pretty-printed. This code corresponds to some translated types and it is in the internal representation of LML code, hereafter LML tree. ToE uses the mappings generated by TT, gSEM, gUTM and gOM and the type information given by type checker to yield translated expressions in the internal representation of LML programs, which are then used by ToFO to generate translated operations in the same format.

Low Level Design

As usual, the parser is essentially specified by a formal grammar; for the translation system, that is given by the British Standards Institution (BSI) definition of VDM's concrete syntax [9]. Furthermore, specifications are internally represented as trees, in a similar way that expressions are represented.

Figure 1: System Architecture.

LML programs are also internally represented as trees. The pretty-printer just translates this internal representation into LML syntax. As this task is quite obvious, no further specification is necessary. Basically, LML's syntax [2, 3] specifies the pretty-printer.

Apart from the parser and pretty-printer, all functions have been formally specified in VDM. However, instead of following the structure of the functional design, splitting the VDM specification in modules corresponding to the main functions of the system, the specification was structured in an object-oriented style [11].

Basically, there is an ADT for each syntactic construct of VDM and LML, denoting the trees generated by the syntactic constructs. In general, those data types have functions or operations for creating trees, accessing the elements of trees, building some of the mappings necessary for the translation and type checking, translating and pretty-printing trees. Some of these operations are exclusive to trees of a particular language. For example, translation and type checking are only available to trees representing VDM constructs.

From what was said above, observe that the main functions of the system are modelled by VDM functions or operations related to those trees. Furthermore, the functions take some of the mappings as arguments, whereas some mappings are part of the state of the main operations. Each mapping also defines an ADT, with functions for accessing and modifying the information that they keep.

The following list gives a general idea of how the VDM specification of the translation system has been structured, by indicating the modules forming the specification:

- VDM concrete syntax grammar [9].
- ADTs representing LML syntactic structures such as type definitions, function definitions, expressions, etc. Those ADTs include the following:
  - trees for each LML syntactic structure;
  - operations for creating and accessing components of those trees;
  - pretty-printers and
  - ADTs for sequences and sets of trees.
- ADTs representing VDM syntactic structures, which consist of
  - trees for each structure [9];
  - operations for manipulating those trees;
  - functions for building the mappings necessary for translation;
  - type inference and checking;

Table 1: Specification modules sizes.

- type translation operations;
- expression translation functions and
- function and operation translation functions.

- ADTs for the various mappings necessary for translation, basically containing functions for creating, accessing and modifying the information associated to those mappings.

- LML syntax [2, 3].

Table 1 shows the approximate size, in lines of text including blank lines and comments, of the specification modules enumerated above.

As VDM does not directly support object-orientation, the modularization technique first described in Reference 12 was used to yield a more reusable and maintainable specification. This approach for structuring the specification proved to be quite satisfactory, simplifying the process of understanding, reusing and maintaining the specification. For example, it is really easy to modify the specification in order to translate a given VDM construct in a different way, or to add a new construct to the language.

Contrasting with this approach, if the specification were structured in the same way as the high level design, it would be necessary to modify many modules in order to introduce a new construct in the language. Also, the size of the modules defining the main functions would be much bigger than the size of the modules defining simple ADTs. Naturally, the smaller modules are easier to understand and to modify.

The system was first specifyed in VDM. Contrasting, it could have been directly specified in the functional language. This approach would be possible, given the abstractness intrinsic to well-written, properly typed, lazy, higher-order programs. However, that does away with one of the primordial characteristics of abstract, not necessarily executable, specifications [13], where one does not have to deal with the intricacies of the concrete objects and algorithms needed to make a piece of "code" executable by a computer.

Indeed, the approach adopted for specifying and structuring the system described here gives a better understanding of the system before any executable code is actually written. This understanding helps programmers to write more reliable and efficient code. Furthermore, programmers can reason about the system at a higher level of abstraction, aiding the identification of bugs and design flaws.

