Camila Revival: VDM meets Haskell*

Joost Visser, J.N. Oliveira, L.S. Barbosa, J.F. Ferreira, and A. Mendes

Departamento de Informática, Universidade do Minho, Braga, Portugal
joost.visser@di.uminho.pt
http://www.di.uminho.pt/

Abstract. We have experimented with modeling some of the key concepts of the VDM specification language inside the functional programming language Haskell. For instance, VDM's sets and maps are directly available as data types defined in standard libraries; we merely needed to define some additional functions to make the match complete. A bigger challenge is posed by VDM's data type invariants, and pre- and post-conditions. For these we resorted to Haskell's constructor class mechanism, and its support for monads. This allows us to switch between different modes of evaluation (e.g. with or without property checking) by simply coercing user defined functions and operations to different specific types.

1 Introduction

CAMILA [2, 3] is "A System for Software Development using Formal Methods" (`http://camila.di.uminho.pt`) developed and used by the Formal Methods group from Braga in the early nineties[1]. The CAMILA specification language is a dialect of VDM [14, 11]. Specification animation is provided via XLISP generation and its run-time system is based upon Henderson's `metoo` [13] library, reimplemented and extended over Betz' XLISP interpreter [6]. This led to the locally developed `xmetoo` interpreter (available since 1986) around which a more elaborate system was designed — the CAMILA toolkit. This includes a language processor, developed in Lex/Yacc/C, a LaTeX-based pretty-printer, a LaTeX-based literate-programming module (`camtex`) and an (experimental) data refinement laboratory based on the SETS [22, 23] calculus. An illustration of the capabilities offered by the system can be found in [1].

This locally brewed system was shelved in favour of VDM-SL [11] (for specification) and Haskell [15] (for development) when these gained momentum in

* Supported by Fundação para a Ciência e a Tecnologia, POSI/ICHS/44304/2002.

[1] The CAMILA project (1990-93) was funded by JNICT (Grant 169/90). Further developments were supported by the U.Minho Informatics Department and INESC Group 2361. CAMILA was named after "Camilo Castelo Branco", a famous 19th century Portuguese novelist who lived in the Minho province. This choice of acronym dates back to 1990, the year when the first version of CAMILA became available and the centenary of Camilo's death.

the late nineties, with the availability of software such as IFAD's VDMTools and Hugs/GHC, respectively.

Haskell and VDM have their own merits, being complementary in several respects. Given the functional flavour of CAMILA and its proximity to VDM, the aim of the CAMILA Revival effort is to let Haskell and VDM meet, so that advantages of their relative merits are taken. In a sense, the rôle played by Lisp in the CAMILA legacy code is to be taken by Haskell this time.

Work in the CAMILA Revival started by rewriting in Haskell the CAMILA standard library (itself an extension of the `metoo` kernel). This led to the `CPrelude` library which, as reported in [17] and available from the UMinho Haskell Libraries[2], includes the animation of the algebras of finite sets and finite maps, both implemented over Haskell's `FiniteMap` module.

More recently, reference [16] describes how VDM-SL code is to be encoded in monadic Haskell, the effort being in designing an interpreter-level software architecture able to cope, in a flexible way, with those features of VDM-SL and VDM^{++} [12] which Haskell doesn't support directly. These include partial behaviour (e.g. datatype invariants and pre-/post-conditions) and persistence (local state). The effort here has been to separate these ingredients and treat them as effects captured by dedicated monads, such as the error, state, and I/O monads. A monadic model of VDM-SL in Gofer, a precursor of Haskell, was proposed earlier by [19].

In the current paper, we experiment with an improvement over the monadic modeling of VDM features of [16] and [19]. In particular, we show how switching between evaluation modes can be controlled with parameterized monads. The parameters are introduced at the type level, which allows such switching to be done statically and without adapting any function declarations.

As a running example, we will use the VDM specification of stacks of odd integers, shown in Figure 1.

