Making Specifications Executable – Using IPTES Meta-IV*

Michael Andersen, René Elmstrøm, Poul Bøgh Lassen, Peter Gorm Larsen
e-mail: mian@ifad.dk, rene@ifad.dk, poul@ifad.dk, peter@ifad.dk
The Institute of Applied Computer Science (IFAD)
Munkebjergvænget 17
DK-5230 Odense M, Denmark

Abstract

This article discusses the extra effort required to make specifications executable. With an origin in essential (but in general non-executable) specification techniques and specification language constructs the limitations of writing specifications in an executable language are discussed. The (executable) example specifications are written in IPTES Meta-IV which is a subset of BSI/VDM-SL.

1 Introduction

In ESPRIT project EP5570 IPTES[1] [Pulli&91] the Ward/Mellor notation of SA/RT [Ward&85] is being formalized to allow execution and animation of SA/RT models. As a part of the IPTES project a language has been developed for specifying data transformations in SA/RT (pure functional transformations).

Since the IPTES project aims at developing a CASE tool with animation of the system models, executability is required from the specifications of the data transformations. But where other approaches have chosen existing programming languages to express functional data transformations, our basis is a (non-executable) specification language. We have selected an executable subset of this language and specified and implemented an interpreter for that subset.

*The work reported here is partially sponsored by the CEC ESPRIT programme, contract no. EP5570

[1]An acronym for "Incremental Prototyping Technology for Embedded real-time Systems".

Our specification language is called IPTES Meta-IV and it is derived from BSI/VDM-SL[2] [BSIVDM91]. This standard also includes an ASCII syntax that we conform to as well. The language includes looseness [Larsen&91] which is a strong concept not normally found in executable specification languages. This topic will be further described in Section 4.

In IPTES the specifications of data transformations form running prototypes and there are benefits as well as drawbacks from this approach. In [Hayes&89] some arguments are put forward against executable specification languages. The arguments run along the lines, that there are a number of very strong specification techniques which in general are non-executable. In this paper we will examine some examples of non-executable specifications (taken from [Hayes&89]), and show how an executable IPTES Meta-IV specification can be derived.

The examples in this paper are written according to the mathematical syntax of VDM-SL. Example functions with signatures are executable whereas the quantified equations in general are not. All the functions listed have originally been written in the standard ASCII syntax and tested with our interpreter. With each example there is a short discussion of the cost of turning the specification into an executable one.

In Section 2 below there is a general introduction to the topics which are discussed. After that Section 3 discusses deterministic operations and ways to use them. Section 4 discusses looseness as opposed to determinism and the special problems of executing loose specifications. Section 5 describes related work in the area of executable specification languages. In Section 6 we sum up the results together with the status of the work on the interpreter for IPTES Meta-IV.

2 Specifications as Prototypes

In this section we take a prototype to denote a system that has the same functionality as the final (intended) system though it may diverge in system structure and performance.

Whether prototypes are used evolutionary, incrementally or on a throw-away basis they can be written in an executable specification language. From that point of view there are several advantages in having executable specification [Hekmatpour&88]. These advantages include:

- Once the specification (and verification) of a system has been completed there is no extra cost involved in producing a prototype. Test data must possibly be produced.
- A precise level of documentation is always available to the developer.

[2]Meta-IV is the original name for the specification language for the VDM methodology also called VDM-SL.

- A specification gradually evolves towards user requirements and has at each stage a precise description of the system.

These advantages justify that one spends some time making specifications executable given that the sacrifices (to clarity, expressiveness etc.) are reasonable.

Existing executable specification languages can be divided into the property-oriented (axiomatic) and the model-oriented ones. This paper is only concerned with one model-oriented executable specification language namely IPTES Meta-IV – an executable subset of VDM-SL.

3 Deterministic Operations

A specification which returns a unique result for a given input is deterministic. When a specification allows a choice between a number of different results it is termed 'loose'.

In the following sections we will examine some deterministic specification techniques and discuss the cost of making them executable. The examples are taken from [Hayes&89].