Presenting part of the VDM specification of the translator is useful to illustrate some of the aspects discussed in this section. In particular, a simplified version of the ADT of VDM type declarations, including the function that translates VDM type declarations into LML type declarations, is a good example.

For simplicity, 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, whereas in VDM the order of the definitions is irrelevant. So, in order to translate VDM type declarations the system has to put them in order. For example, a declaration group 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.
```

Note that 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

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

where

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

Type expressions are either basic, like NAT, REAL and BOOL, or declared type names:

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

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

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

VDM type declarations can then be represented by

$$\mathit{TypeDecs} = \textbf{set of } \mathit{TypeDec}$$

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

$$\begin{aligned}
&\mathit{WF_TypeDecs}(s) \quad \triangleq \\
&\quad (\forall d, e \in s \cdot d \neq e \;\Rightarrow\; \mathit{tpname}(d) \neq \mathit{tpname}(e)) \\
&\quad \wedge\ \mathit{noCycles}(s) \\
&\quad \wedge\ \forall e \in s \cdot \\
&\qquad \textbf{is-}\mathit{DecType}(\mathit{type}(e)) \;\Rightarrow \\
&\qquad \exists d \in s \cdot \mathit{tpname}(\mathit{type}(e)) = \mathit{tpname}(d)
\end{aligned}$$

which is true when the following properties are valid:

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

  $A = B$

  $A = \text{NAT}.$

- There are no cyclic declarations, that is, types defined in terms of themselves. This constraint is established by the function *noCycles* whose definition was omitted for brevity. In this way, declarations like

  $B = A$

  $A = B$

  are not allowed.

- All used type names are defined. So declarations like

  $A = B$

  $C = \text{REAL}$

are not allowed since B is not declared.

It remains now to present the type translation function. However, as that function yields LML type declarations, its specification depends on the specification of the ADT of LML type declarations, which we partly describe first. As the order of declaration is important in LML, type declarations are modelled by

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

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 formalized by the following function

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

Finally, the function that performs the type translation from VDM to LML:

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

It receives well-formed VDM type declarations and gives associated well-formed LML type declarations. Note that the specification of 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. For that end, the specification uses several non-executable features of VDM.

Implementation

The implementation of the translation system is based on its specification. In fact, the implementation is structured in the same way as the specification.

One of the standard ways to implement a parser is to use a parser generator [10], which automatically generates a parser from a grammar specifying the syntax of the language. Initially, the "Functional Parser Generator" (FPG) [14] was used. FPG was the only available parser generator written in LML, and consequently the only one that easily interacts with LML programs. Unfortunately, after some experiments, it was clear that this prototype tool is not robust enough to be used for generating parsers for a language having such a complex syntax as VDM.

Also, it seemed impractical and inefficient to develop a parser in a functional language from scratch, even using well known techniques such as those given in Reference [15].

In view of these difficulties, a compromise solution was adopted, following the approach used to implement the parser of the LML compiler. In summary, an imperative parser generator [16] translates VDM concrete syntax into a prefixed version of it, which can be straightforwardly parsed by a simple LML program. The second parser is necessary because the first cannot easily interact with the translation system, since they are written in different languages. So, the output of the first parser is used by the second for generating a tree in LML. This tree can then be manipulated by the translation system.

Alternatively, an imperative parser generator could be used to generate a parsing table for VDM, which could be converted to a format suitable to be processed by a parsing algorithm implemented in LML. This approach would be similar to FPG's approach, except that the parsing table would be generated by an imperative program, rather than a functional program. It seems that the adopted approach is simpler and, based on the experience with FPG, more efficient than this one.

