An Executable Subset of Meta-IV with Loose Specification*

Peter Gorm Larsen and Poul Bøgh Lassen
The Institute of Applied Computer Science (IFAD)
Munkebjergvænget 17, DK-5230 Odense M, Denmark
E-mail: peter@ifad.dk and poul@ifad.dk

Abstract

In ESPRIT project no. EP5570 called IPTES[1] a methodology and a supporting environment for incremental prototyping of embedded computer systems is developed. As a part of this prototyping tool an interpreter for an executable subset of a VDM dialect is developed. Based on a comparative study of different notations inspired by VDM we have now selected an executable subset of the BSI/VDM-SL[2] notation. This executable subset is interesting because it enables the designer to use loose specification. None of the executable VDM dialects which we have investigated contain as large a part of looseness as our subset does. In this article we will focus mainly on which constructs we have in this subset and how we have dealt with the looseness. Furthermore we will sketch the connection between the semantics of our subset and the semantics for the full BSI/VDM-SL.

1 Introduction

IPTES is an ESPRIT research project which aims at development of a methodology and a supporting environment for incremental prototyping of embedded computer systems. Initially a system is described by means of a high-level graphical specification and design language SA/RT[3] (see [Ward&85]) where BSI/VDM-SL (also called Meta-IV) is used in the so-called mini-specifications to specify sequential components. The SA/RT specifications are made executable by a transformation to high-level timed Petri nets (see [Ghezzi&91]), while the mini-specifications are interpreted by an interpreter which is going to conform to the operational semantics as described in this article. Parts of the specification can then gradually be transformed towards actual code, and in this way

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

[1]IPTES is an acronym for "Incremental Prototyping Technology for Embedded real-time Systems".

[2]BSI/VDM-SL is an acronym for "British Standards Institution/Vienna Development Method Specification Language". However, since the standardization effort is being carried forward within ISO it is possible that in the future this language will just be called VDM-SL.

[3]SA/RT is an acronym for the "Structured Analysis Real Time extension".

heterogeneous prototyping of the system can be performed. More information about the IPTES architecture can be found in [Leon&91].

We have chosen to use an executable subset of the BSI/VDM-SL. This language is developed in order to harmonize the different VDM dialects into one standard language. This standardization effort is currently done under the auspices of BSI and ISO and it involves the definition of a concrete syntax, an abstract syntax, a static semantics, and a dynamic semantics (see [BSIVDM91]). We have taken these four parts as our starting point and have selected a subset of BSI/VDM-SL with minor adjustments to make the language fit the IPTES architecture.

An important quality of BSI/VDM-SL is that it contains constructs which can be loosely specified. When a specification is deterministic it simply denotes one model (corresponding to the only valid implementation of the functionality of the specification). When a specification is loose it denotes a set of models (corresponding to the different valid implementations of the specification).

Our executable subset has an operational semantics (written using the complete version of the BSI/VDM-SL notation). When a specification written in BSI/VDM-SL contains looseness it will denote a set of possible models in the dynamic semantics. In our operational semantics an arbitrary of these models is returned. The relation between the dynamic semantics for BSI/VDM-SL and this operational semantics is further analysed in this article.

After this introduction we will explain the notion of loose specification. We will then present the constructs which we have selected for our subset. Then we will explain the semantics of the subset and shortly compare it to the standard dynamic semantics. This is followed by a few examples illustrating the expressiveness provided by the generality of the patterns in our subset. After that, we will compare our approach to some of the related work on executable subsets of VDM-SL. Finally we will give a few concluding remarks and identify future work.

2 Loose Specification

Loose specification occurs if a specifier wants to express that it does not matter which particular value an expression yields as long as it fulfills certain requirements. Thus, loose specification arises because a specification generally needs to be stated at a higher level of abstraction than that of the final code of the system. When loose specification is used, the question of how to interpret this looseness is often ignored. However, the interpretation is important, especially if a specification must be proven to implement another specification.