3.1 Specifying by Inverse

In the case that there are no known functions with which to specify like e.g. when dealing with novel concepts, it can sometimes be useful to constrain a function by applying the inverse of a known function. As an example, suppose that (integer) square root is to be specified, but that only (integer) squaring is known. Then the following would apply:

$$\forall\, r, n \in \mathbf{N} \cdot \mathit{Isqrt}(n) = r \;\Leftrightarrow\; r^2 \leq n < (r+1)^2$$

The constraining equation is not executable because the result r lies in $\mathbf{N}$ and the set of natural numbers is an infinite set. However a little reasoning (about integer squaring) will overcome this problem. Since

$$\forall\, i \in \mathbf{N} \cdot i \leq i^2$$

we can limit our search for r to the set $\{1, \ldots, n\}$ and then the specification becomes executable:

$$\mathit{Isqrt} : \mathbf{N} \to \mathbf{N}$$

$$\mathit{Isqrt}\,(n) \triangleq$$
$$\quad \textbf{let}\ r \in \{1, \ldots, n\}\ \textbf{be st}\ r^2 \leq n \wedge n < (r+1)^2\ \textbf{in}\ r$$

In this example we used integer arithmetic to put restrictions on the set in which to search for r. In many cases one can limit the search space by applying knowledge external to the specific problem but still within the general field (here integer arithmetic).

It should be noted that countably infinite sets alone do not exclude executability if one is using a lazy evaluation language. Since VDM-SL is a strict evaluation language infinite sets cannot be handled.

3.2 Combining Clauses in a Specifications

Many specifications need to combine a number of properties to have a working (complete) specification and the operators of predicate calculus can be used to express such combinations.

Consider the following specification for sorting a sequence:

$$\forall\, in, out \in \mathbf{N}^* \cdot$$
$$\quad Sort(in) = out \;\Leftrightarrow\; IsOrdered(out) \wedge IsPermutation(in, out)$$

where the auxiliary predicates are defined as:

$$IsOrdered : \mathbf{N}^* \to \mathbf{B}$$
$$IsOrdered\,(l) \triangleq$$
$$\quad \forall\, i, j \in \mathbf{inds}\ l \cdot i < j \;\Rightarrow\; l(i) \leq l(j)$$

$$IsPermutation : \mathbf{N}^* \times \mathbf{N}^* \to \mathbf{B}$$
$$IsPermutation\,(l_1, l_2) \triangleq$$
$$\quad \forall\, e \in (\mathbf{elems}\ l_1 \cup \mathbf{elems}\ l_2) \cdot$$
$$\qquad \mathbf{card}\ \{i \mid i \in \mathbf{inds}\ l_1 \cdot l_1(i) = e\} = \mathbf{card}\ \{i \mid i \in \mathbf{inds}\ l_2 \cdot l_2(i) = e\}$$

This specification says absolutely nothing about *how* to sort the sequence – it just states that the output must be sorted and tells what the relationship between input and output is (the output must be some permutation of the input).

The specification is, however, not executable because given any input *in* we have no restriction on the set in which to find the output *out*. The cost of making it executable is relatively small – by looking at the predicate we discover that any output *out* must lie in the set of permutations of *in*. Hence we can write

$$\forall\, in \in \mathbf{N}^* \cdot \forall\, out \in Permutations(in) \cdot$$
$$\qquad Sort(in) = out \;\Leftrightarrow\; IsOrdered(out)$$

where the function *Permutations* generates the set of permutations of *in* from which *out* is chosen. The cost is that we must construct the set of permutations

instead of just testing if a candidate *is* a permutation. In IPTES Meta-IV the definitions could be:

$$
\begin{array}{l}
Sort : \mathbf{N}^* \to \mathbf{N}^* \\
Sort\,(in) \triangleq \\
\quad \textbf{let } out \in Permutations(in) \textbf{ be st } IsOrdered(out) \textbf{ in } out
\end{array}
$$

$$
\begin{array}{l}
Permutations : \mathbf{N}^* \to \mathbf{N}^*\textbf{-set} \\
Permutations\,(l) \triangleq \\
\quad \textbf{cases } l : \\
\qquad [], [e] \quad \to \{l\}, \\
\qquad \textbf{others } \to \bigcup \{Permute(l, i) \mid i \in \textbf{inds } l\} \\
\quad \textbf{end}
\end{array}
$$

$$
\begin{array}{l}
Permute : \mathbf{N}^* \times \mathbf{N}_1 \to \mathbf{N}^*\textbf{-set} \\
Permute\,(l, i) \triangleq \\
\quad \{[l(i)] \frown l' \mid l' \in Permutations([l(j) \mid j \in \textbf{inds } l \backslash \{i\}])\}
\end{array}
$$

Here we took the predicate that expressed the permutation relation between input and output and used it in a constructive way to constrain the search for the output.