2 Monadic invariants and conditions

In VDM, datatypes can be constrained by providing a predicate as datatype invariant together with the structural definition. Functions can be constrained on their input type by providing a predicate on that type as pre-condition. Likewise, a predicate on result and input types can be specified as post-condition.

A straightforward way of adding datatype invariants, pre- and post-conditions to Haskell programs would be to weave invocations to the predicates that model them into the function definitions. However, this has two drawbacks.

1. The function definitions become more complex, since they need to be augmented with appropriate dataflow for checking conditions and taking appropriate actions when conditions are validated.
2. There is no way of influencing the type of action taken on condition violation, except by changing the weaved-in code. In particular, when a prototype has

[2] See `http://wiki.di.uminho.pt/wiki/bin/view/PURe/Camila`.

```
types

  Stack = seq of int
  inv s = forall a in set elems s & odd(a);

functions

  empty : Stack -> bool
  empty(s) == s = [];

  push : int * Stack -> Stack
  push(p,s) == [p] ^ s
  pre odd(p) ;

  pop  : Stack -> Stack
  pop(s) == tl s
  pre not empty(s);

  top  : Stack -> int
  top(s) == hd s
  pre not empty(s);
```

Fig. 1. Running example: a VDM specification of stacks of odd integers. We use VDM-SL/VDM^{++} surface syntax in the Camila revival, while the Camila dialect is kept as well, for continuity.

evolved into a deliverable system, property testing can be turned off only at the cost of numerous changes throughout the code.

To avoid these drawbacks, we will use monads to encapsulate the checking behaviour. To this end, we introduce a type class to capture invariants, as well as a monad and a parameterized monad transformer to encapsulate checking and reporting behaviour.

2.1 Constrained datatypes

To model datatype invariants in Haskell, we follow [20, 16] and introduce the type class `CData` (for Constrained Data), as follows:

```
class CData a where
    -- | Invariant as boolean function.
    inv :: a -> Bool
    inv a = True      -- default
```

The `CData` type class has as its single member a boolean predicate `inv` on the datatype being constrained.

An example of a constrained datatype is the stack of odd numbers of Figure 1 which is specified in Haskell as follows:

```
newtype Stack = Stack { theStack :: [Int] }
instance CData Stack where
   inv s = all odd (theStack s)
```

Thus, we use a `newtype` definition with a single constructor `Stack` and destructor `theStack`. The invariant is specified as an instance of the `CData` type class. We do not use a Haskell type synonym to model stacks, because the instance specification would then not constrain just stacks, but any list of integers that might occur in our system.

Datatype constraints propagate through products, sums, lists, etc, as the following definitions illustrate:

```
instance (CData a, CData b) => CData (a,b) where
    inv (a,b) = (inv a) && (inv b)

instance (CData a, CData b) => CData (Either a b) where
    inv (Left a) = inv a
    inv (Right b) = inv b
```

Similar definitions can be provided to make datatype constraints propagate through other datatype constructors of Haskell's standard libraries, such as lists.

2.2 A monad with checking behaviour

A `CamilaMonad` is defined as a monad with additional operations that support condition and invariant checking.

```
class Monad m => CamilaMonad m where
  -- | Check precondition
  pre :: Bool -> m ()
  -- | Check postcondition
  post :: Bool -> m ()
  -- | Check invariant before returning data in monad
  returnInv :: CData a => a -> m a
```

The `pre` and `post` members take a boolean value as argument, and return a monadic computation. Depending on the chosen monad instance, this computation may or may not check the given condition, and may take different kinds of action when a condition violation is detected. The `returnInv` member stores a given value of a constrained datatype in the monad, and may check its constraint while doing so.

The `CamilaT` monad transformer adds checking effects to a given base monad. Which effects exactly depends on the phantom mode argument.

```
data CamilaT mode m a = CamilaT {runCamilaT :: m a} deriving Show
instance Monad m => Monad (CamilaT mode m) where
  return a = CamilaT $ return a
  ma >>= f = CamilaT (runCamilaT ma >>= runCamilaT . f)
```

The monad transformer is parameterized with a phantom mode argument to control the behaviour at constraint violation.