As shown in [Larsen&89], [Wieth89], and [Larsen90] there are at least two different ways of interpreting such loose specifications. These two different interpretations have been called 'nondeterminism' and 'underdeterminedness'. When a loosely specified construct is interpreted as underdetermined it denotes the set of all possible deterministic implementations of that construct. If the same construct is interpreted as nondeterministic instead, it denotes the set of all possible implementations of that construct (including the nondeterministic ones). In this work we have chosen to use the underdetermined interpretation, where we, by means of a deterministic algorithm, will select an arbitrary

result (from the set of deterministic implementations) satisfying the specification. We have chosen the underdetermined interpretation in order to achieve the property of referential transparency. If the execution of a specification written in our executable subset of BSI/VDM-SL complies with the user's expectation, it means that there exists at least one model which is satisfactory. However, this is no guarantee that all the other models satisfy his requirements to the functionality. Thus, in general, it is not sufficient simply to give such a loose specification to somebody else for implementation. Some knowledge about which of the models that has been selected by means of the deterministic algorithm must also be taken into account.

3 Constructs in the Executable Subset

Not all BSI/VDM-SL constructs are executable. This applies to implicit function and operation definitions and all computations over infinite sets (types with infinitely many values). In addition to these constructs, a number of executable constructs have been excluded from our subset. An SA/RT specification gives a hierarchical and a graphical overview of the structure of a system. At the bottom level of such a specification, mini-specifications are used to specify how the input values are transformed to output values in a sequential way. In IPTES such algorithms will be described by means of an executable subset of BSI/VDM-SL which is presented in this article.

We have therefore analysed the expressive power that is necessary to describe the algorithms in the mini-specifications. The result of this analysis was that the individual mini-specifications are relatively small and have a simple structure. As a result of this, and the fact that we cannot expect industrial users of the IPTES tools to be familiar with functional programming, we have excluded advanced features like lambda expressions, polymorphism, locally defined functions, and recursive let expressions.

Included in the subset are state definitions, constant (value) definitions, type definitions, explicitly defined functions and operations. The full generality of patterns from BSI/VDM-SL and most of the expression and statement constructs are also included. In order to support a simple library facility including auxiliary functions, a subset of the module concept from BSI/VDM-SL is also adopted in the language.

3.1 Abstract Syntax

We will informally explain what is included, and use BSI/VDM-SL to describe selected parts of the abstract syntax for our subset. Most of these parts will be used in the semantic description in the following section.

Modules

A simple library facility is included in the subset by allowing modules with simple imports and exports. This has been restricted compared with the modules from BSI/VDM-SL so that state changing operations cannot be exported or imported. In the same way it is not allowed to have cyclic imports. Type information about the imported and exported entities enables type checking of a module without analyzing the imported modules.

Type Definitions

The type definitions contain a number of basic types and a number of type constructors. The basic types include Boolean, numeric types (natural numbers, integers, and reals), characters, and enumeration types (quote literals). The type constructors include a composite type constructor (producing records), a union type constructor, a product type constructor, an optional type constructor, a set type constructor, a sequence type constructor and a map type constructor. Thus all type constructors from BSI/VDM-SL are included except a few numeric types, the token type, the non-empty sequence type, the function type, and the injective map type. Invariants on the type definitions can also be used in the same way as in BSI/VDM-SL.

State Definitions

The state definition consists of a composite type (the components of the composite type can be considered as the global variables), possibly an invariant on the state type, and possibly an initialization function. This is equivalent to BSI/VDM-SL, except that the initialization function is not formulated as a truth-valued predicate like in BSI/VDM-SL but as an expression returning the initial value. This change is made to ensure executability in the general case.

Value Definitions

Constant (value) definitions are also included in the same way as in BSI/VDM-SL. Thus, the left-hand sides of the definitions can be general patterns.

Function Definitions

The function definitions are similar to a combination of explicit and implicit function definitions from BSI/VDM-SL. However, our abstract syntax differs from the explicit function definition part of BSI/VDM-SL by disallowing polymorphic functions and Curried functions, and by supporting a post-condition. The interpreter will then be able to check whether the arguments used in a function call satisfy the pre-condition and whether the result of the function also satisfies the post-condition.

Operation Definitions

The abstract syntax for operation definitions is similar to the abstract syntax for function definitions. The main difference is that the body is a statement instead of an expression. In addition, there is some information about which parts of the state are used by the operation. Finally it is possible to declare a number of local variables used in the operation.

Our abstract syntax differs from the explicit part of BSI/VDM-SL by allowing a post-condition to be connected to an explicitly defined operation. The locally defined variables are also dealt with differently in BSI/VDM-SL, where they can be defined inside any block statement. However, we have decided that we will permit only locally defined variables at the outermost level in operations. Thus, we have not included block statements.

