A Proof Obligation Generator for VDM-SL

Bernhard K. Aichernig[1] and Peter Gorm Larsen[2]

[1] Graz University of Technology, Institute of Software Technology (IST), Münzgrabenstr. 11/II, 8010 Graz, Austria
[2] Institute of Applied Computer Science (IFAD), Forskerparken 10, 5230 Odense M, Denmark

Abstract. In this paper an extension of the IFAD VDM-SL Toolbox with a proof obligation generator is described. Static type checking in VDM is undecidable in general and therefore the type checker must be incomplete. Hence, for the "difficult" parts introducing undecidability, it is up to the user to verify the consistency of a specification. Instead of providing error messages and warnings, the approach of generating proof obligations for the consistency of VDM-SL specifications is taken. The overall goal of this work is to automate the generation of proof obligations for VDM-SL. Proof obligation generation has already been carried out for a number of related notations, but VDM-SL contains a number of challenging constructs (e.g. patterns, non-disjoint union types, and operations) for which new research is presented in this paper.

1 Introduction

During the last few years the interest in formal software development has been growing rapidly. One of the main reasons for this is the availability of tools to support the developer in using these formal methods. This paper describes an extension of the IFAD VDM-SL Toolbox [13, 24], a commercial CASE tool supporting the Vienna Development Method (VDM). Amongst other features the Toolbox provides parsing and type checking of specifications written in VDM-SL, the specification language of VDM, which has been standardised under ISO [23, 26].

VDM is one of the most widely used formal methods, and it can be applied to the construction of a large variety of software systems. It is a *model-oriented* method, i.e. its formal descriptions (VDM specifications) consist of an explicit model of the system being constructed. A system design is generated through a series of specifications, where each specification is more concrete and closer to the implementation than the previous one [21]. Each of these development steps introduces a formal refinement statement which, when appropriately verified, ensures the (relative) correctness of the implemented system.

The existing type checker of the IFAD VDM-SL Toolbox supports extensive consistency checks according to the static semantics of the ISO standard. However, the problem of static type checking VDM-SL specifications is that it is undecidable in general, which means that the consistency checks must be incomplete. Thus, it is up to the user to check the "difficult parts" in a specification, which causes the undecidability. The existing type checker generates error messages and warnings to identify places with possible inconsistencies. In VDM-SL the undecidability is introduced by partial operators, functions with pre-conditions, union types and subtypes. Division is one of the most well-known examples of partial operators. Considering the expression a/b, the requirement for the expression to be consistent is that b is not equal to zero. In general, this cannot be guaranteed by static type checking, which means the expression may or may not be consistent.

Instead of generating error messages, this work improves the existing type checker by generating proof obligations for those parts of a specification which cause the undecidability and cannot be checked automatically. These proof obligations (PO) are unproved theorems stating that a certain condition must hold in order to ensure that a specification is consistent in itself. For a/b the condition for consistency is $b \neq 0$. If all POs generated for a specification can be proved, then the specification is consistent. Therefore, the POs are designed to be loaded into the proof tool of the Toolbox, which is currently under development at IFAD [3, 2]. However, many of the generated proof obligations are trivial or simple, and can be informally justified by simply inspecting the PO.

The notion of POs for VDM has been based on previous work [21, 20, 6]. The main contribution of this work is the automation of the generation of POs. In addition, Sections 5 and 6 present new work in the areas of patterns and explicit operations where no existing research for POs was present.

The proof obligation generator (POG) is an extension of the existing specification of the static semantics and has been formally specified using an executable subset of VDM-SL. To distinguish between the meta and the object level in this paper, the specification parts of the POG are pretty printed using the mathematical concrete syntax of VDM-SL, and the used examples with their generated POs are printed in the ASCII VDM-SL notation.

After this introduction an overview of the existing type checker is provided. This is followed by a series of sections presenting the proof obligation generator. In this technical part of the paper the approach dealing with the unique features of VDM-SL are explained and a little familiarity with VDM-SL is assumed. In Section 7 the relation to existing work is given and finally Section 8 contains some concluding remarks and identification of possible directions for future work.

2 The Existing Type Checker

This section is an overview of the existing type checker focusing on the parts which are essential for the POG. The specification document of the existing type checker contains more than 250 pages organised in 12 different modules. In addition there is a test environment with a large number of test cases used for regression testing. Due to the fact that the IFAD VDM-SL Toolbox is a commercial product, the specification document is confidential. Thus, we are only able to show very small extracts from this specification here.

2.1 Rejection and Acceptance

Type checking for VDM-SL is somewhat different from what is usually found in programming languages [7, 10]. According to the standard of VDM-SL [23] a dual strategy of type checking is applied: (1) Impossibly consistent specifications are rejected. For example, the type checker will raise an error message if a specification contains the expression $1 + \mathit{true}$. This kind of rejection approach is usually used for traditional programming languages. (2) Definitely consistent specifications are accepted. For example, $1 + 2$ is definitely consistent.

However, for many VDM-SL constructs it cannot be decided if they are definitely consistent, although they are possibly consistent. These are the "difficult parts" introducing the undecidability. For example, the expression a/b is possible consistent if a and b are of a numeric type, but it cannot be statically checked that b is unequal to zero. In fact, only some very simple specifications are accepted by the existing type checker to be definitely consistent.

According to the dual strategy in the static semantics the type checker can be used in two different modes: (1) in possible mode (POS) error messages indicate not possibly consistent parts; and (2) in definite mode the absence of error messages indicates the acceptance of a specification. In the IFAD VDM-SL Toolbox the two modes of type checking are selected by setting an option [16].