Using the type classes specified above, we can now implement functions with pre- and post-conditions, e.g. the `top` function of our stack example of Figure 1.

```
top :: CamilaMonad m => Stack -> m Int
top s = do
 pre (not $ empty s)
 let result = head $ theStack s
 post (((result:(tail $ theStack s)) == theStack s) && (odd result))
 return result
```

The `top` function in Haskell is monadic, which is indicated by the result type `m Int`. The monad `m` is a parameter, which is constraint to be at least a `CamilaMonad m` to guarantee availability of the `pre` and `post` functions. These functions are actually invoked at the beginning and the end of the function, taking predicates on the input argument `s` and the output value `result` as arguments. Note that no post-condition was present in the VDM specification of Figure 1; we added it here just to demonstrate the use of `post`.

3 Modes of evaluation

Various different choices are possible for the phantom mode argument. We will implement the following modes:

free fall In the free fall mode, no checking is performed whatsoever. When violations of invariants or conditions occur, they will go undetected, which may lead to inconsistent results, or run-time failures at later stages of the computation.

warn In warn mode, invariants and conditions are checked. When found violated, a warning will be issued, but computation proceeds as if nothing happened. This may again lead to inconsistent result or run-time failures at later stages, but at least some diagnostic information will be emitted.

fail In fail mode, invariant and conditions are checked, and when found violated a run-time error is forced immediately.

error In error mode, invariants and conditions are checked, and when found violated an error or exception will be thrown.

Below, we will explain how each of these modes can be implemented with appropriate instances of the `CamilaMonad` class. This explanation is only for those readers who are well-versed in Haskell, and want to know what goes on 'under the hood'. Others may skip the remainder of this section.

3.1 Free fall

To select free fall mode, we will use the following type:

```
data FreeFall
```

This type has no constructors, which is fine, because we will only use it in phantom position, i.e. to instantiate data type parameters that do not occur on the right-hand side.

The free fall mode can be used in combination with any monad, because it doesn not require any specific behaviour for checking or reporting. So, we define the following instance of the `CamilaMonad`:

```
instance Monad m => CamilaMonad (CamilaT FreeFall m) where
  pre p = return ()
  post p = return ()
  returnInv a = return a
```

Here, the pre- and post-condition members simply ignore their boolean argument, and the invariant-aware return simply does not check the invariant. Thus, when using this instance of the Camila monad, no checking or reporting occurs.

A convenience function can be defined to enforce free fall mode without explicitly providing types:

```
freeFall :: CamilaT FreeFall m a -> CamilaT FreeFall m a
freeFall = id
```

Thus, when the `freeFall` combinator is applied to a monadic function, it will be coerced to use the Camila monad with free fall mode.

3.2 Warn

In warn mode, we want to perform checks and, on violation, emit warnings but keep running as if all is well. We define the following datatype constructor:

```
data Warn
```

To enable reporting, we need a monad with writing capabilities. This can be the `IO` monad, for instance:

```
instance MonadIO m => CamilaMonad (CamilaT Warn m) where
  pre p = unless p $ liftIO $ putErr "Pre-condition violation"
  post p = unless p $ liftIO $ putErr "Post-condition violation"
  returnInv a = do
    unless (inv a) $ liftIO $ putErr "Invariant violation"
    return a
instance MonadIO m => MonadIO (CamilaT mode m) where
  liftIO = CamilaT . liftIO
```

The `unless` combinator runs its monadic argument conditionally on its boolean argument. Thus, when a violation occurs, a warning string is emitted on standard error.