Expressions

The expressions in our subset include all the expression constructs from BSI/VDM-SL except for the iota expression, the lambda expression and instantiation of polymorphic functions.

The major differences between the expressions in our subset and those in BSI/VDM-SL are:

- general bindings from BSI/VDM-SL (type bind and set bind) have been restricted to set bindings. This restriction is made in order to ensure executability.
- pattern matching cannot be constrained with additional type information.

In the following we will present the abstract syntax for those kinds of expressions we will use to illustrate the semantics with.

The abstract syntax for a let expression is:

$$\begin{array}{l} \mathit{LetExpr} :: \mathit{loc} : \mathit{Pattern} \xrightarrow{m} \mathit{Expr} \\ \qquad\qquad \mathit{in} \ : \mathit{Expr} \end{array}$$

where the *loc* field contains a collection of patterns to be matched against the corresponding expressions. The let expression denotes the value of the *in* expression evaluated in an environment where the pattern matchings have been performed. However, it should be noted that we have restricted the collection of definitions to non-recursive ones because the mutually recursive definitions used in BSI/VDM-SL are not needed for the IPTES mini-specifications. Another difference from the standard is that we (for the same reason) have removed the possibility of defining local functions.

The abstract syntax for a let-be-such-that expression is:

$$\begin{array}{l} \mathit{LetBeSTExpr} :: \mathit{bind} : \mathit{SetBind} \\ \qquad\qquad\qquad\quad \mathit{st} \quad : \mathit{Expr} \\ \qquad\qquad\qquad\quad \mathit{in} \quad : \mathit{Expr} \end{array}$$

where the *bind* is a binding of a pattern to a finite set, *st*, is a predicate using the pattern identifiers from the *bind*. The expression denotes the value of the *in* expression in an environment where a successful pattern matching has been performed and the *st* predicate is satisfied. This abstract syntax is equivalent to the one used in BSI/VDM-SL except that the general binding has been restricted to a set binding for the reason explained in the beginning of this section. This expression may contain looseness and the next section illustrates how this looseness is dealt with.

The abstract syntax for quantified expressions ($\forall$ and $\exists$) is:

$$\begin{array}{l} \mathit{AllOrExistsExpr} :: \mathit{quant} : \mathit{AllOrExistsQuantifier} \\ \qquad\qquad\qquad\quad \mathit{bind} \ : \mathit{SetBind}\textbf{-set} \\ \qquad\qquad\qquad\quad \mathit{pred} \ : \mathit{Expr} \end{array}$$

$$\mathit{AllOrExistsQuantifier} = \textsc{all} \mid \textsc{exists}$$

This is equivalent to BSI/VDM-SL except that the general binding has been restricted to a set binding as explained above. However, the operational semantics differs from the BSI/VDM-SL semantics which will be discussed in the next section.

Patterns

All pattern constructs from BSI/VDM-SL have been included in our subset. Here we will only present the abstract syntax for a set union pattern because is the only pattern this construct which will be given semantics in the next section.

The abstract syntax for set union patterns is:

$$\begin{aligned} \mathit{SetUnionPattern} :: & \mathit{lp} : \mathit{Pattern} \\ & \mathit{rp} : \mathit{Pattern} \end{aligned}$$

where *lp* union *rp* are matched against a set value. The two patterns must be matched to two disjoint subsets of the set value. This construct is used when one wants to split a set into two disjoint sets and the resulting binding is therefore loosely specified.

Bindings

As explained above we have included only the set binding in order to make bindings executable.

Statements

We have included all the statements from BSI/VDM-SL except for the block statement, the non-deterministic statement, the identity statement, and the exit mechanism.

4 The Semantics of the Executable Subset

The semantics presented here is operational, and it is inspired by [Bjørner91] where a stack semantics of a Simple Applicative Language (SAL) is presented. However, we are only using a stack of environments, and not a stack of values. This difference is caused by the fact that the target for the development of the interpreter in [Bjørner91] was a stack machine, while our target is a high level programming language (C++).

The semantics of the executable subset of BSI/VDM-SL is itself described using the complete version of BSI/VDM-SL[4]. However, expressions in BSI/VDM-SL cannot have side-effects and therefore, operations cannot be called inside expressions. We have for notational convenience chosen to allow calls of operations, which do not change the state, inside expressions because these operations do not cause side-effects anyway.