As mentioned in the introduction not only partial operators cause undecidability, but also the rich type system of VDM-SL. This is mainly due to the fact that the union type is an ordinary set-theoretic union without injection and projection functions. In addition subtypes are described by type invariants restricting the "legal" values of a type in a set-theoretic manner without any requirement for explicitly "casting" the values to be members of the subtype. In general such invariants can be arbitrarily complex and thus it is impossible to statically determine whether an expression belongs to a required subtype.

The following algorithm for checking the compatibility of two types reflects the dual strategy of type checking in VDM-SL.

2.2 Type Compatibility

The central task of the type checker is to check if two types, an expected and an actual, are compatible. Due to the rich type system of VDM-SL a simple comparison is not possible in general. Furthermore, the task is statically undecidable in general, because of non-disjoint unions and subtypes [10].

The operation[3] performing the compatibility check *IsCompatible* takes three arguments. The first argument specifies the kind of type checking that should be performed (POS or DEF). The next two arguments ($\mathit{TpR}1$, $\mathit{TpR}2$) are the types that must be checked. They are of type *TypeRep*, the internal type representation in the type checker.

[3] An operation is used, because the auxiliary operations *IsOverlapping* and *IsSubType* refer to the environment state containing type names and their type binding.

```
$IsCompatible : (\mathrm{POS} \mid \mathrm{DEF}) \times TypeRep \times TypeRep \xrightarrow{o} \mathbb{B}$
$IsCompatible\,(i, TpR1, TpR2) \triangleq$
  if $TpR1 = TpR2$
  then return true
  else cases $i$:
         $\mathrm{POS} \to IsOverlapping(TpR1, TpR2)$,
         $\mathrm{DEF} \to IsSubType(TpR1, TpR2)$
     end;
```

In the trivial case that the two types are the same, true is returned. Otherwise, depending on the kind of type-checking (possible or definite), the function checks whether the types are overlapping or whether one is a subtype of the other.

Two types are possibly compatible if they are overlapping, which means that the sets of values they denote are non-disjoint. In VDM-SL two different basic types are overlapping if both of them are numeric types. If two types are not overlapping, they cannot possibly be consistent (rejection).

In definite mode all values denoted by an actual type must be compatible to an expected type. Therefore, the algorithm has to check that the set of values denoted by the actual type is a subset of the set of values denoted by the expected type. In short, the actual type must be a subtype of the expected type. For example, every numeric type is a subtype of the type `real` denoting the real numbers.

2.3 Well-formedness of Expressions

According to the strategy of rejection and acceptance, the static semantics defines two kinds of predicates in order to check whether an expression is well-formed. These predicates denote the possible and definite well-formedness of an expression [10, 23].

If the existing type checker of the IFAD VDM-SL Toolbox performs possible type checking (POS mode) an expression is rejected if it is not possibly well-formed. In definite mode (DEF) the type checker only accepts expressions if they are definitely well-formed, which means that an error message is raised if the definite well-formedness cannot be statically checked. As an example the specification of the well-formedness predicate for a division expression is given below:

```
$wf\text{-}NUMDIV : (\mathrm{POS} \mid \mathrm{DEF}) \times BinaryExpr \xrightarrow{o} \mathbb{B} \times TypeRep$
$wf\text{-}NUMDIV\,(i, \textsf{mk-}BinaryExpr\,(lhs, \text{-}, rhs)) \triangleq$
  let mk- $(wf\text{-}lhs, lhstype) = wf\text{-}Expr\,(i, lhs)$,
      mk- $(wf\text{-}rhs, rhstype) = wf\text{-}Expr\,(i, rhs)$,
      $ExpectedLhsAndRhsType = \textsf{mk-}BasicTypeRep\,(\mathrm{REAL})$ in
  let $lhscomp = IsCompatible\,(i, lhstype, ExpectedLhsAndRhsType)$,
        $rhscomp = IsCompatible\,(i, rhstype, ExpectedLhsAndRhsType)$ in
    (if $\neg$ $lhscomp$
     then $GenErr("Lhs\ of\ '/'\ is\ not\ a\ numeric\ type")$ ;
     if $\neg$ $rhscomp$
     then $GenErr("Rhs\ of\ '/'\ is\ not\ a\ numeric\ type")$
     elseif $i = \mathrm{DEF} \wedge rhstype \neq$ mk-$BasicTypeRep\,(\mathrm{NATONE})$
     then ($GenErr("Rhs\ of\ '/'\ must\ be\ non\ zero")$ ;
           return mk- (false, $ExpectedLhsAndRhsType$) );
     return mk- ($wf\text{-}lhs \wedge wf\text{-}rhs \wedge lhscomp \wedge rhscomp$,
                 $ExpectedLhsAndRhsType$) )
```

The choice of possible and definite well-formedness is conveyed to the well-formedness operation via the first parameter i. The second parameter is the expression to be checked, which in this case is a binary expression with the binary operator '/'. The signature shows that in addition to the well-formedness of subexpressions, the type of the expression is returned. Analysing the body, four parts can be distinguished: (1) The well-formedness and the types of the left- and right-hand side (*lhs* and *rhs*) are determined. (2) The check whether the left- and right-hand side are compatible to the expected type. (3) Error Messages are raised. Note the additional error message in definite mode: To be accepted, *rhs* must be of type $\mathbb{N}_1$, which denotes the natural numbers excluding zero. This type is needed for static acceptance, because no other basic numeric type in VDM-SL excludes zero. (4) The well-formedness and type of the division expression are returned. It is the goal of this work to generate proof obligations instead of error messages in definite mode. In the following section this modification of the type checker is explained.

3 The Proof Obligation Generator

The existing type checker specification document was extended with one extra module and the total size became approximately 300 pages [4]. The test cases and the test environment were naturally also adapted to take these changes into account.