The adopted solution required a small redesign of the main functions of the system, given that two parsers were necessary. Fortunately, the only modification was to add a new function following P, the original parser, corresponding to the second parser. The specification and implementation of this extra function are trivial; it basically builds trees in LML, starting from a prefixed direct representation of those trees. The first parser was directly generated from the concrete VDM syntax grammar.

The implementation of the pretty-printer is straightforward. It was implemented in a standard way, by using pattern matching to map LML trees, representing LML programs, to strings with the syntax of their associated LML programs.

The other three components of the system specification correspond to the VDM specification of the ADTs of LML programs, VDM specifications, and the mappings with information necessary for the translation. The implementation of those components reflects the object-oriented structure of their specification. As LML does not have any support for object-orientation, this structure was simulated by LML modules defining ADTs, in a similar way as done for the VDM specification.

Those components were implemented in an unusual way. The implementation was derived, partly automatically, from the specification using the method suggested in References 4 and 17. In this method, concrete specifications are directly implemented in LML, according to translation schemes. This task can be automatically performed by a system such as the one being described here. More abstract specifications should be refined into specifications written in an explicit style. This task cannot be automatically performed by software. In order to refine specifications, programmers can proceed in a completely *ad hoc* way, finding a more concrete specification, explicitly defined, and proving it correct with respect to the more abstract, implicitly defined. Alternatively, programmers can use a consistent set of semantic preserving rules to aid the process, deriving a correct implementation from the specification. This derivation can be done by means of refinement or transformation rules, as described in References 18 and 19.

The derivation of the prototype implementation from the specification was straightforward. Some refinement was necessary in order to obtain an explicit specification, but that, however, was simple and superficial, since the objective was not to obtain an efficient implementation. For example, no data refinement was performed in order to obtain the explicit specification; that was not necessary since the data structures of VDM are implemented in LML, as discussed before. In fact, the explicit specification was obtained by chosing very simple algorithms that satisfy the abstract specification. Those aspects justify the simplicity of the refinement process, despite the fact that the abstract specification uses several non-executable features of VDM.

For instance, note that the specification of the function that performs the type translation from VDM to LML could not be directly implemented, since it uses some of VDM's non-executable features. So a simple and executable, but inefficient, algorithm that implements the specification has to be presented:

$$Vdm\_Lml1 : TypeDecs \to LMLTypeDecs$$

$$
\begin{array}{l}
Vdm\_Lml1(td) \quad \triangleq \\
\quad \textbf{if } WF\_TypeDecs(td) \\
\quad \textbf{then if } td \neq \{\,\} \\
\qquad \textbf{then let } s = \{e \mid e \in td \bullet \\
\qquad\qquad\qquad \forall f \in td \cdot \textbf{is-}DecType(type(f)) \;\Rightarrow \\
\qquad\qquad\qquad tpname(type(f)) \neq tpname(e)\} \;\textbf{ in} \\
\qquad\quad \textbf{let } e \in s \;\textbf{ in} \\
\qquad\quad Vdm\_Lml1(td - \{e\}) \frown [e] \\
\qquad \textbf{else } [\,] \\
\quad \textbf{else } [\,]
\end{array}
$$

The algorithm encoded by this explicit VDM function 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.

In order to prove the correctness of this refinement step, the following proof obligation should be discharged:

$$td \in \mathit{TypeDecs} \vdash \mathit{WF\_TypeDecs}(td) \Rightarrow \textsf{post-}\mathit{Vdm\_Lml}(td,\ \mathit{Vdm\_Lml}1(td))$$

The predicate $\textsf{post-}\mathit{Vdm\_Lml}(td,\ \mathit{Vdm\_Lml}1(td))$ is an abbreviation for the post-condition of *Vdm_Lml*, where *std* is replaced by $\mathit{Vdm\_Lml}1(td)$. The proof obligation establishes that *Vdm_Lml*1 is an implementation of *Vdm_Lml* and can be easily discharged. The proof is simple, although tedious and it is omitted here. In the development of the translator, only a few proof obligations were formally discharged.

Now the LML prototype can be derived. Using the translation rules given in Reference 4, the VDM type definitions presented in the previous section were translated to the following LML code:

```
#define Typename (List Char)
#define TypeDecs (Set TypeDec)
#define LMLTypeDecs (Seq TypeDec)
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 implementation of the type translation function uses the ADTs of sets and sequences implemented in LML. So consider the following: set difference, $-$, is represented by `setdif`, sequence concatenation is `concat`, the choice expression $\textsf{let}\ e \in S\ \textsf{in}$ 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`, and $[e, f, g]$ is translated to `makeseq [e,f,g] eq`. In this way, the explicit specification of the type translation function is translated to a corresponding LML function 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*. Also, `tpname` and `tpname'` are, respectively, the selectors for *TypeDec* and *DecType*.

In most cases the kind of *ad hoc* refinement illustrated in the previous pages was not necessary. Instead, some basic and well known refinement and transformation rules, from a group of around fifty rules defined in Reference 20, were used. For instance, the rules guarantee that predicates like

$\mathbf{let}\ x_e = e'\ \mathbf{in}$

$z = g(a, ..., x, x_e)$

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

$z = \mathbf{let}\ x_e = e'\ \mathbf{in}$

$\quad g(a, ..., x, x_e)$

which can be directly translated to LML. Another rule, for transforming predicates related to lists, was presented at the end of the description of the requirements for the translator. Another simple rule indicates that predicates in the form

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

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

should be first transformed into

$z = \mathbf{if}\ p(a, ..., x)$

$\quad \mathbf{then}\ g_1(a, ..., x)$

$\quad \mathbf{else}\ g_2(a, ..., x).$

and then translated into LML. There are also more elaborated rules which are omitted here. Those rules are not automatically applied by the current implementation of the translation system, but it could be extended to deal with some of them.

In order to implement the translation system, approximately 90% of the main operations and functions of the VDM specification were refined. Approximately 80% of the refinements used transformation rules only, whereas around 20% of the refinements were performed in an *ad hoc* way. However, usually, it was also necessary to apply some very simple transformation rules following an *ad hoc* refinement.

After the refinement, which was performed manually, the parts of the system specification related to type translation were transliterated by hand to LML, following the translation schemes, generating a system for translating type definitions. This type translation subsystem was then used to automatically translate (bootstrap) the type definitions of the remaining parts of the specification. Following this step, the remaining function and operation definitions were translated by hand to LML, concluding the implementation.

Using this approach for deriving an implementation, the process of obtaining an implementation essentially consists in carrying out the necessary refinement. In this way, the reasoning needed for implementing the system is actually done at the VDM level, not at the programming language level. This approach has the advantage that most design flaws are detected early, before any executable code is actually written, contrasting with an implementation directly derived from the high level design. The effectiveness of this approach is guaranteed by the quasi-strong semantic coupling between the specification and the programming language.

This experiment gives even more confidence that functional languages are adequate for prototyping. The only serious problems are related to parsing, but those could be solved by an efficient implementation of a functional parser generator. Other functions could probably be more naturally and efficiently implemented in an imperative way. For instance, this is the case of the function for generating fresh names; this function is necessary for implementing term substitution, type inference, and for assigning names to auxiliary functions resulting from the translation. A similar drawback is the implementation of VDM states in LML: the state components are given as arguments to functions and returned as results, whereas they would be naturally implemented as global variables in an imperative language.

The main reason for chosing LML as the implementation language is that it is widely used and available. In general, any other high order, strongly typed, purely functional language would be adequate. In fact, this is also valid for the prototype derivation method. Although a language providing a more sophisticated module system, overloading, a mechanism for defining ADTs and exceptions would probably simplify the implementation and lead to a more elegant and reliable code, probably there would not be significant advantages in that choice. However, if another functional language has to be used, a translation system can be automatically derived for it; basically, it would only be necessary to provide a new pretty-printer for the translation system described here.

There is no evidence in the implementation of the translator that laziness is really an essential feature. On the other hand, it is not clear whether strictness would improve the efficiency of the prototype. For example, consider that in a strict implementation some of the information stored in the mappings necessary for the translation would be computed even if they were not going to be used. Contrast this approach with a lazy implementation, where this information is only computed when necessary, reducing the translation time. Furthermore, the strict implementation tries to evaluate the components of the VDM state, given as arguments to LML functions.