Instead of the `IO` monad (a sin bin with many capabilities beyond writing), we can take a more pure approach where a simple writer monad is used:

```
instance CamilaMonad (CamilaT Warn (Writer [CamilaViolation])) where
  pre p = if p then return () else tell [PreConditionViolation]
  post p = if p then return () else tell [PostConditionViolation]
  returnInv a = if (inv a) then return a else do
    tell [InvariantViolation]
    return a
```

And likewise for the writer monad transformer:

```
instance Monad m
      => CamilaMonad (CamilaT Warn (WriterT [CamilaViolation] m)) where
  pre p = if p then return () else tell [PreConditionViolation]
  post p = if p then return () else tell [PostConditionViolation]
  returnInv a = if (inv a) then return a else do
    tell [InvariantViolation]
    return a
```

And we can lift writing capabilities to the level of the camila monad:

```
instance MonadWriter w m => MonadWriter w (CamilaT mode m) where
  tell w = CamilaT $ tell w
  listen ma = CamilaT $ listen $ runCamilaT ma
  pass maf = CamilaT $ pass $ runCamilaT maf
```

To enforce warn mode without need to write explicit types, we define a convenience function again:

```
warn :: CamilaT Warn m a -> CamilaT Warn m a
warn = id
```

3.3 Fail

In fail mode, we want to perform checks, as in warn mode, but when violations are detected we want to force an immediate fatal error, rather than emit a warning. We define the following datatype to select fail mode:

```
data Fail
```

Fail mode can work with any monad, since no writing capabilities are needed.

```
instance Monad m => CamilaMonad (CamilaT Fail m) where
  pre p = if p then return () else fail "Pre-condition violation"
  post p = if p then return () else fail "Post-condition violation"
  returnInv a = if (inv a) then return a
                           else fail "Invariant violation"
```

Thus, when violations are detected, the standard `fail` function is used to force an immediate fatal error.

The following convenience function enforces fail mode (called `fatal`, since `fail` is already taken):

```
fatal :: CamilaT Fail m a -> CamilaT Fail m a
fatal = id
```

3.4 Error

In error mode, when a violation is detected, we want to signal an non-fatal error, i.e. an error that allows recovery. Haskell offers a type class `Error` that has as instances all types used as error types, among which:

```
instance Error IOError
instance Error CamilaViolation
```

Instead of defining a dedicated type constructor to select error mode, we will use these error types themselves to select error mode, using that specific error type.

We can define camila monads that operate with error mode on the basis of any monad with error capabilities. The `IO` monad is an example:

```
instance MonadError IOError m => CamilaMonad (CamilaT IOError m) where
  pre p  = if p then return ()
                else throwError $ userError "Pre-condition violation"
  post p = if p then return ()
                else throwError $ userError "Post-condition violation"
  returnInv a
         = if (inv a) then return a
                      else throwError $ userError "Invariant violation"
```

The `userError` function raises an `IOError` which can be caught higher in the call chain.

If we use our designated `CamilaViolation` as error type, the following instance can be defined:

```
instance MonadError CamilaViolation m
      => CamilaMonad (CamilaT CamilaViolation m) where
  pre p  = if p then return () else throwError PreConditionViolation
  post p = if p then return () else throwError PostConditionViolation
  returnInv a
         = if (inv a) then return a else throwError InvariantViolation
instance MonadError e m => MonadError e (CamilaT e m) where
  throwError e = CamilaT $ throwError e
  catchError ma f = CamilaT $ catchError (runCamilaT ma) (runCamilaT . f)
```

Thus, an error of type `CamilaViolation` is raised with `throwError`, whenever a violation is detected.