The central idea underlying the approach presented here (proposed in [10, 11]) is to extend type checking by generating proof obligations for the "difficult" parts which cause the undecidability. In this section we will describe how the existing type checker has been extended and modified to perform this task. The proof obligations (PO) should replace the error messages, raised by the definite type checker.

Fig. 1 provides an overview of the data flow to and from the type checker. A VDM-SL specification is syntax checked by the parser which produces an abstract syntax tree of the specification. The abstract syntax tree is the input for the type checker (static semantics). If type checking in possible mode fails, a list of error messages is generated. In definite mode the type checker becomes the proof obligation generator. These proof obligations can be inspected through the graphical user interface (GUI) and/or they can be loaded into the proof tool.

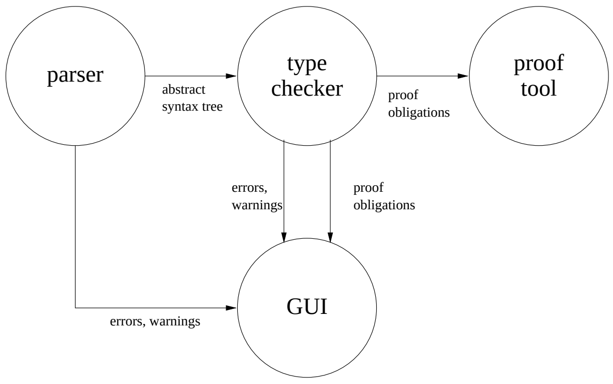

Fig. 1. Interaction of the new type checker.

The main advantage of this approach is the combination of a type checker and a proof tool. In the cases where the type checker is not able to check the consistency, the proof theory in the proof tool can be applied to prove the consistency. The link between the two components are the generated proof obligations. Minor simplifications and proofs of trivial proof obligations could be made directly by the POG. However, in the present approach the decision has been made to forward these tasks to the proof tool, which is designed for these purposes.

3.1 Proof Obligations

A proof obligation for type consistency is a statement that must be proved in order to ensure type consistency. Therefore, the POG is a conditional type checker which returns true under the assumption that the POs generated during the type check can be proved. This process is called conditional type checking. In order to be able to prove POs, they contain context information. The general form of a proof obligation is that of a proof rule, where the assumptions are generated out of the context and the conclusion is the property that must hold. Thus, the general form of a PO is

$$\frac{\text{context information}}{\text{predicate}}$$

In the examples below, the proof obligations will be given in an ASCII representation which is:

```
PO: context information ==> predicate
```

A proof obligation is modelled as a proof (inference) rule consisting of a list of hypotheses and a conclusion. The rule states that the conclusion holds whenever the hypotheses hold [6, 20]. The abstract syntax for a proof rule is:

$$\begin{aligned} \mathit{PrfRule} :: \mathit{hyp} &: \mathit{Sequent}^* \\ \mathit{con} &: \mathit{AllExpr}; \end{aligned}$$

To be compatible with the proof theory a hypothesis is defined as a sequent $P \vdash R$. Its meaning is that the local conclusion R is derivable from the local assumptions P in a subproof. Formally, a sequent is composed of local variables, a sequence of local assumptions, and a local conclusion:

$$\begin{aligned} \mathit{Sequent} :: \mathit{vars} &: \mathit{Name}^* \\ \mathit{lhyp} &: \mathit{AllExpr}^* \\ \mathit{lcon} &: \mathit{AllExpr}; \end{aligned}$$

However, it turned out that for formulating the proof obligations the use of local assumptions is not needed. Therefore, the hypotheses are sequents without local assumptions ($\vdash R$). In the following, we skip the turnstile in the hypotheses, when we present a proof obligation.

AllExpr is a syntactic extension of VDM-SL expressions with type judgements (e.g., $3 : \mathbb{N}$). This extension is necessary in order to reason over types.

$$\mathit{AllExpr} = \mathit{Expr} \mid \mathit{TypeJudgement};$$

$$\begin{aligned} \mathit{TypeJudgement} :: \mathit{expr} &: \mathit{Expr} \\ \mathit{type} &: \mathit{TypeRep} \end{aligned}$$

3.2 Context Generation

Context information forms the hypothesis of a proof obligation. Therefore, the existing specification of the type checker has been extended in order to generate the context. The context of a certain location in a specification is the summation of logical conditions that must hold in order to reach this location during evaluation. The following example illustrates the notion of context:

```
if is-nat(x) then x + 1 else 1
```

To reach `x + 1` during the evaluation, the condition `is-nat(x)` must be true. Therefore, the context of `x + 1` is `is-nat(x)`. Obviously, the context of `1` in the else branch is `not is-nat(x)`.

For generating proof obligations we distinguish between two kinds of context:

Type judgements of the form *identifier* : *Type* are used to record membership of a type.This may be read as "*identifier* is of type *Type*". They are provided as context information to infer over types in the typed logic of partial functions and to provide the proof tool with the needed type information. The type information is stored in the environment of the type checker.

Boolean expressions like `is-nat(x)` in our example above are stored in a context stack, while performing the consistency check. When checking the then-branch of an if-expression the test expression (above: `is-nat(x)`) is pushed onto the stack. Checking the else branch, we first remove the test expression from the stack (pop), then the negated test expression (above: `not is-nat(x)`) is pushed onto the stack.

4 Proof Obligations for Expressions and Functions

When type checking VDM-SL expressions, two categories of consistency checks can be distinguished: (1) Type compatibility and (2) domain checking, which checks whether functions and operators are applied to their defined domains. Both may be undecidable. The first because of union and subtypes, the second because of partial operators and functions with pre-conditions. In addition to these two categories the new POG also generates satisfiability obligations for implicitly defined functions and operations [21].