As it is not obvious that strictness would lead to a more efficient implementation, it seems more adequate to use a lazy language because of the extra expressiveness inherent to them. For instance, this expressiveness is useful for implementing a purely functional interactive interface for prototypes. Such an interface cannot be done in a natural and purely functional way using a strict language.

Laziness would also be of interest for animating specifications using the style of Reference 21, where underspecified operations are animated by functions returning a possibly infinite list of the results of the operation. By contrast, the approach introduced in Reference 4, and supported by the system discussed here, chooses a particular deterministic implementation of an underspecified operation.

Debugging

A consequence of following the approach of Reference 4 is that the bugs detected in the final program are related to errors in the specification or to mistakes in the refinement process. For fixing them, it is enough either to fix the specification or to correct a refinement. In this way, it is much easier to understand, locate and fix a bug, since this involves analyzing an abstract specification, not a concrete program having many details which are irrelevant to the reasoning.

Not many flaws were found when debugging the implementation. All of them were related to simple errors in the specification; so, no major redesign of the system was necessary. Most probably, the system had only a small quantity of bugs because most development effort was done at the specification level; contrary to the programming language level, the abstraction of the specification surely leads to more understandable and error-prone formal text, which is not the case of most "real" code.

A major difficulty for detecting bugs is the lack of a useful debugging tool for LML.

Evaluation: Performance and Use

The main objective for developing this system was to exercise the methodology suggested in Reference 4, in order to validate it in practice; efficiency was not much of a concern during its development. For example, the specification was only superficially refined: in order to obtain a more efficient implementation, more refinement steps would be necessary. Alternatively, the generated prototype could be optimized using functional calculi such as that introduced in Reference 22.

The interface of the translation system is very basic. Also, many other features and options should be added, if the system was to be really useful in a practical environment.

So far, the main experiment undergone was the bootstrapping of the system itself, when it was used to translate around 100 VDM type definitions, roughly 200 lines of VDM concrete syntax, without comments and blank lines, generating some 1800 lines of LML code. In a SPARCStation 2 with 32Mb of main memory, this translation took around 6 minutes. Considering the minimum effort required to derive the implementation, the performance of the obtained prototype is quite good.

In graduate courses on formal methods, the system has also been used with success for deriving prototypes of various specification case studies, including a process scheduling system, a payroll system and a stock control system. Prototypes of those case studies have also been manually derived using the approach of Reference 4.

In fact, the translator generates prototypes much faster than anyone can manually generate. For instance, in order to derive a working prototype of one of the case studies mentioned above a single person took on average 20 hours, whereas the translator could generate an equivalent working prototypes in at most 4 minutes, using a SPARCStation 2 with 32Mb of main memory. Furthermore, the use of the translator completely eliminates several errors that usually appear after a manual derivation: syntactic errors, type checking errors, name clashing errors and, more dangerous, semantic errors caused by the lack of attention from the person deriving the prototype. Those errors delay even more the manual process and compromise the reliability of the derived prototypes. Indeed, the translation tool is essential for derivation of prototypes from non-trivial specifications.

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[1], part of it is bootstrapped using the system itself. The bootstrapping was very important to test and validate the system.

[1]The object is available from the authors.

The system was developed over a period of five months. The authors were responsible for the design. One of them specified and implemented the main parts of the system. Two other people implemented the parsers and pretty-printers, on a part time basis.

This paper has described the whole development trajectory of this system, commenting on the lessons learned at each stage. In particular, the paper comments on the adequacy of functional languages for implementing the translation system and discusses whether laziness is really an essential feature for this kind of application.

The system was specified in VDM and implemented using an unusual approach, directly deriving LML code from the specification. The system was a very interesting case study to validate the development method introduced in Reference 4; in fact, the system provides machine support for this method, which proved to be very effective for quickly deriving a correct prototype for animating a high-level logical specification.