[4]In order to increase the readability of the operational semantics we have chosen to indicate the block structure by means of indentation instead of grouping statements together in blocks by means of brackets. We have been able to do this because we have used the LaTeX macros produced by Jan-Bert Oostenenk (see [Oostenenk90]).

In this section we first present the semantic domains, and then we explain the principles of the evaluation functions illustrated by means of a few examples taken from the full definition of the operational semantics for our executable subset.

4.1 Semantic Domains

The semantic domains describe the type of the structures which will be used for specifying the operational semantics for the abstract syntax.

$ENV_L = ENV^*$

The main structure in the semantic domains is the environment. The environment ENV_L is organized as a stack of function application environments ENV. When a function is called, it must establish a local environment containing its own definitions such as the formal parameters.

$ENV = BlkEnv^*$

Expressions can define a local environment called a block environment (*BlkEnv*). For example a let expression will produce a local environment for which the scope is the body of the let-expression. The function application environment is therefore organized as a stack of block environments where the first block environment pushed on the stack contains the instantiation of the formal parameters. When the value of an identifier is looked up this will happen in a top down manner down through the block environments.

$BlkEnv = IdVal^*$

A block environment is a sequence of *IdVal*'s, each containing an identifier and its associated value. A *BlkEnv* could alternatively have been modeled as a map. Modeling it as a sequence allows a more controlled error recovery as illustrated by the function *UnionMatch* which is presented in section 4.2.2. Naturally, some auxiliary functions have been defined to manipulate the environment (e.g. *PushBlkEnv* and *PopBlkEnv*). They are called by the evaluation functions.

When possible, the values in the semantic domain are specified in terms of the corresponding BSI/VDM-SL constructs, except that they are all tagged.

$VAL = BasicType \mid SET \mid \ldots$

$SET :: bd : VAL\text{-}\mathbf{set}$

These tags are used by the interpreter for dynamic type checking.

4.2 Evaluation Functions

The evaluation functions in this operational semantics of our executable subset differ from the evaluation functions for the dynamic semantics of the BSI/VDM-SL. In order to take looseness into account, the evaluation functions from the dynamic semantics of the BSI/VDM-SL return all possible results of evaluating a syntactic construct in a given

environment. In this work, an arbitrary one of the possible results will be returned when we deal with values. However, when we deal with pattern matching, all possible resulting binding environments are returned, and taken into account. This is necessary because there are situations (e.g. the let-be-such-that construct) where additional constraints are put on the matching afterwards. It is also important for quantified expressions where looseness in the pattern gives different binding environments which must all satisfy the predicate. Since the algorithm selecting an arbitrary value is deterministic, looseness of both functions and operations will be interpreted as underdeterminedness (i.e. if a function is loosely specified it will always return the same result given the same arguments). This will of course mean that two abstractly equal values (e.g. $\{1,2,3\}$ and $\{2,1,3\}$) are implemented so that they have the same concrete representation in the implementation.

All the evaluation functions use the state (and they ought therefore to be called operations in the BSI/VDM terminology). The state type in our operational semantics has as one of its components the stack of environments presented in the previous section:

$\Sigma :: \ldots$
$\quad env_l : ENV_L$

4.2.1 Expressions

The evaluation function for expressions uses the state, it takes a syntactic expression as argument and returns a semantic value. Thus, the signature of *EvalExpr* is:

$EvalExpr : Expr \xrightarrow{o} VAL$

where *VAL* is the type of an arbitrary value which the syntactic expression can evaluate to. The expressions contain patterns, and the patterns can be loosely specified. We will now describe the evaluation functions of a few expressions, starting with the let expression.

The semantics of the let expression is defined as:

$EvalLetExpr : LetExpr \xrightarrow{o} VAL$

$EvalLetExpr\,(mk\text{-}LetExpr(loc_m, in_e)) \triangleq$
dcl $pat_{lp} : Pattern^* := [\,]$,
$\quad val_{lv} : VAL^* := [\,]$;
for all $pat_p \in$ **dom** loc_m
do $val_{lv} := val_{lv} \frown [EvalExpr(loc_m(pat_p))]$;
$\quad pat_{lp} := pat_{lp} \frown [pat_p]$;
let $env_s = PatternListMatch(pat_{lp}, val_{lv})$ **in**
$\quad$ **if** $env_s \neq \{\}$
$\quad$ **then let** $env \in env_s$ **in**
$\qquad\qquad PushBlkEnv(env)$;
$\qquad\qquad$ **let** $in_v = EvalExpr(in_e)$ **in**
$\qquad\qquad\quad PopBlkEnv()$;
$\qquad\qquad\quad$ **return** in_v
$\quad$ **else error**