4.1 Type Compatibility

A compatibility check fails if the actual type and an expected type are overlapping, but the first is not a subtype of the second. This undecidable case occurs if the actual type is a union type or if the expected type is a subtype of the actual type.

Consider the function `f`, which increases its argument if it is a natural number, otherwise 1 is returned.

```
f: bool | nat -> N1
f(x) == if is-nat(x) then x + 1 else 1
```

The returned value is restricted by an invariant to the natural numbers unequal to zero: `N1 = nat inv n == n <> 0`. The function is consistent, which means it will not cause a run-time error. However, the existing type checker fails to accept it, because: (1) the plus operator in `x + 1` expects numeric types and therefore `x` must be compatible to (here: a subtype of) `real`. However, the actual type of `x` is the union type `bool | nat` which only overlaps the type `real`, but is not a subtype of it. (2) To check the consistency of the return value the type checker would have to statically determine, whether the invariant function is true or false, which is is not possible.

Therefore the extended type checker raises two proof obligations stating the needed properties that in cases: (1) `x` is of type `real` and (2) the invariant holds:

```
PO1: is-nat(x)  ==>  x:real
PO2: x:bool | nat  ==> inv-N1( if is-nat(x) then x + 1 else 1 )
```

`PO1` trivially states that if `x` is a natural number then it is a real number. Note that this theorem is only valid for VDM-SL, with non-disjoint types. This PO can be proved automatically by the proof tool. `PO2` states that for the function body the implicitly defined invariant function for `N1` holds.

4.2 Domain Checking

For domain checking, proof obligations are generated for partial operators and for function applications if the function has a pre-condition. These POs ensure that the operands/parameters are in the defined domain. Examples of partial operators are division, head and tail. A pre-condition in a function restricts the domain by an additional predicate. The use of such a function generates a proof obligation stating that the pre-condition function holds. Below the general POs for `a / b` and `g(a)`, where g is defined with a pre-condition, are given:

```
PO1: context ==> b <> 0
PO2: context ==> pre-g(a)
```

Below we illustrate how such domain checking is carried out for partial operators. Again, the well-formedness operation of a division expression serves as the demonstrating part of the specification to illustrate the changes made for the POG. To see the modifications compare the following with the specification in Section 2.3:

```
$wf\text{-}NUMDIV\text{-}POG : (\mathrm{POS} \mid \mathrm{DEF}) \times BinaryExpr \overset{o}{\rightarrow} \mathbb{B} \times TypeRep$

$wf\text{-}NUMDIV\text{-}POG\,(i, \mathsf{mk\text{-}}BinaryExpr\,(lhs, \text{-}, rhs)) \triangleq$
  let mk- $(wf\text{-}lhs, lhstype) = wf\text{-}Expr\,(i, lhs)$,
      mk- $(wf\text{-}rhs, rhstype) = wf\text{-}Expr\,(i, rhs)$,
      $ExpectedLhsAndRhsType = \mathsf{mk\text{-}}BasicTypeRep\,(\mathrm{REAL})$ in
  let $lhscomp = IsCompatible\,(i, lhstype, ExpectedLhsAndRhsType)$,
      $rhscomp = IsCompatible\,(i, rhstype, ExpectedLhsAndRhsType)$ in
  (if $\neg\, lhscomp$
   then if $i = \mathrm{DEF}$
        then $GenPO(\mathsf{mk\text{-}}TypeJudgement\,(lhs, ExpectedLhsAndRhsType))$
        else $GenErr(\text{"}Lhs\ of\ '/'\ is\ not\ a\ numeric\ type\text{"})$ ;
   if $\neg\, rhscomp$
   then if $i = \mathrm{DEF}$
        then $GenPO(\mathsf{mk\text{-}}TypeJudgement\,(rhs, ExpectedLhsAndRhsType))$
        else $GenErr(\text{"}Rhs\ of\ '/'\ is\ not\ a\ numeric\ type\text{"})$ ;
   if $i = \mathrm{DEF} \wedge rhstp \neq \mathsf{mk\text{-}}REP`BasicTypeRep\,(\mathrm{NATONE})$
   then $GenPO(\mathsf{mk\text{-}}BinaryExpr\,(rhs, \mathrm{NE}, \mathsf{mk\text{-}}RealLit\,(0), \mathsf{nil}\,))$ ;
   if $i = \mathrm{DEF}$
   then return mk- $(wf\text{-}lhs \wedge wf\text{-}rhs, ExpectedLhsAndRhsType)$
   else return mk- $(wf\text{-}lhs \wedge wf\text{-}rhs \wedge lhscomp \wedge rhscomp,$
                    $ExpectedLhsAndRhsType)$ )
```

The main difference compared to the previous specification is the generation of proof obligations in DEF mode, instead of error messages: If the compatibility check is not successful a PO is generated (*GenPO*) stating that the expression (*rhs* or *lhs*) is of the expected type. This is done using a type judgement. For domain checking a PO stating that the right-hand side is not equal (NE) to zero is generated. In definite mode only the well-formedness of the left- and right-hand side is returned, because the compatibility and the domain constraint are assumed to be true. This assumption is verified later using the proof tool by proving the generated proof obligations. The operation *GenPO* takes the conclusion of the PO as an argument. During type checking the context is collected from the abstract syntax tree and is stored separately and updated in a context stack. This context together with the conclusion argument is used by *GenPO* to synthesise a proof obligation.