In order to produce a industrial strength version of the translation system, several approachs could be followed, reusing some of the artifacts that resulted from the development described in this paper. First, the generated prototype could be optimized using functional calculi [22]. However, in this context, it seems more appropriate to work at a higher level of abstraction, further refining the VDM specification, the low level design, until it is sufficiently concrete to be directly translated to an efficient implementation. In fact, based on the successful development described in this paper, this approach seems to be the more appropriate. Alternatively, a completely different approach for "low level design" and implementation could be adopted, but still reusing the high-level design which describes in detail what is necessary to perform the translation: special operations, the tables that they manipulate, etc.

The translation system could be easily adapted to generate code in another high order, strongly typed, purely functional language. For instance, a translator from VDM to Haskell [8] could be obtained by simply providing a new pretty-printer for the translation system described here. Unfortunately, the translator cannot be so easily adapted to accept specifications in a language that is not similar to VDM. For example, in order to translate Z [5] specifications it would be necessary to provide extra mechanical support for calculating operations' pre-conditions, identifying units of specification that correspond to operations, and translating the operators that are used to structure specifications.

Although there are a number of works on derivation of functional prototypes from model-based specification languages [4], most of them just describe the derivation method, by either presenting the translation of specific case studies or giving an informal description of the translation rules. Indeed, the VDM specification of the system described here completely formalizes those translation rules.

References 23, 24, and 25 report on systems for automatically doing the translation from specification to prototype. The first is basically a syntax directed translator in which case only a very restricted subset of VDM can be automatically translated; much more can be done by the system described here. From the viewpoint of software development, its interest is restricted. On the other hand, References 24 and 25 consider a larger subset of VDM than considered here, but they were developed using completely different techniques; both are completely implemented in imperative languages.

Acknowledgements

Ana Bastos and Simone Santos implemented the parsers and pretty-printers of the system described here. The anonymous referees helped to improve the paper.

References

[1] Cliff Jones. *Systematic Software Development using VDM.* Prentice-Hall, second edition, 1986.

[2] Lennart Augustsson and Thomas Johnsson. The chalmers Lazy-ML compiler. *The Computer Journal*, 32(2):127–141, April 1989.

[3] Lennart Augustsson. The interactive Lazy ML system. *Journal of Functional Programming*, 3(1):77–92, 1993.

[4] Paulo Borba and Silvio Meira. From VDM specifications to functional prototypes. *Journal of Systems and Software*, 21(3):267–78, June 1993.

[5] J. Michael Spivey. *The Z Notation: A Reference Manual.* Prentice-Hall, 1989.

[6] Kevin Lano and Howard Haughton, editors. *Object-oriented Specification Case Studies*, chapter 6: Object-oriented Specification in VDM^{++}. Object-oriented Series. Prentice-Hall International, 1993.

[7] Silvio Meira and Ana Cavalcanti. MooZ case studies. In S. Stepney, editor, *Comparative Study of Object Orientation in Z.* Springer-Verlag, 1992.

[8] Paul Hudak, Simon Peyton Jones, Philip Wadler, et al. Report on the functional programming language haskell. *ACM SIGPLAN Notices*, 27(5), May 1992. Version 1.2.

[9] VDM Specification Language – Proto-Standard. Technical report, British Standards Institution, December 1992. BSI IST/5/19 N-246B.

[10] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, 1987.

[11] Bertrand Meyer. *Object-Oriented Software Construction.* Prentice Hall, 1988.

[12] David Parnas. A technique for the specification of software modules with examples. *Communications of the ACM*, 15(5):330–336, May 1972.

[13] Ian Hayes and Cliff Jones. Specifications are not (necessarily) executable. *IEE/BCS Software Engineering Journal*, 4(6):330–338, November 1989.

[14] G. Uddeborg. A functional parser generator. Licenciate Thesis, Department of Computer Sciences, Chalmers University of Technology, 1988.

[15] Philip Wadler. How to replace failure by a list of successes. In Jean-Pierre Jouannaud, editor, *Functional Programming Languages and Computer Architecture*, number 201 in Lecture Notes in Computer Science. Springer-Verlag, September 1985.