The local definitions from loc_m are collected in two sequences; val_{lv} contains a sequence of semantic values and pat_{lp} contains a sequence of syntactic patterns. All possible ways of matching the patterns against the values are collected in env_s which is a set of binding environments where every binding environment contains the environment for one possible matching. If the matching fails an empty set is returned and the evaluation of the whole let expression fails. Otherwise, an arbitrary one of these resulting binding environments is chosen, and pushed on top of the environment stack. Even though this description states that any binding environment can be chosen, the implementation will ensure underdeterminism by returning the same binding environment given the same set env_s. Then the body expression is evaluated in the new context and the environment is popped off the stack again, before the resulting value is returned.

The operational semantics presented here corresponds (functionally) to the dynamic semantics of BSI/VDM-SL except that only one value is returned instead of the set of all possible values. Furthermore, as mentioned in the section about the abstract syntax we have chosen not to include mutually recursive definitions.

The semantics of the let-be-such-that expression is defined as:

$EvalLetBeSTExpr : LetBeSTExpr \xrightarrow{o} VAL$

$EvalLetBeSTExpr\,(mk\text{-}LetBeSTExpr(bind_b, st_e, in_e)) \triangleq$
  dcl $env_s : BlkEnv\text{-}$**set** $:= \{\}$;
  for all $env \in EvalSetBind(bind_b)$
  do $PushBlkEnv(env)$;
    let $st_v = EvalExpr(st_e)$ **in**
      if $is\text{-}BOOL(st_v)$
      then let $mk\text{-}BOOL(b) = st_v$ **in**
                if b
                then $env_s := env_s \cup \{env\}$
      else error;
    $PopBlkEnv()$;
  if $env_s \neq \{\}$
  then let $env \in env_s$ **in**
          $PushBlkEnv(env)$;
          let $in_v = EvalExpr(in_e)$ **in**
            $PopBlkEnv()$;
            return in_v
  else error

For all possible binding environments which can be constructed from the set bind ($bind_b$) the "such that" expression (st_e) is evaluated. The binding environments in which the st_e evaluates to **true** are collected, and an arbitrary one is selected and pushed on top of the stack. The body expression is then evaluated in this context and the environment is popped off the stack again before the resulting value is returned.

The definition above corresponds closely to the dynamic semantics of the complete version of BSI/VDM-SL. The only difference is that here only an arbitrary one of the possible values are returned, while all possible values are collected and returned in [Larsen90].

The semantics of the quantified expressions ($\forall$ and $\exists$) is defined as:

$EvalAllOrExistsExpr : AllOrExistsExpr \xrightarrow{o} VAL$

$EvalAllOrExistsExpr\,(mk\text{-}AllOrExistsExpr(quant, bind_{sb}, pred_e)) \triangleq$
  dcl $env_s : BlkEnv$-**set**,
      $cont : \mathbf{B} := \mathbf{true}$;
  $env_s := EvalSetBindSet(bind_{sb})$;
  while $env_s \neq \{\,\} \wedge cont$
  do let $env \in env_s$ **in**
        $PushBlkEnv(env)$;
        let $pred_v = EvalExpr(pred_e)$ **in**
          if $is\text{-}BOOL(pred_v)$
          then let $mk\text{-}BOOL(b) = pred_v$ **in**
                cases *quant*:
                  ALL $\rightarrow cont := b$,
                  EXISTS $\rightarrow cont := \neg\, b$
                end
          else error;
        $env_s := env_s - \{env\}$;
        $PopBlkEnv()$;
   cases *quant*:
    ALL $\rightarrow$ **return** $mk\text{-}BOOL(cont)$,
    EXISTS $\rightarrow$ **return** $mk\text{-}BOOL(\neg\, cont)$
   end

For all possible binding environments which can be constructed from the set of bindings ($bind_{sb}$) the predicate expression ($pred_e$) is evaluated. It is then tested whether it is worth continuing (e.g. if it is a universal quantification we can leave the loop when we have found the first binding for which the predicate evaluates to **false**).