4.3 Satisfiability

Satisfiability obligations state that it must be possible to find a model for implicitly defined constructs such that for all valid input a valid output must exist. The satisfiability of an implicit function is thus another VDM-SL concept that is covered by the POG. Implicit functions are defined by means of pre- and post-conditions. A post-condition is a truth valued expression which specifies what must hold after the function is evaluated. A post-condition can refer to the result identifier and the parameter values. Consider an implicit specification of the division:

```
Div( a:real, b:real) r:real
pre b <> 0
post a = r * b
```

The pre-condition restricts the domain of `div`. The post-condition states a relation between the operands `a, b` and the result `r`. For an implicit function definition to be consistent, there must exist a return value satisfying the post-condition when the pre-condition is satisfied. This property is called the satisfiability of implicit functions. Below the satisfiability PO for `Div` is given:

```
PO: b <> 0 ==> exists r:real &  a = r * b
```

As in explicit functions the pre-condition is taken as the context. A similar strategy is used for implicitly defined operations.

5 Pattern Matching

Pattern matching is one of the advanced features of VDM-SL. All constructs containing patterns are powerful tools in specifying. However, patterns raise some difficulties in type checking that are worth investigating. The use of patterns depends on the expressions containing them. In this section let and cases expressions will serve as the demonstrating example constructs.

A pattern is a template for a value of a particular class of type. It is always used in a context where it is matched to a value of a certain type. Consider the set enumeration pattern {`a, 2`}. This pattern matches only set values with two elements, where one of the elements is `2`. The identifier `a` is called a pattern identifier and matches any value, of any type. The match value `2` can only be matched against the value itself. Possible values matching the set enumeration pattern are {1, 2} or {true, 2}. A full description of all different kinds of patterns can be found in [12, 31].

In general, type checking patterns can be seen as checking if the patterns can match associated values. If they cannot, the specification has to be rejected. A pattern identifier, for example, is compatible to all types, but a set enumeration pattern expects a set type. This corresponds to type checking for possible consistency (POS mode). However, definite consistency demands more advanced checks, which depend on the usage of patterns. In the following it will be demonstrated that these definite checks are often statically undecidable and how the extended type checker overcomes this problem by generating proof obligations.

5.1 Let expressions

In let expressions patterns are used to improve the readability of complicated expressions or to decompose complex structures. The general form of a simple let expression is let *pattern* = *value* in *expr*. We consider the following example in order to investigate the behaviour of the type checker:

```
let {a,b} = if true then {1,2} else 0 in a + b
```

In practice the example would be useless and of poor quality, because the else branch is in fact “dead code”. However, the expression is consistent and serves to look into the edges of type checking pattern expressions. The expression evaluates to 3. The if expression is of the union type `set of nat | nat`. As explained, the set enumeration pattern matches only with set values. In possible mode the type checker checks if the pattern possibly matches the value, i.e. if the set of values the pattern denotes overlaps the values denoted by the expression. In the example above this is the case. To be definite consistent the type checker has to guarantee the match. This is not possible, without evaluating the expression. A check of this kind is undecidable in general and therefore a proof obligation stating that the pattern matches the expression/value is needed. Using an existential quantification we are able to formulate a predicate “*pattern* matches *value*”. The match proof obligation for the example above looks like:

```
PO: exists a, b:nat & {a,b} = if true then {1,2} else 0
```

Verifying this PO ensures formally that the pattern matches the value. During the process of generating a proof obligation, the pattern {a,b} is turned into an expression. Note that for patterns containing a match value in any case a proof obligation has to be generated to ensure the definite consistency. The following example makes this obvious:

```
let {a,2} = {1,2} in a
```

The let expression above evaluates to 1. The match value 2 restricts the matching property: The pattern matches only sets of two elements where 2 is one of the elements. This is not decidable in general. Therefore the PO

```
PO: exists a:nat & {a,2} = {1,2}
```

is generated.

5.2 Cases Expressions

The use of patterns in cases expressions is very different from what was presented above. Cases expressions allow the choice of one from a number of expressions on the basis of the value of a particular expression. Consider the following cases expression:

```
let a in set {1, 2, {1,0}, {2,0}} in
cases a:
   1,
   {1,b}        ->  DoThis(1),
   2,
   {2} union b  ->  DoThat(2),
   others       ->  0
end
```

The possible values of `a` are defined by a let be-such-that expression. Here patterns are used to define the branches of a cases expression. The expression which is going to be evaluated is selected by pattern matching, where several patterns can be used to define one branch.

Cases expressions need special treatment in definite type checking. The existing type checker had to be improved, because errors were returned in almost all cases. Type checking the example above produced several error messages "Pattern cannot match". However, the expression is in fact type consistent. The reason for the overly strict behaviour of the type checker was, like for let expressions above, that every pattern was checked for its well-formedness in definite mode.

However, the semantics of pattern matching used this way is different from that in let-expressions. Type checking a pattern in definite mode assures that it will match definitely. This approach is not feasible in a cases expression, where pattern matching serves to select, not to define a value. The patterns in the cases expression above have to be able to match to `a`, which means that the sets of values they denote must overlap the value domain of the expression `a`. Therefore, patterns in cases expressions only need to be possibly well-formed (consistent). If the possible well-formedness check fails for a pattern, this pattern can never match and the corresponding expression will become "dead code" (holds for a single pattern). Figure 2 shows the type relations for cases expressions. The grey areas indicate the set of values to which the patterns in the cases expression match. The white area ($expr.a$) is the set of values denoted by the type of a. The different possibilities for the pattern types are: (1) The pattern is overlapping $expr.\ a$ ($p1$ to $p3$). (2) The pattern does not overlap $expr.a$ ($p4$) and therefore cannot match. (3) The pattern is a subtype of $expr.a$ ($p5$). (4) The type of $expr.a$ is a subtype of the pattern. (5) A pattern identifier, which matches with every value, occurs. It covers the whole type universe of VDM-SL. (6) The `others` alternative is present. It covers the white area of the domain of $expr.a$, which is not covered (overlapped) by the alternative patterns.