[16] John Levine, Tony Mason, and Doug Brown. *UNIX Programming Tools—lex & yacc*. O'Reilly & Associates, 1992.

[17] Paulo Borba and Silvio Meira. From model based specifications to functional prototypes. In *IEEE TENCON'91 Session on Rapid Prototyping with Functional Programming Languages*, New Delhi, India, August 1991.

[18] Carroll Morgan. *Programming from Specifications*. Prentice Hall, second edition, 1994.

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

[20] Paulo Borba. From formal specifications to functional prototypes (in portuguese). MSc. Thesis, Universidade Federal de Pernambuco, Departamento de Informática, Brazil, July 1991.

[21] M. Johnson and P. Sanders. From Z specifications to functional implementations. In John Nicholls, editor, *Z User Workshop*, Workshops in Computing. Springer-Verlag, December 1990.

[22] Lambert Merteens. *Algorithmics—towards programming as a mathematical activity*. North-Holland, 1987.

[23] G. O'Neil. Automatic translation of VDM specifications into Standard ML programs. *The Computer Journal*, 35(6), 1992.

[24] Manfred Haß. Development and application of a Meta-IV compiler. In *VDM'87, VDM—A Formal Method at Work*, volume 252 of *Lecture Notes in Computer Science*. Springer-Verlag, March 1987.

[25] Michael Andersen, René Elmstrom, Poul Lassen, and Peter Larsen. Making specifications executable using IPTES Meta-IV. In *Euromicro'92*, pages 521–8. September 1992.

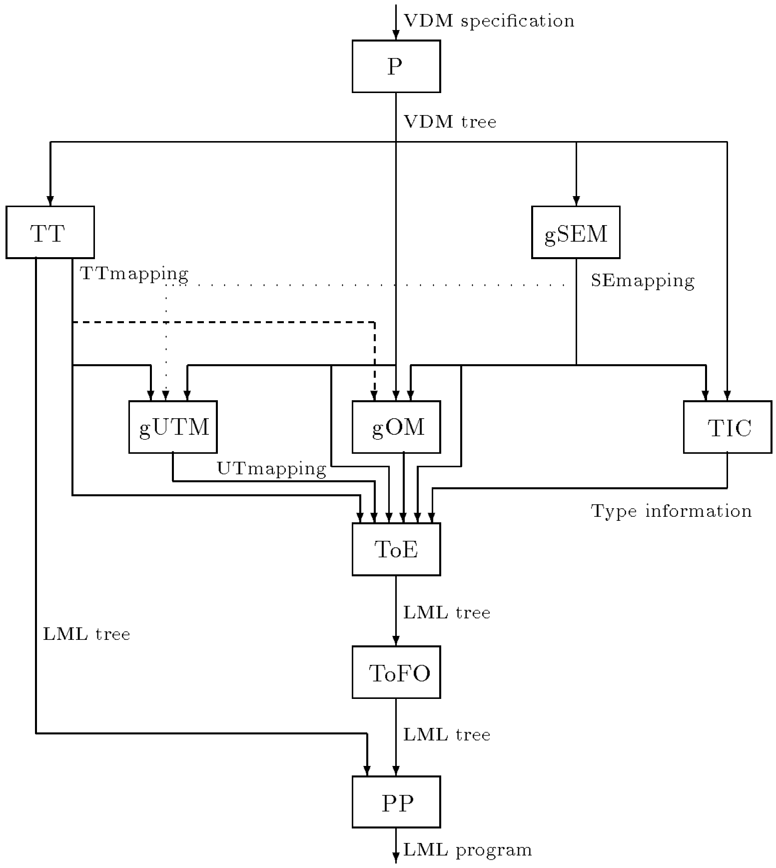

Figure 1: System Architecture.

Module	Size
1	350
2	1386
3	1640
4	966
5	96

Table 1: Specification modules sizes.