The dynamic semantics for BSI/VDM-SL is based on the three valued logic called LPF (Logic for Partial Functions) used in [Jones90]. This logic requires unbounded parallelism which naturally we cannot deal with when we are executing the specification. Thus, this operational semantics of the quantified expressions differs from the semantics of BSI/VDM-SL. The logic of the operational semantics can be considered as a conditional and/or between all the possible bindings. Thus, the two generalized forms of de Morgan's rule[5] still hold with our semantics, which we consider quite important.

4.2.2 Patterns

The general pattern matching operation *PatternMatch* uses the state and takes a syntactic pattern and a semantic value to match against the pattern as arguments, and returns the set of possible binding environments.

[5]The generalized forms of de Morgan's rule state that one of the quantifiers (universal or existential) over a predicate can be represented by negating the other quantifier with the negated predicate.

$PatternMatch : Pattern \times VAL \xrightarrow{o} BlkEnv\text{-}\mathbf{set}$

A binding environment is a block environment which instantiate the unbound variables (pattern identifiers) introduced in the pattern. If a pattern matching fails an empty set will be returned. *PatternMatch* examines the syntactic pattern and calls the corresponding matching operation.

As an example we present *MatchSetUnionPattern* which matches a set union pattern to a set value.

$$
\begin{array}{l}
MatchSetUnionPattern : SetUnionPattern \times VAL \xrightarrow{o} BklEnv\text{-}\mathbf{set} \\
MatchSetUnionPattern\,(mk\text{-}SetUnionPattern(lp_p, rp_p), val_v) \triangleq \\
\quad \mathbf{dcl}\ envres_{sl} : BlkEnv\text{-}\mathbf{set} := \{\}; \\
\quad \mathbf{if}\ is\text{-}SET(val_v) \\
\quad \mathbf{then\ let}\ mk\text{-}SET(val_{sv}) = val_v\ \mathbf{in} \\
\qquad\qquad \mathbf{for\ all}\ (setl_{sv}, setr_{sv}) \in SetChop(val_{sv}) \\
\qquad\qquad \mathbf{do\ let}\ envl_s = PatternMatch(lp_p, mk\text{-}SET(setl_{sv})\,), \\
\qquad\qquad\qquad\qquad envr_s = PatternMatch(rp_p, mk\text{-}SET(setr_{sv})\,)\ \mathbf{in} \\
\qquad\qquad\qquad\qquad \mathbf{if}\ envl_s \neq \{\} \wedge envr_s \neq \{\} \\
\qquad\qquad\qquad\qquad \mathbf{then}\ envres_{sl} := envres_{sl} \cup \\
\qquad\qquad\qquad\qquad\qquad\qquad UnionMatch\,(\{EnvMerge(tmp_1, tmp_2) \mid \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad tmp_1 \in envl_s, tmp_2 \in envr_s\})\,; \\
\qquad\qquad \mathbf{return}\ envres_{sl} \\
\quad \mathbf{else\ error}
\end{array}
$$

$$
\begin{array}{l}
SetChop : VAL\text{-}\mathbf{set} \to (VAL\text{-}\mathbf{set} \times VAL\text{-}\mathbf{set})\text{-}\mathbf{set} \\
SetChop\,(val_{sv}) \triangleq \\
\quad \{(setl_{sv}, setr_{sv}) \mid setl_{sv}, setr_{sv} \in \mathcal{F}\ val_{sv} \cdot \\
\qquad\qquad (setl_{sv} \cup setr_{sv} = val_{sv}) \wedge (setl_{sv} \cap setr_{sv} = \{\})\}
\end{array}
$$

All possible disjoint pair of sets are chopped from the set value val_{sv} by *SetChop*. For all possible set-pairs, the sets are matched against two patterns lp_p and rp_p. This generates two sets of binding environments which, if both matchings succeed, are combined to obtain the joint set of possible binding environments. Before adding this set to the result it is filtered by *UnionMatch* which is defined as follows:

$$
\begin{array}{l}
UnionMatch : BlkEnv\text{-}\mathbf{set} \xrightarrow{o} BlkEnv\text{-}\mathbf{set} \\
UnionMatch\,(blk_{sl}) \triangleq \\
\quad \mathbf{return}\ \{StripDoubles\,(blk_l) \mid blk_l \in blk_{sl} \cdot \\
\qquad\qquad\qquad\qquad \forall\,(id, v_{1v}) \in \mathbf{elems}\ blk_l, (id, v_{2v}) \in \mathbf{elems}\ blk_l \cdot v_{1v} = \\
v_{2v}\}
\end{array}
$$

UnionMatch filters a set of binding environments for inconsistency and redundancy. A binding environment is considered inconsistent if an identifier is associated with two different values. This situation can occur if repeated pattern identifiers have been used but

are matched to different values. This is not a valid binding environment and is therefore removed. In *UnionMatch* this is done by returning only binding environments where possible identical pattern identifiers are associated to identical values.