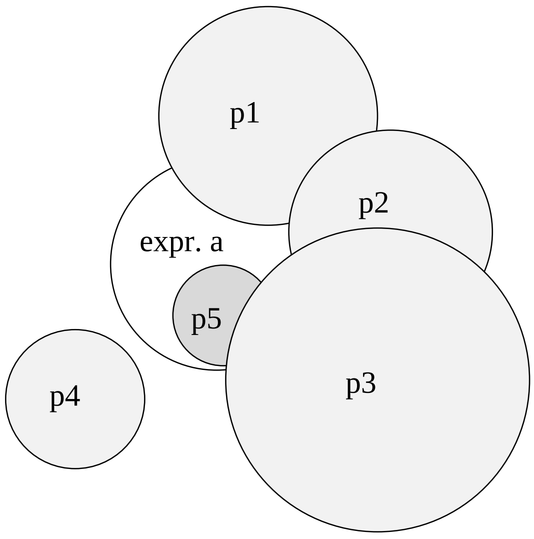

Fig. 2. The sets of values which match the patterns in a cases expression.

In VDM-SL the `others` alternative is not mandatory, if the expression to which the alternative patterns are matched (*expr.a*) is fully covered (overlapped) by these patterns. Thus, the `others` construct can be skipped, if one of the alternatives will match the expression definitely. In the example above the others construct can be omitted, because the other alternatives fully cover the possible values. To ensure the coverage of the expression by the patterns in a cases expression without an others branch or a pattern identifier a proof obligation has to be generated. The coverage PO for the above cases expression without an others construct would be:

```
PO:  a in set {1, 2, {1,0}, {2,0}}
     ==> exists b: nat | set of nat &
           a in set {1,2}  or
           {1,b} = a       or
           {2} union b = a
```

The proof obligation states that for all `a` in the set, a match must be possible. The match values are summarised in a value set. The provability of this proof obligation shows the definite consistency of the example even without the `others` branch.

6 Operations

In VDM-SL functionality can be defined both by functions and operations. What separates operations from functions is the use of states (variables in traditional programming languages). Operations can manipulate local and global states and are therefore imperative. As functions they can be defined both implicitly and explicitly. An explicit operation contains a sequence of statements which is very similar to statements found in ordinary programming languages.

The main problem concerning our work is that no proof theory covering the full imperative part of VDM-SL exists. Also the proof tool does not yet deal with operations and statements. The main reason for this lack is the totally different way of proving in the imperative world: Due to side effects [30], the whole history of state changes has to be considered.

Traditional proof systems for statements use a pre-post strategy where a pre-predicate describes what is supposed to hold before execution of a specific statement and a post-predicate describes what will hold after executing the statement [19, 5]. A few proof rules for VDM-SL statements, which are adapted for exception handling, can be found in [22].

Implicit operations can be treated as implicit functions, which means a satisfiability proof obligation is generated to ensure consistency. However, explicit operations need another approach.

6.1 Generating Assertions

The central idea in the POG to ensure the definite consistency of VDM-SL operations is to generate assertions and include them into the specification. An assertion is a pre-predicate stating the property that must hold before a statement. Thus, in type checking operations the generated proof obligations are not proof rules but assertions. The abstract syntax for an assertion is:

$$\begin{aligned} \mathit{Assertion} :: \mathit{loc} &: \mathit{Location} \\ \mathit{prd} &: \mathit{Sequent}; \end{aligned}$$

The assertion is composed of a sequent and a location. As in a PO above, a sequent states the needed property. The need for a sequent will be explained later. The location is needed due to the fact that an assertion is always related to a certain location in the specification.

$$\begin{aligned} \mathit{Location} :: \mathit{oper} &: \mathit{Name} \\ \mathit{stmt} &: \mathbb{N}_1 \end{aligned}$$

The *Location* is defined by the operation name and the nth statement for which the assertion represents a pre-predicate. Nested statements are counted in the order of their appearance in the specification. Consider the following specification to calculate the average of a set of numbers:

```
SmallNat = nat
inv s == s < 256

Average: set of SmallNat ==> real
Average (s) ==
  ( dcl sum : SmallNat := 0;
    for all e in set s do
      ( if sum > 255 - e then exit <ToLarge>;
        sum := sum + e );
    return sum / card s )
```

As in traditional programming languages the numbers are restricted to a certain size (`SmallNat`). The operation takes the set of numbers as an argument and uses the local state `sum` and a loop to compute the result. In the loop an exception `<ToLarge>` is raised if the `sum` is going to exceed its maximum.

The loop statement is a good example of a construct for which it is impossible to generate a context as in the sections above. Therefore the POG generates and inserts the following assertions to ensure definite consistency:

```
Average: set of SmallNat ==> real
Average (s) ==
  ( --PO1: inv-SmallNat(0)
    dcl sum : SmallNat := 0;
    for all e in set s do
      ( if sum > 255 - e then exit <ToLarge>;
        --PO2: inv-SmallNat(sum + e)
        sum := sum + e );
    --PO3: card s <> 0
    return sum / card s )
```

`PO1` must hold before the first declaration statement to ensure that the initialisation of sum will not violate the invariant. `PO2` will be inserted before the assignment statement. It must be proved in order to ensure that the assignment will not exceed the maximum of sum. Finally, `PO3` has to be included before the return statement. It can be seen as the obligatory pre-condition of the division expression. The return statement is definitely not consistent, because `PO3` is not provable.

In the following we will change the specification to be consistent and will explain how we treat expressions in statements.