In general it is often possible to use predicates that relate input to output to construct sets of values in which to search for the result.

3.3 Negation in Specifications

In the previous subsection we illustrated the power of combining clauses by conjunctions. In this subsection we will analyze an example in which negation is used. In many cases it is easier to express what the solution should *not* be instead of listing all the properties that the solution should have.

If we want to specify the greatest common divisor (gcd) for a set of numbers, we can start by defining a common divisor predicate

$$
\begin{array}{l}
IsCD : \mathbf{N}_1 \times \mathbf{N}\textbf{-set} \to \mathbf{B} \\
IsCD\,(d, s) \triangleq \\
\quad \forall\, i \in s \cdot i \textbf{ mod } d = 0
\end{array}
$$

and then state what a greatest common divisor must fulfill:

$$
\begin{array}{l}
\forall\, s \in \mathbf{N}\textbf{-set}, d \in \mathbf{N}_1 \cdot \\
\qquad gcd(s) = d \;\Leftrightarrow\; IsCD(d, s) \wedge \neg\,(\exists\, e \in \mathbf{N}_1 \cdot IsCD(e, s) \wedge e > d)
\end{array}
$$

So a GCD is a common divisor itself, and there must not exist a common divisor that is even greater.

Here we can restrict d (and e) to members of the set $\{1, \ldots, Min(s)\}$, because a common divisor cannot be larger than the smallest of the elements in the argument set. Computing the lowest number of a set can be specified by saying that it must *not* be the case that the set contains an even lower number than the one computed.

$$
\begin{array}{l}
Min : \textbf{N-set} \to \textbf{N} \\
Min\,(s) \triangleq \\
\quad \textbf{let } m \in s \textbf{ be st } \neg\,(\exists\, e \in s \cdot e < m) \textbf{ in } m
\end{array}
$$

With the above mentioned restriction we can now write the following executable specification:

$$
\begin{array}{l}
gcd : \textbf{N-set} \to \textbf{N}_1 \\
gcd\,(s) \triangleq \\
\quad \textbf{let } d \in \{1, \ldots, Min(s)\} \textbf{ be st} \\
\qquad IsCD(d, s) \wedge \neg\,(\exists\, e \in \{1, \ldots, Min(s)\} \cdot IsCD(e, s) \wedge e > d) \textbf{ in } d
\end{array}
$$

As in Section 3.1, we used integer arithmetics to constrain an infinite set to a finite subset thus making it executable.

In all the examples shown so far we have in different ways constrained infinite definition domains to finite sets thus turning specifications into executable ones. The generated sets in which to search may be arbitrarily large (depending on the input). Of course this can give poor performance, but then it should be remembered that IPTES Meta-IV is an executable specification language and not an ordinary programming language where efficiency could be a major concern.

4 Looseness

An important specification technique is the use of looseness which is when a (part of a) specification can return any value from a set of equally valid possibilities. For each choice there exists a corresponding implementation and these implementations might differ in their behavior (internally or externally).

A problem which is present when executing loose specifications is that the execution has to choose among the possibilities (implementations) to deliver a single value as result, hence it cannot display all the looseness in the specification. Furthermore an implementation might make other choices (than those made in the execution of the specification) meaning that the results cannot directly be compared. This is one of the main arguments against executable specifications. In this section we will show how it can still be meaningful to use looseness and

executable specifications in combination.

There are some computer systems where their actual behavior should not be to closely determined by the specification and for such purposes looseness is a useful specification technique because it enables the designer to postpone certain decisions to a later stage in the development (as e.g. the final implementation).

Looseness can also be seen as a means to specify at a higher level of abstraction than that of a final implementation (which is often advisable) thus leaving as much freedom to the implementor as possible.

The next sections will discuss different types of looseness and try to illustrate their impact on a computation.