If a pattern matching using repeated pattern identifiers succeeds (is consistent) it will result in duplicate entries in the binding environment which is redundant information. *StripDoubles* therefore removes all duplicate entries.

5 Examples

In this section we will present a few examples of patterns using loose specification and repeated pattern identifiers illustrating the expressiveness provided by the generality of the patterns in our subset.

To illustrate the use of loose specification we present a version of the *MergeSort* algorithm.

$$
\begin{array}{l}
\mathit{MergeSort} : \mathbf{N\text{-}set} \to \mathbf{N}^* \\
\mathit{MergeSort}\,(\mathit{set}) \triangleq \\
\quad \textbf{cases}\ \mathit{set} : \\
\qquad \{\,\} \to [\,], \\
\qquad \{e\} \to [e], \\
\qquad \mathit{set}_1 \cup \mathit{set}_2 \to \mathit{Merge}(\mathit{MergeSort}(\mathit{set}_1), \mathit{MergeSort}(\mathit{set}_2)) \\
\quad \textbf{end}
\end{array}
$$

$$
\begin{array}{l}
\mathit{Merge} : \mathbf{N}^* \times \mathbf{N}^* \to \mathbf{N}^* \\
\mathit{Merge}\,(\mathit{seq}_l, \mathit{seq}_r) \triangleq \\
\quad \textbf{cases}\ (\mathit{seq}_l, \mathit{seq}_r) : \\
\qquad ([\,], \mathit{seq}) \to \mathit{seq}, \\
\qquad (\mathit{seq}, [\,]) \to \mathit{seq}, \\
\qquad ([e_l] \frown \mathit{seq}'_l, [e_r] \frown \mathit{seq}'_r) \to \textbf{if}\ e_l < e_r \\
\qquad\qquad\qquad\qquad \textbf{then}\ [e_l] \frown \mathit{Merge}(\mathit{seq}'_l, \mathit{seq}_r) \\
\qquad\qquad\qquad\qquad \textbf{else}\ [e_r] \frown \mathit{Merge}(\mathit{seq}_l, \mathit{seq}'_r) \\
\quad \textbf{end}
\end{array}
$$

MergeSort takes a set of natural numbers and returns a sorted list of the elements in the set. If *set* contains more than one element it is split up, using the loosely specified union match, into two disjoint sets set_1 and set_2. These sets are sorted recursively and the results are merged by *Merge*.

If we wanted to ensure that set_1 and set_2 are of approximately the same cardinality, this could be done by the let-be-such-that expression which is also included in our subset. The last entry in the case expression in *MergeSort* should then be replaced with:

$$
\begin{array}{l}
s \to \textbf{let}\ \mathit{set}_1 \cup \mathit{set}_2 \in \{s\}\ \textbf{be st abs}\ (\textbf{card}\ \mathit{set}_1 - \textbf{card}\ \mathit{set}_2) \leq 1\ \textbf{in} \\
\qquad\qquad \mathit{Merge}(\mathit{MergeSort}(\mathit{set}_1), \mathit{MergeSort}(\mathit{set}_2))
\end{array}
$$

The let-be-such-that expression is still loosely specified as the pattern matching of the set union against *set* (the only element in the bind set $\{set\}$) can result in a number of possible binding environments. The difference from the original solution is the additional constraint (the such-that predicate) which is placed on the possible binding environments.

To illustrate the use of repeated pattern identifiers, we have specified an invariant for the type *PointSet*.

$PointSet = (X \times Y)$**-set**

inv $ps \triangleq \forall (x, y_1) \in ps, (x, y_2) \in ps \cdot y_1 = y_2$

This shows how the use of repeated pattern identifiers can describe, in a compact form, the condition on a set of type *PointSet*, that points which have the same the first coordinate must also have the same second coordinate.