To enforce error mode with specific error types, the following convenience functions can be used:

```
errorMode :: MonadError e m => CamilaT e m a -> CamilaT e m a
errorMode = id

camilaViolation :: MonadError CamilaViolation m
  => CamilaT CamilaViolation m a -> CamilaT CamilaViolation m a
camilaViolation = id
```

```
newtype Stack = Stack { theStack :: [Int] }
instance CData Stack where
   inv s = all odd (theStack s)

empty :: Stack -> Bool
empty s = theStack s == []

push :: CamilaMonad m => Int -> Stack -> m Stack
push n s = do
  pre (odd n)
  returnInv $ Stack (n: theStack s)

pop :: CamilaMonad m => Stack -> m Stack
pop s = do
  pre (not $ empty s)
  returnInv $ Stack $ tail $ theStack s

top :: CamilaMonad m => Stack -> m Int
top s = do
  pre (not $ empty s)
  let result = head $ theStack s
  return result
```

Fig. 2. Example in Haskell.

4 Example

Here we pick up the example of stacks of odd numbers from the introduction. Figure 2 shows the full Haskell code into which that example specification can be translated. The type definition of `Stack` and the monadic function definition of `top` were discussed above.

The predicate `empty` to test for emptyness does not need to be monadic, because it does not need to check invariants or conditions. The `push` and `pop` functions are monadic for two reasons. They have a pre-condition, and they need to check the invariant of the returned stack.

4.1 Taking top of an empty stack

For purposes of demonstration, we define two programs, each involving a different violation. In the first example, we take the top of an empty stack. This is illegal, and if it goes unchecked, it can only lead to a fatal error.

```
testTopEmptyStack :: CamilaMonad m => m Int
testTopEmptyStack = do
  s <- initStack    -- create empty stack
  n <- top s
  return n
```

We can run this example in the four different evaluation modes defined above, as follows:

```
> runCamilaT $ freeFall testTopEmptyStack
*** Exception: Prelude.head: empty list

> runCamilaT $ warn testTopEmptyStack
Pre-condition violation
*** Exception: Prelude.head: empty list

> runCamilaT $ fatal testTopEmptyStack
*** Exception: Pre-condition violation

> runCamilaT $ errorMode testTopEmptyStack
*** Exception: user error Pre-condition violation
```

In free fall mode, a run-time exception occurs without any warning. As the message of the exception indicates, the cause is application of the standard prelude function `head` to an empty list. In warn mode, the same exception occurs, but a warning is issued before, when the pre-condition violation is detected. In fail mode, a run-time exception occurs at the moment of detection, as the message indicates, before even attempting application of the `head` function to an empty list. Finally, in error mode, an exception is raised at the same moment, but the text `user error` in the message indicates that this exception is actually catchable, and not necessarily fatal.

This difference between fail mode and error mode becomes clear when we try to catch the generated exceptions:

```
> (runCamilaT $ fatal testTopEmptyStack)
  `catchError` \_ -> putStrLn "CAUGHT" >> return 42
*** Exception: Pre-condition violation

> (runCamilaT $ errorMode testTopEmptyStack)
  `catchError` \_ -> putStrLn "CAUGHT" >> return 42
CAUGHT
```

Thus, the exceptions that occur in error mode can be caught, higher in the call chain, while in fail mode the exception always gets propagated to the top level.

4.2 Pushing an even number onto the stack

A different situation occurs when we try to push an even number onto the stack, as follows:

```
testPushEvenOnStack :: CamilaMonad m => m Int
testPushEvenOnStack = do
  s <- initStack
  s <- push 0 s
```

```
  n <- top s
  return n
```

This test function violates the datatype invariant that requires the stack to hold odd numbers only. But, without checking, this violation will not lead to an immediate error, but might go unnoticed until much later.

We can run this second test function in our four different evaluation modes, as follows:

```
> runCamilaT $ freeFall testPushEvenOnStack

> runCamilaT $ warn testPushEvenOnStack
PreConditionViolation
InvariantViolation
PostConditionViolation

> runCamilaT $ fatal testPushEvenOnStack
*** Exception: PreConditionViolation

> runCamilaT $ errorMode testPushEvenOnStack
*** Exception: user error (PreConditionViolation)
```

In free fall mode, no exceptions occur. The violation of the data type invariant simply goes unnoticed. In warn mode, three subsequent warnings are issued, corresponding to the various checking moments in the `push` and `top` functions (the latter with the post condition inserted, see Section 2.2, for demonstration purposes). Again, no exception occurs. In fail and error mode, the behaviour is as in the case of `testTopEmptyStack`.

