Journal of Universal Computer Science, vol. 11, no. 10 (2005), 1580-1612
submitted: 17/6/05, accepted: 23/9/05, appeared: 28/10/05 © J.UCS

From Algebras to Objects: Generation and Composition

A. M. Cruz
(Sidereus, Consultoria Informática, SA, Porto, Portugal
mcz@sidereus.pt)
L. S. Barbosa
(Di-Cctc, Universidade do Minho, Braga, Portugal
lsb@di.uminho.pt)
J.N. Oliveira
(Di-Cctc, Universidade do Minho, Braga, Portugal
jno@di.uminho.pt)

Abstract: This paper addresses *objectification*, a formal specification technique which inspects the potential for object-orientation of a declarative model and brings the 'implicit objects' explicit. Criteria for such objectification are formalized and implemented in a runnable prototype tool which embeds Vdm-sl into Vdm++. The paper also includes a quick presentation of a (coinductive) *calculus* of such generated objects, framed as generalised Moore machines.

Key Words: software formal specification, object-orientation, object composition
Category: D.1.5, D.2.1, D.2.2

1 Introduction

The object oriented (OO) programming paradigm offers many advantages when compared to old-style, *flat* imperative programming. Concepts such as encapsulation, abstraction, inheritance, or "information hiding" are supplied in a natural manner which eases the process of software maintenance.

However, it is often the case that the expressive power of a particular programming language or paradigm is misused by programmers. Sooner or later, these will end up writing unintelligible (authorship dependent) code which is hard to maintain — a symptom observable at all programming levels. For instance, arbitrary recursion and/or (side) effects have been considered harmful in functional programming. Instead, programmers are invited to structure their code around generic program devices such as e.g. *fold/unfold* combinators (which bring discipline to recursion) and *monads* (which bring discipline to effects). The main advantage is that these devices (or program combinators) are semantically richer than their more primitive counterparts and come equipped with calculi which make it possible to reason about programs [Bird and Moor, 1997, Jacobs and Rutten, 1997].

What about object-orientation? Is a given class library always a "good" design once it seems to look like what can be found in the real world? Under which design principles can we restructure "spaghetti" C++ code, for instance? Can we re-write the same code around fewer, "better" structured classes? Should we *think* (specify) in terms of objects or are objects just an attractive implementation device? In any case how do we reason compositionally about them?

The questions above are not easy to answer. Anyway, it won't be difficult to find object oriented code crowded with classes some of which could be dispensed with. This happens wherever they don't resort to what the OO paradigm offers that other paradigms don't, namely, the threaded interaction between objects. Conversely, many functional programs can be found where the obsessive presence of accumulators and (side)effects suggests that programmers are writing functionally what they think imperatively.

Similar symptoms affect the area of formal modelling, in particular since formal notations are becoming available "in pairs" — a declarative notation plus an "object oriented" extension. Examples of this are VDM-SL/VDM++ [Fitzgerald and Larsen, 1998, Fitzgerald et al., 2005], Z and Z^{++}[Spivey, 1989, Lano, 1991], and Z and Object-Z [Smith, 2000], among others. How these notation pairs should be used is not always obvious. Because the OO extension supersedes the original notation, users often end up sticking to the former and ignoring the latter. Sooner or later, what could be elegantly modelled by state-less mathematic formulæ ends up being modelled over instance variable assignments and the like, in a way such that a little gap remains between the OO specification and a candidate reification.

In this paper we try to bridge the two worlds — the declarative and the object oriented — by discussing criteria under which a declarative specification in notation N can be *objectified* into notation N++. For a particular such pair of notations, VDM-SL/VDM++, we present a tool which animates such criteria in a way such that models written in VDM-SL are converted (*objectified*) to VDM++ objects. By *objectification* we mean stepwise addition of OO ingredients which are consistent with the original declarative specification. Of course, different ingredients will lead to different kinds of objectification.