6.2 Expressions in Statements

Like the return statement above, statements may include expressions. Therefore undecidability could occur in an expression. Consider the consistent version of the specification:

```
Average2: set of SmallNat ==> real
Average2 (s) ==
  ( dcl sum : SmallNat := 0;
    for all e in set s do
      ( if sum > 255 - e then exit <ToLarge>;
        sum := sum + e );
    return if s <> {}
           then sum / card s
           else 0 )
```

Average2 returns zero if the set of numbers is the empty set by using an if expression. To be able to use the context in the expression the POG collects the context in expressions as in functions. The difference from functions is that the POG type checking an operation generates a sequent assertion. The sequent notation is used to express the context of an expression as local context. The generated assertions for Average2 demonstrate this approach:

```
Average2: set of SmallNat ==> real
Average2 (s) ==
  ( --PO1: inv-SmallNat(0)
    dcl sum : SmallNat := 0;
    for all e in set s do
      ( if sum > 255 - e then exit <ToLarge>;
        --PO2: inv-SmallNat(sum)
        sum := sum + e );
    --PO3: s <> {} ==> card s <> 0
    return if s <> {}
           then sum / 0
           else 0 )
```

Now PO3 states that under the assumption that s is unequal to the empty set, the cardinality of s is unequal to zero. Which is by definition of cardinality true (==> is the turnstile $\vdash$).

7 Relation to Previous Work

Some similar work has been done before, but the specified proof obligation generator is the first one for VDM-SL covering the advanced topics of pattern matching and the non-functional part (operations and states). The central idea influencing this work comes from Flemming M. Damm, Hans Bruun, and Bo Stig Hansen [10, 11]. But also some other tools (methods) already generating proof obligations motivated this extension of the IFAD VDM-SL Toolbox with a proof obligation generator.

RAISE: In the RAISE Specification Language [15, 27] an expressive notion of types is also used for type checking. As a separate facility, not part of type checking, proof obligations called "confidence conditions" may be generated. These rule out, e.g., dynamic type errors.

PVS: PVS (Prototype Verification System) [25] is an environment for specification and verification consisting of a specification language, a parser, a type checker, and an interactive theorem prover. The type checker in PVS generates proof obligations called TCCs (Type Correctness Condition). However, PVS only supports (tagged) disjoint union types, which do not cause undecidability like the non-disjoint unions in VDM-SL. The TCCs relate to subtypes, partial operators and the termination of recursive functions.

B-Method: The B-Toolkit [17] has a proof obligation generator, that can be invoked from the "Main Environment". The POs are generated according to the correctness criteria which are required to hold within the B-Method [1]. Thus, for example the criteria requires that an Abstract Machine initialisation must establish the invariant, and that each operation re-establishes the invariant.

Z/EVES: The Z/EVES system [28, 29], a Z front-end to the theorem prover EVES [9] provides domain checking for Z specifications, which is the generation of proof obligations to ensure that all functions and operators are applied with parameters inside their domain. The same is done in our work, but Z does not have a type system as rich as VDM [18]. Our work covers a wider area, e.g. union types and patterns. Thus, domain checking is a subset of our consistency checks.

SPARK: SPARK is an annotated subset of Ada, designed to eliminate ambiguities and insecurities of the full Ada language [8]. The SPARK Examiner, a tool which checks conformance of a program to the rules of SPARK, also generate POs called "verification conditions". Mandatory annotations in a program are used to generate these POs. However, SPARK does not allow exceptions and overloading and has simplified scope and visibility rules. Unlike in our approach to operations, where annotations (assertions) are generated as proof obligations, in SPARK the annotations are added by the programmer in order to be able to generate POs [14].

8 Concluding Remarks

In this paper we have presented an approach for automatic generation of proof obligations for VDM-SL. We have shown the general strategy and presented more details about the kinds of constructs for which this kind of proof obligation generation has not been done before. The approach presented has also been formalised in terms of a VDM-SL specification for which a few extracts have been shown. From a tool point of view this work is an improvement of the existing VDM-SL Toolbox. For a more detailed presentation of this work we refer the reader to [4].

At present no work has been done on generating proof obligations for termination of recursive functions. This work could also be extended by a closer integration with the proof tool being developed. In particular we envisage a possibility for the proof tool to automate the proof of type judgements by sending it to the POG.

The proof obligation generation approach presented here has several advantages: (1) A larger subset of specifications can be formally checked according to their internal consistency. (2) By using a proof tool many simple proof obligations can be proved automatically. Thus, the link of automatic type checking and theorem proving extends the set of specifications that can be checked automatically. (3) Proof obligations provide more information than an error message. The user who is familiar with the basics of logic reasoning, will often detect the missing parts in a specification. (4) By feeding the type errors as proof obligations into a theorem prover, it leaves less for human examination, which reduces the possibility of errors being disregarded by the user.

We feel that the combination of a proof obligation generator as presented here and a proof support tool currently being developed will be a powerful combination. However, more work is needed to make a tighter integration between such tools.

Acknowledgement

The authors would like to thank their friends and colleagues for a careful reading of this article: Sten Agerholm, Brigitte Fröhlich, Jeppe Nyløkke Jørgensen, Paul Mukherjee and Anne Berit Nielsen. Particularly Sten Agerholm has provided considerable input on the directions of the work reported in this paper. We are grateful to the support of the Austrian Federal Ministry of Science, Research and the Arts (Kurt Gödel Scholarship) for Bernhard K. Aichernig's stay at IFAD. Finally, we want to thank Bernhard's professor, Peter Lucas, who made his stay possible.