5 Related work

VDM conversion into Gofer. A strategy for automatically translating VDM-SL specifications into Gofer (a precursor of the Haskell language) and a tool that performs such translations are presented in [19]. The use of monads in this translation is limited to state and error monads for modeling VDM state variables and exceptions. Neither datatype invariants nor pre-conditions on explicit functions are translated. Pre- and post-conditions on implicit functions are translated into boolean predicates, rather than monadic functions, and are not invoked during evaluation.

VDM conversion into Lazy ML. Reference [7] describes a method for converting model-based specifications (an executable subset of VDM) into Lazy ML. Monads are not used. Type invariants are translated to functions which are invoked at input parameter passing time (rather than at value return time). Operations are modeled by functions whose functionality is augmented with state (both at input and output level).

Irish VDM. Reference [8] describes Haskell libraries, including QuickCheck support, which implement the operators of Irish VDM[3]. Sets are modeled by strictly ordered lists and finite maps are lists of pairs whose first projections are strictly ordered. Particular attention is paid to the proof obligations associated with each operator implementation. Finite relations are modeled by their powerset transposes, later to be specialized into *directed irreflexive multigraphs*.

Programatica. This is a system for the development of high-confidence software systems and executable systems specification[4] which encourages users to state and validate properties of software as an integral part of the programming process. Assertions are type-checked to ensure a base level of consistency with executable portions of the program and annotated with "certificates" that provide evidence of validity. Different forms of certificate are supported, offering a wide range of validation options — from low-cost instrumentation and automated testing, to machine-assisted proof and formal methods. A suite of "property management" tools provides users with facilities to browse or report on the status of properties and associated certificates within a program, and to explore different validation strategies. Programatica finds its inspiration in type theory rather than set theory-based methods like VDM.

Jakarta Commons Logging (JCL). The Jakarta project of the Apache Software Foundation offers logging support in the form of a `LogFactory` class that creates concrete implementations of a `Log` interface[5]. The interface offers methods like `fatal`, `error`, and `warn` to emit messages to consoles and/or log files. To decide which concrete implementation to create, a layered discovery process is applied, involving inspection of configuration attributes and reflective techniques, such as class loading and resource discovery. Thus, switching between logging behaviours can be accomplished e.g. by moving `jar` files on or off the class path. Similar with the reporting in our monadic Haskell model of VDM property checking, the cross-cutting logging behaviour is altered without changing code, but by selection of different concrete types that implement the same interface. In contrast with our approach, JCL accomplishes the switching dynamically, and outside the semantics of the Java language.

VDMTools Existing VDM environments, such as IFAD's VDMTools [10, 11], offer the possibility of enabling different debugging and dynamic checking features during interpretation of VDM specifications. Checking of invariants, of pre-conditions, and of post-conditions can be turned on and off individually. Our monadic model can be seen as the (executable) specification of such an interpreter, though we offer variability in the checking behaviour (four different modes), for all three types of check simultaneously.

[3] `http://www.cs.tcd.ie/Andrew.Butterfield/IrishVDM`
[4] `http://www.cse.ogi.edu/PacSoft/projects/programatica`.
[5] `http://jakarta.apache.org/commons/logging/`

6 Concluding remarks

We have shown a novel way of modeling VDM-SL functions, possibly involving pre- and post-conditions and constrained datatypes, as monadic Haskell functions. We defined an extended monad interface (`CamilaMonad`) that can be instantiated in different ways to enable different modes of evaluation. Each mode offers different behaviour regarding property checking and violation reporting. Switching between modes is controlled by a single phantom type argument, which may be supplied via convenience wrapper-functions.