4.1 External Visible Looseness

We use the term external visible looseness when a specification allows different implementations that differ in their external behavior. To show an example we use the sorting example from Section 3.2 but now we use records instead of natural numbers, where the records have a key field and a data field. A record (from a bank system) could have the structure:

$$\begin{aligned} \mathit{Transaction} :: & \mathit{account} : \mathbf{N} \\ & \mathit{amount} : \mathbf{R} \end{aligned}$$

and *IsOrdered* must be changed so it uses the account numbers.

$$\mathit{IsOrdered} : \mathit{Transaction}^* \to \mathbf{B}$$

$$\begin{aligned} &\mathit{IsOrdered}\,(l) \triangleq \\ &\quad \forall\, i, j \in \mathbf{inds}\ l \cdot i < j \ \Rightarrow\ l(i).\mathit{account} \leq l(j).\mathit{account} \end{aligned}$$

Now the ordering predicate only tells that the sequence of records should be sorted with respect to the key field (the account number). That would allow records with identical keys and different data to appear in different order in the result.

Consider a list with only two *Transaction*s, each having the same account number but with different amounts. The set of permutations of this list holds two lists since the elements in the list *are* different (w.r.t. the amount field). Both these sequences satisfies the *IsOrdered* predicate and the specification can return either one.

The implementation of loosely specified constructs in the IPTES Meta-IV interpreter is under-determined[3] and the consequences are that the interpreter cannot exhibit the described external visible looseness.

[3]Meaning that our choices are static.

It is however rather important to be aware of all the looseness that is present in a specification and for that purpose we have envisaged a looseness analysis tool to show the different choices that can be made (and their consequences) during a computation.

The importance lies in the fact that it should be possible to compare (the execution of) the specification with an implementation. If the specification contains looseness then it should be possible to check that the result (from the implementation) is a legal one i.e. that the specification allows that specific implementation.

In other words – when external visible looseness is contained in a specification one has to know all possible results in order to show that an implementation fulfills the implementation relation[4].

4.2 Internal Visible Looseness

A system can contain loosely specified components and still show an overall deterministic behavior. As an example consider a function to compute the sum of a sequence of natural numbers.

$$
\begin{array}{l}
Sum : \mathbf{N}^* \to \mathbf{N} \\
Sum\,(l) \triangleq \\
\quad \textbf{cases } l : \\
\quad\quad [] \quad\quad \to 0, \\
\quad\quad [e] \quad\quad \to e, \\
\quad\quad l' \frown l'' \to Sum(l') + Sum(l'') \\
\quad \textbf{end}
\end{array}
$$

The function uses recursion on the structure on the data (the sequence) by splitting up the sequence into two smaller parts. This is a loose specification because the pattern $(l' \frown l'')$ is a loose pattern – it does not tell how to split the sequence but rather that some splitting among the possible ones should be selected and applied.

If we tried to compute $Sum([2,7,1])$ and used the above mentioned looseness analysis tool to tell us our possibilities at any moment then it would tell us that the so-called sequence-concatenation-pattern could result in the following bindings of the variables l' and l''

$$
\begin{array}{l}
\{l' \mapsto [2], l'' \mapsto [7,1]\} \\
\{l' \mapsto [2,7], l'' \mapsto [1]\}
\end{array}
$$

In each case the concatenation of l' and l'' yield the argument sequence and any choice will end up with the same result (10).

[4]The implementation relation is the one that states that the implementation 'satisfies' the specification by only producing results allowed by the specification.

Since some (internal) choices can lead to external visible looseness one cannot separate the two types of looseness. One should also be aware of the different ways to interpret looseness [Wieth89] when implementing the specifications. There are (at least) two ways to interpret looseness - either under-determinedly or non-deterministically. They correspond to a static respectively a random choice among the possibilities leading to deterministic respectively non-deterministic behavior.

5 Related Work

There exist a number of other executable languages which have more or less been inspired by VDM. In [Plat&89] an overview of existing tool support for VDM is presented. Only two projects dealing with an executable subset of VDM-SL are given in that overview: the Meta-IV compiler project from Kiel University [Haß87], and the EPROS project where both an interpreter and a compiler for a language called EPROL (strongly inspired by VDM) have been developed [Hekmatpour&88]. In addition to these two we have looked at 'me too' [Alexander&90]. The main difference between our executable language and the existing ones is the *generality* of the pattern matching and the powerful let ... be st ... in ... construct. It is exactly these facilities for introducing a combination of looseness and executability that we have used in our examples in this paper.