6 Related Work

There exist a number of other executable languages which have been more or less inspired by VDM. In [Plat&89] an overview of existing tool support for VDM is presented. However, only two projects dealing with an executable subset of VDM-SL are given in that overview. These are the Meta-IV compiler project from Kiel University (see [Haß87]), and the EPROS project where both an interpreter and a compiler for a language called EPROL (strongly inspired by VDM) have been developed (see [Hekmatpour&88]). In addition to these two we have looked at 'me too' (see [Alexander&90]). The main difference between our executable language and the existing ones is the generality of the pattern matching.

None of the existing executable languages inspired by VDM which have been mentioned above contains more than pattern identifiers, tuple patterns, and record patterns. However, a number of languages for functional programming exist which also support sequence patterns. In addition, the pattern matching in all the existing languages is deterministic. However, it is clear that by taking an approach where we deal with loosely specified patterns we will lose some efficiency. On the other hand, the generality of the patterns we are using provides the user with much more flexibility in writing the (executable) specifications. For instance it would not be possible to write any of the examples from the previous section in any of the other executable languages inspired by VDM.

7 Concluding Remarks

We have defined an executable subset of a VDM dialect which is more powerful than any of the existing ones we are aware of. Even though the syntax has been selected specially to provide the necessary expressive power for the mini-specifications in IPTES, we hope to be able to implement it so that it can be used outside the IPTES environment as well. We hope to be able to do this because we feel that such an interpreter will be useful for other applications as well.

We also consider it a strength of the subset that it is as closely related to the forthcoming standard language (both BSI and ISO VDM-SL) as it is. Thus, the users of this interpreter will have plenty of existing literature at their disposal, because a lot of publications are expected to be using the standard notation.

We still have to finish the actual implementation of the interpreter according to the operational semantics presented here. The current state of the project (primo August 1991) is that we have finished the specification of the syntax and semantics. A parser which constructs abstract syntax trees have also been implemented and we are in the process of implementing C++ classes for the most common VDM-SL types. We hope that these classes and the operational nature of the semantics will allow us to design an interpreter which structure is closely related to the definition of the semantics, thereby achieving a short and reliable design in implementation phase. It is an open question how efficient our interpreter will be, due to the generality we have included. However, we expect to know much more about the efficiency at the time of the VDM symposium as the first version of the interpreter is planned to be completed in December 1991.

Acknowledgments

We would like to thank John Dawes, Stephen Bear, Hans Toetenel and especially Nico Plat and Jan Storbank Pedersen for valuable remarks on an earlier version of this article. In addition we have had constructive remarks from our colleagues at IFAD.

The IPTES consortium is formed by IFAD (Denmark), VTT (Finland), MARI (United Kingdom), CEA/LETI (France), ENEA (Italy), Synergie (France), DIT/UPM (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.

[Bjørner91] Dines Bjørner. *Software Architectures and Programming Systems Design.* Submitted to publisher.

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

[Ghezzi&91] Carlo Ghezzi, Dino Mandrioli, Sandro Morasca and Mauro Pezzé. A Unified High-Level Petri Net Model For Time-Critical Systems. *IEEE Transactions on Software Engineering*, 17(2), 1991.

[Haß87] Manfred Haß. Development and Application of a Meta IV Compiler. In *VDM – A Formal Method at Work*, Springer–Verlag, 1987.

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

[Jones90] Cliff B. Jones. *Systematic Software Development Using VDM (second edition).* Prentice Hall, 1990.

[Larsen&89] Peter Gorm Larsen, Michael Meincke Arentoft, Brian Monahan and Stephen Bear. Towards a Formal Semantics of The BSI/VDM Specification Language. In *Information Processing 89*, North-Holland, 1989.

[Larsen90] Peter Gorm Larsen. *The Dynamic Semantics of the BSI/VDM Specification Language.* Technical Report, August 1990.

[Leon&91] Gonzalo León, J.A. de la Puente, M.A. Ruz and E.W. Sink. *Definition of the IPTES Architecture.* Technical Report, May 1991.
IPTES Doc.id : IPTES-UPM-1-V2.3.

[Oostenenk90] Jan-Bert Oostenenk. *Typesetting VDM with VDMSL macros.* Technical Report, NPL, 1990.

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

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

[Wieth89] Morten Wieth. Loose Specification and its Semantics. In *Information Processing 89*, North-Holland, 1989.