We envision several usage scenarios in which different evaluation modes and the ability to switch between them are useful. The *free fall* mode is useful after a fully tested and verified system is taken into operation. For such as system, correct operation is guaranteed, and all checking and reporting can be turned off. When a system is taken into operation without full verification, the *fail* mode may be useful to kill the system as soon as a condition is violated, rather than letting it run and risk dispersion of inconsistencies throughout the system. The *warn* mode can be useful when assessing the robustness of a system through fault injection [9]. A trace will be emitted of all violations that occur as a consequence of an injected fault, showing which parts of the system are vulnerable. Finally, during unit and functional testing, it is most appropriate to evaluate in *error* mode, which forces early exceptions that are catchable by a testing environment and provide information about the origin of test failures.

With our approach, all these scenarios can peacefully coexist, because the system does not need to be changed to evaluate it in a different mode. In addition, the switching is accomplished within the semantics of the source language, rather than through instrumentation of compiler-generated code.

6.1 Relevance to Overture

Both the Overture and the Camila Revival projects aim to create tool support for VDM dialects. Overture is Java based, and aspires to be an industry-strength development environment. We have chosen Haskell as base language, and our primary goal is to create a platform for experimentation with research ideas. We hope the outcome of such experiments may lead to inspiration for future developments in projects such as Overture.

For instance, we believe that our approach to grammar engineering and grammar-centered language tool development [5] deserves wider adoption. The relational data model calculator VooDooM [4], as well as the reverse engineering of VDM data models from relational ones [21], demonstrate how advanced model transformation and generation techniques can augment the standard language tools support, such as editing and compilation.

The monadic model for VDM property checking presented in the current paper has relevance for VDM compiler/interpreter construction efforts. It provides an answer to how property checking may be understood semantically. When constructing a compiler or interpreter, such semantics need to be implemented. When compiling to Java, for instance, our monadic model so far suggests to

consider using *class parameters* to switch between evaluation modes, possibly using a model of monads in Java.

6.2 Future work

Many future challenges exist, both regarding our monadic model of VDM and more generally in the Camila Revival project as a whole.

The messages generated upon detection of violations presently do not identify which property exactly is concerned. The various member functions of the `CamilaMonad` can be easily extended to provide room for line numbers or specific messages. In [20] a method for specifying gradually more specific error messages is proposed in the context of a monadic treatment of relational data models. Apart from the four modes and their implementations discussed here, further modes may be explored, for instance supporting a richer error type. In [16], we have already experimented with monadic support of VDM's notion of state and we intend to enrich the improved monadic model present here with such functionality.

Other ongoing work in the Camila Revival project, apart from the monadic model, includes the development of static analysis tools for VDM, and conversion tools to, from, and within VDM, of which VooDooM is a first instance [4]. Further, we are developing a component model [18] to capture component behaviour and interaction.

References

1. J.J. Almeida, L.S. Barbosa, F.L. Neves, and J.N. Oliveira. Bringing camila and sets together — the `bams.cam` and `ppd.cam` camila Toolset demos. Technical report, DI/UM, Braga, December 1997. [45 p. doc.].
2. J.J. Almeida, L.S. Barbosa, F.L. Neves, and J.N. Oliveira. Camila: Formal software engineering supported by functional programming. In A. De Giusti, J. Diaz, and P. Pesado, editors, *Proc. II Conf. Latino Americana de Programacin Funcional (CLaPF97)*, pages 1343–1358, La Plata, Argentina, October 1997. Centenario UNLP.
3. J.J. Almeida, L.S. Barbosa, F.L. Neves, and J.N. Oliveira. Camila: Prototyping and refinement of constructive specifications. In M. Johnson, editor, *6th Int. Conf. Algebraic Methods and Software Technology (AMAST)*, number 1349 in Lecture Notes in Computer Science, pages 554–559. Springer-Verlag, December 1997. 6th International Conference, AMAST'97, Sydney, Australia, 13–17 December 1997, Proceedings.
4. T. Alves, P. Silva, J. Visser, and J.N. Oliveira. Strategic term rewriting and its application to a VDM-SL to SQL conversion. In *Proceedings of the Formal Methods Symposium (FM'05)*. Springer, 2005. To appear.
5. T. Alves and J. Visser. Development of an industrial strength grammar for VDM. Technical Report DI-PURe-05.04.29, Universidade do Minho, April 2005.
6. D. Betz. Xlisp: An experimental object oriented language. Technical report, Manchester, 1985.