6 Concluding Remarks

We have presented a number of cases where non-executable specifications could be transformed into executable ones without any significant loss of expressiveness or level of abstraction. In many cases it is even possible to maintain the same style of specification when constrained to executable specifications.

There is a cost connected with turning a non-executable specification into an executable one. It is our experience though that the cost is often relatively small because the necessary reasoning only requires little extra knowledge about the problem domain. This applies to our examples as well. Apart from that it is our experience that a number of other advantages exist – they include:

- The designer can convince himself that the specification denotes what was intended by trying out small examples.
- It is still possible to carry out formal correctness proofs.
- Testing techniques can be used to validate specification.

We acknowledge, though, that there are a number of problems involved in restricting specifications to executable cases. One of the problems is that adding

information to a specification in order to make it executable can make the specification less clear. Another one is that the techniques we have presented in this article to make specifications executable require additional information about the problem domain. This information might not be present in all cases (especially when dealing with novel concepts). Furthermore adding information to a specification can in some cases make it more difficult to prove properties about the specification.

On the other hand it is our experience when writing specifications in BSI/VDM-SL that even when working with non executable specifications one can often take advantage of restricting the specification to an executable case expressed in IPTES Meta-IV. In this way formal (non-executable) specifications in BSI/VDM-SL can be developed in an incremental prototyping way by using (restricted) executable IPTES Meta-IV specifications in the process before producing the final specification which can be subject to e.g. formal proof or (more commonly) informal validation techniques like reviews.

The status of our work on the IPTES Meta-IV language is that a scanner, a parser and a basic version of an interpreter for the language have been implemented and tested. We plan to include full type checking of the language, and support debugging and different analysis methods (like e.g. the looseness analysis mentioned in Section 4.1).

Acknowledgements

We would like to thank Nico Plat, Ole Bjerg Larsen and Marcel Verhoef for many valuable remarks on earlier drafts of this article.

The IPTES consortium is formed by IFAD (Denmark), VTT (Finland), MARI (United Kingdom), CEA/LETI (France), ENEA (Italy), Synergie (France), Universidad Politécnica de Madrid (Spain), Telefónica I+D (Spain), and Politecnico di Milano (Italy).

References

[Alexander&90] Heather Alexander and Val Jones. *Software Design and Prototyping using Me Too.* Prentice Hall, 1990. 279 pages.

[BSIVDM91] *VDM Specification Language – Proto-Standard.* Technical Report, British Standards Institution, December 1991. 237 pages. BSI IST/5/50 N-231.

[Haß87] Manfred Haß. Development and Application of a Meta IV Compiler. In *VDM – A Formal Method at Work*, pages 118–141, Institut für Informatik und Praktische Mathematik, Springer–Verlag, March 1987. 24 pages.

[Hayes&89] I.J. Hayes, C.B. Jones. Specifications are not (necessarily) executable. *Software Engineering Journal*, 330–338, November 1989.

[Hekmatpour&88] Sharam Hekmatpour and Darrel Ince. *Software Prototyping, Formal Methods and VDM.* Addison-Wesley, 1988. 222 pages.

[Larsen&91] Peter Gorm Larsen and Poul Bøgh Lassen. An Executable Subset of Meta-IV with Loose Specification. In *VDM'91 Symposium*, VDM Europe, Springer-Verlag, March 1991. 15 pages.

[Plat&89] Nico Plat and Hans Toetenel. *Tool support for VDM.* Technical Report 89-81, Delft University of Technology, November 1989. 52 pages.

[Pulli&91] P. Pulli, R. Elmstrøm, G. León, J.A. de la Puente. IPTES– Incremental Prototyping Technology for Embedded real-time Systems. In *ESPRIT Information Processing Systems and Software, Results and Progress of Selected Projects 1991*, pages 497–512, Esprit, Commission of the European Communities, November 1991. 16 pages.
IPTES Doc.id. :IPTES-IFAD-61-V1.1.

[Ward&85] P.T. Ward and S.J. Mellor. *Structured Development for Real-Time Systems.* Volume 1-3, Yourdon Press, New York, 1985-1986.

[Wieth89] Morten Wieth. Loose Specification and its Semantics. In G.X. Ritter, editor, *Information Processing 89*, pages 1115–1120, IFIP, North-Holland, August 1989. 6 pages.