Among several criteria for identifying potential classes for inclusion in a OO-model, Coad and Yourdon [Coad and Yourdon, 1991] suggest the following:

> *The potential class must have a set of identifiable operations that can change the value of its attributes in some way.*

This paper will focus on this particular dimension of object orientation. We will infer *class* hierarchies from flat, functional models or programs by identifying such attribute-changing operations. This identification of the instance variables in each class — the *internal state* of objects of that class — is thus a cornerstone of the *objectification* process. Formally the inferred objects behave like (generalizations of) Moore machines [1] which we regard as a particular family of coalgebras for a functor capturing a signature of attributes and methods [Rutten, 2000]. Once such a process is applied to a functional specification, reasoning about the inferred objects and their composition patterns can be done coinductively [Turi and Rutten, 1998, Vene, 2000]: *bisimulation* provides a powerful proof technique to establish behavioural equivalence. Later in the paper we introduce an algebra of Moore machines suitable for this purpose.

A few words on the paper's structure. Our starting point is an empirical study on the *objectification* of model-oriented specifications involving the VDM-SL/VDM++ [Fitzgerald and Larsen, 1998, Fitzgerald et al., 2005] pair of notations. Therefore, the first part of this paper presents a prototype system which embeds VDM-SL into VDM++ according to some formal criteria. In a sense, this suggests that models should be written first in VDM-SL and then lifted to VDM++, whereupon they can be enriched with OO specification details which could not be expressed algebraically in VDM-SL. So, there is a separation of concerns: static semantics first, at algebraic level, dynamic semantics later, at the coalgebraic one. Section 2 presents an overview of the proposed strategy and tool. Class identification and method inference are discussed in sections 3 and 4, respectively.

The second part of the paper starts with a discussion of what has been achieved and motivates the formalization of the inferred objects as (generalised) Moore machines, whose calculi is presented in section 6. The paper closes with section 7 which discuses related work and draws some conclusions which lead to plans for future work.

2 Overview of the objectification process

This section describes in a concrete way the proposed strategy for generating objects from functional specifications. Such a strategy is implemented by a tool (fully specified in [Cruz, 2004]) which converts VDM-SL functional models into object oriented VDM++ models. Concretely, a parser for VDM-SL supplies the abstract syntax tree (AST) of the source VDM-SL model specification to the objectifier. The main goal of the objectifier is to identity classes of objects in the AST of the VDM-SL functional specification, by transforming this specification into an object oriented specification in VDM++. Once a VDM++ AST is produced, the corresponding concrete syntax is then synthesized by an appropriate *pretty-printer* tool. This paper is concerned only with the intermediate AST transformation process.

Let us follow the overall strategy through an example. Assume that we are given the following functional model of a *Stack* of strings,

[1] In classical automata theory a *Moore* machine [Moore, 1966] is an automaton where each state is associated to an output symbol.

types

$Elem = char^*;$

$Stack = Elem^*;$

functions

$push : Stack \times Elem \rightarrow Stack$

$push\,(s, e) \triangleq$

$[e] \curvearrowright s;$

$pop : Stack \rightarrow Stack$

$pop\,(s) \triangleq$

tl s

pre $s \neq []$;

$top : Stack \rightarrow Elem$

$top\,(s) \triangleq$

hd s

pre $s \neq []$;

$empty : () \rightarrow Stack$

$empty\,() \triangleq$

$[];$

$isEmpty : Stack \rightarrow \mathbb{B}$

$isEmpty\,(s) \triangleq$

$s = [];$

which are regarded as *sheets* (or *pages*) in another functional model, that of a *Folder* [Barbosa and Oliveira, 2003]. This is defined as a couple of stacks

types

$Folder :: S1 : Stack$

$\qquad\quad S2 : Stack$

which are initially empty,

$new : () \rightarrow Folder$

$new\,() \triangleq$

mk-$Folder\,(empty\,(), empty\,())$

where one can insert or remove pages,

functions

$$
\begin{aligned}
& insert : Folder \times Elem \rightarrow Folder \\
& insert\,(f, e) \triangleq \\
& \quad \textsf{mk-}Folder\,(f.S1, push\,(f.S2, e)); \\
& remove : Folder \rightarrow Folder \\
& remove\,(f) \triangleq \\
& \quad \textsf{mk-}Folder\,(f.S1, pop\,(f.S2)) \\
& \textsf{pre}\ \ \neg\, isEmpty\,(f.S2)\ ;
\end{aligned}
$$