References

1. J.-R. Abrial. *The B Book – Assigning Programs to Meanings.* Cambridge University Press, August 1996.
2. S. Agerholm and J. Frost. An Isabelle-based theorem prover for VDM-SL. In *Proceedings of the 10th International Conference on Theorem Proving in Higher Order Logics (TPHOLs'97)*, LNCS. Springer-Verlag, August 1997.
3. S. Agerholm and J. Frost. Towards an integrated CASE and theorem proving tool for VDM-SL. FME'97, September 1997.
4. Bernhard K. Aichernig. A Proof Obligation Generator for the IFAD VDM-SL Toolbox. Master's thesis, Technical University Graz, Austria, March 1997.
5. K. Apt. Ten Years of Hoare's Logic: A survey - Part I. *ACM-TOPLAS*, 3(4):431–483, Oct 1981.
6. Juan Bicarregui, John Fitzgerald, Peter Lindsay, Richard Moore, and Brian Ritchie. *Proof in VDM: A Practitioner's Guide.* FACIT. Springer-Verlag, 1994. ISBN 3-540-19813-X.
7. Hans Bruun, Flemming Damm, and Bo Stig Hansen. An Approach to the Static Semantics of VDM-SL. In *VDM '91: Formal Software Development Methods*, pages 220–253. VDM Europe, Springer-Verlag, October 1991.
8. Bernard Carre, William Marsh, and Jon Garnsworthy. SPARK: A Safety-Related Ada Subset. In *Ada UK Conference*, pages 1–19, August 22 1992.
9. Dan Craigen, Sentot Kromodimoeljo, Irwin Meisels, Bill Pase, and Mark Saaltink. Eves: An overview. In S. Prehn and W.J. Toetenel, editors, *VDM'91 – Formal Software Development Methods*, pages 389–405. Springer-Verlag, October 1991.

10. Flemming Damm, Hans Bruun, and Bo Stig Hansen. On Type Checking in VDM and Related Consistency Issues. In *VDM '91: Formal Software Development Methods*, pages 45–62. VDM Europe, Springer-Verlag, October 1991.
11. Flemming M. Damm and Bo Stig Hansen. Generation of Proof Obligations for Type Consistency. Technical Report 1993-123, Department of Computer Science, Technical University of Denmark, December 1993.
12. John Dawes. *The VDM-SL Reference Guide*. Pitman, 1991. ISBN 0-273-03151-1.
13. René Elmstrøm, Peter Gorm Larsen, and Poul Bøgh Lassen. The IFAD VDM-SL Toolbox: A Practical Approach to Formal Specifications. *ACM Sigplan Notices*, 29(9):77–80, September 1994.
14. Jon Garnsworthy, Ian O'Neill, and Bernhard Carré. Automatic Proof of Absence of Run-time Errors. In *Ada UK Conference*. London Docklands, October 1993.
15. The RAISE Language Group. *The RAISE Specification Language*. The BCS Practitioners Series. Prentice-Hall, 1992.
16. The VDM Tool Group. User Manual for the IFAD VDM-SL Toolbox. Technical report, IFAD, May 1996. IFAD-VDM-4.
17. Howard Haughton. *Specification in B: An Introduction Using the B Toolkit*. World Scientific Publishing, 1996.
18. I.J. Hayes, C.B. Jones, and J.E. Nicholls. Understanding the Differences Between VDM and Z. *FACS Europe*, pages 7–30, Autumn 1993.
19. C.A.R. Hoare. An Axiomatic Basis for Computer Programming. *Communications of teh ACM*, 12(10):576–581, October 1969.
20. Cliff Jones, Kevin Jones, Peter Linsay, and Richard Moore, editors. *mural: A Formal Development Support System*. Springer-Verlag, 1991. ISBN 3-540-19651-X.
21. Cliff B. Jones. *Systematic Software Development Using VDM*. Prentice-Hall International, Englewood Cliffs, New Jersey, second edition, 1990. ISBN 0-13-880733-7.
22. Peter Gorm Larsen. *Towards Proof Rules for VDM-SL*. PhD thesis, Technical University of Denmark, Department of Computer Science, March 1995. ID-TR:1995-160.
23. P.G. Larsen, B. S. Hansen, H. Brunn, N. Plat, H. Toetenel, D. J. Andrews, J. Dawes, G. Parkin, and et. al. Information Technology – Programming languages, their environments and system software interfaces – Vienna Development Method – Specification Language – Part 1: Base language, ISO/IEC 13817-1, December 1996.
24. Paul Mukherjee. Computer-aided Validation of Formal Specifications. *Software Engineering Journal*, pages 133–140, July 1995.
25. Sam Owre, John Rushby, Natarajan Shankar, and Friedrich von Henke. Formal Verification for Fault-Tolerant Architectures: Some Lessons Learned. In J.C.P. Woodcock and P.G. Larsen, editors, *FME'93: Industrial-Strength Formal Methods*, pages 482–501. Formal Methods Europe, Springer-Verlag, April 1993. Lecture Notes in Computer Science 670.
26. Nico Plat and Peter Gorm Larsen. An Overview of the ISO/VDM-SL Standard. *Sigplan Notices*, 27(8):76–82, August 1992.
27. The RAISE Method Group. *The RAISE Development Method*. The BCS Practitioners Series. Prentice-Hall International, 1995.
28. Mark Saaltink. Z and EVES. In J.E. Nicholls, editor, *Z User Workshop, York 1991*, pages 223–242. Springer-Verlag, 1992. Workshops in Computing.
29. Mark Saaltink. The Z/EVES system. Technical report, ORA Canada, September 1995.

30. R.D. Tennent. *Principles of Programming Languages.* Prentice-Hall International, Englewood Cliffs, New Jersey 07632, 1981.
31. The VDM Tool Group. The IFAD VDM-SL Language. Technical report, IFAD, May 1996. IFAD-VDM-1.

This article was typeset using the LaTeX macro package with the LLNCS2E class.