7. P. Borba and S. Meira. From VDM specifications to functional prototypes. *Journal of Systems and Software*, 3(21):267–278, June 1993.
8. A. Butterfield. Haskell library for Irish VDM. Technical report, Dept. of Computer Science, Trinity College, Dublin University., April 2005.
9. J. Carreira, H. Madeira, and J. G. Silva. Xception: Software fault injection and monitoring in processor functional units. *IEEE Transactions on Software Engeenering*, 24(2), February 1998.
10. R. Elmstrom, P.G. Larsen, and P.B. Lassen. The IFAD VDM-SL toolbox: a practical approach to formal specifications. *SIGPLAN Notices*, 29(9):77–80, 1994.
11. J. Fitzgerald and P.G. Larsen. *Modelling Systems: Practical Tools and Techniques for Software Development* . Cambridge University Press, 1st edition, 1998.
12. J. Fitzgerald, P.G. Larsen, P. Mukherjee, N. Plat, and M. Verhoef. *Validated Designs for Object–oriented Systems*. Springer, New York, 2005.
13. P. Henderson. Me Too: A language for software specification and model-building — preliminary report. Technical report, Univ. Stirling, Dec. 1984.
14. C.B. Jones. *Software Development — A Rigorous Approach*. Series in Computer Science. Prentice-Hall International, 1980. C.A. R. Hoare.
15. S.L. Peyton Jones. *Haskell 98 Language and Libraries*. Cambridge University Press, Cambridge, UK, 2003. Also published as a Special Issue of the Journal of Functional Programming, 13(1) Jan. 2003.
16. A. Mendes and J.F. Ferreira. PUReCAMILA: A System for Software Development using Formal Methods. Technical report, U. Minho, Feb. 2005.
17. A. Mendes, J.F. Ferreira, and J.M. Proença. CAMILA Prelude. Technical report, U. Minho, Sep. 2004. In Portuguese.
18. S. Meng and L.S. Barbosa. On refinement of generic state-based software components. In C. Rettray, S. Maharaj, and C. Shankland, editors, *10th Int. Conf. Algebraic Methods and Software Technology (AMAST)*, pages 506–520, Stirling, August 2004. Springer Lect. Notes Comp. Sci. (3116). Best Student Co-authored Paper Award.
19. P. Mukherjee. Automatic translation of VDM-SL specifications into Gofer. In J.S. Fitzgerald, C.B. Jones, and P. Lucas, editors, *FME*, volume 1313 of *Lecture Notes in Computer Science*, pages 258–277. Springer, 1997.
20. C. Necco. Polytypic data processing. Master's thesis, Facultad de C. Físico Matemáticas y Naturales, University of San Luis, Argentina, 2005.
21. F.L. Neves, J.C. Silva, and J.N. Oliveira. Converting informal meta-data to VDM-SL: A reverse calculation approach. In *VDM in Practice! A Workshop co-located with FM'99: The World Congress on Formal Methods, Toulouse, France*, September 1999.
22. J.N. Oliveira. A Reification Calculus for Model-Oriented Software Specification . *Formal Aspects of Computing*, 2(1):1–23, April 1990.
23. J.N. Oliveira. Software Reification using the SETS Calculus . In Tim Denvir, Cliff B. Jones, and Roger C. Shaw, editors, *Proc. of the BCS FACS 5th Refinement Workshop, Theory and Practice of Formal Software Development, London, UK*, pages 140–171. ISBN 0387197524, Springer-Verlag, 8–10 January 1992. (Invited paper).