turn pages forwards or backwards,

$$
\begin{aligned}
& forward : Folder \rightarrow Folder \\
& forward\,(f) \triangleq \\
& \quad \textsf{let}\ x = top\,(f.S2)\ \textsf{in} \\
& \quad \textsf{mk-}Folder\,(push\,(f.S1, x), pop\,(f.S2)) \\
& \textsf{pre}\ \ \neg\, isEmpty\,(f.S2)\ ; \\
& backward : Folder \rightarrow Folder \\
& backward\,(f) \triangleq \\
& \quad \textsf{let}\ x = top\,(f.S1)\ \textsf{in} \\
& \quad \textsf{mk-}Folder\,(pop\,(f.S1), push\,(f.S2, x)) \\
& \textsf{pre}\ \ \neg\, isEmpty\,(f.S1)\ ;
\end{aligned}
$$

and read the current page:

$$
\begin{aligned}
& read : Folder \rightarrow Elem \\
& read\,(f) \triangleq \\
& \quad top\,(f.S2) \\
& \textsf{pre}\ \ \neg\, isEmpty\,(f.S2)\ ;
\end{aligned}
$$

The analysis of the signature of these algebraic models unveils four sorts *Folder*, *Stack*, *Elem* and $\mathbb{B}$ (the Booleans) — candidate to be promoted to classes — which are such that:

1. *Elem* never appears both as parameter and result of a function.
2. $\mathbb{B}$ never appears as parameter of a function.
3. *Stack* appears both as parameter and result of functions *push* and *pop*.
4. *Folder* appears both as parameter and result of several functions (*insert*, *forward*, *backward* and *remove*).

Functions such as *push*, *pop*, *insert*, *forward*, *backward* and *remove* are named *fold algebras* in functional programming terminology [Bird and Moor, 1997]. In general, a function with signature $B \times A \xrightarrow{\theta} A$ is called a *fold algebra*, because it can "fold" its computation cumulatively over type A:

$$b_0\,\theta\,(b_1\,\theta\,(b_2\,\theta\ldots))$$

Many programming schemata over inductive types are based on such algebras, e.g. in finite list processing. For example, the VDM-SL *elems* function can be specified as a *fold* whose recursive step is based on fold-algebra $b\,\theta\,a \triangleq a \cup \{b\}$.

Under a slight change in perspective, algebra θ can also be regarded as a *state-transition function* (vulg. *method*) — $a_1 = b\ \theta\ a_0$ describes the transition of an automaton which inputs b and steps from state a_0 to state a_1:

$$\textcircled{a_0} \xrightarrow{b} \textcircled{a_1}$$

From this viewpoint, inhabitants of datatype A are regarded as *states*, that is, instance variables in OO-terminology.

The identification of algebras in the original specification which — such as *push* and *pop* — have this alternative *coalgebraic meaning* [Rutten, 2000] is central to the objectification strategy described in this paper, whereby one obtains the "corresponding (obvious) methods". Such will be, in fact, the flavour of our objectification method, although things won't be as easy and immediate as that. We will also see that methods can be obtained by combining pairs of suitably typed functions, for instance

$$\langle top, pop\rangle : Stack \rightarrow Elem \times Stack \tag{1}$$

where the "split" combinator is such that $\langle f, g\rangle x = (f\ x, g\ x)$. A method POP with semantics $POP = \langle top, pop\rangle$ is indeed what an OO-programmer would write in the first place, instead of the two individual functions. These splits are known in the functional programming literature as instances of the *state transformer monad* (ST) [Wadler, 1990].

Of course, not every "split" of two functions will lead to a ST-instance: one of the functions is required to involve a "state-sort" both at argument and result level, in order for the combination of the two functions to enable input-dependent state updating and observation. In summary, what seems to be central to the problem is the identification of "state-sorts": these will be the sorts which will lead to classes under this objectification criterion.

In our example, sort $\mathbb{B}$ is primitive in VDM-SL and *Elem* is modelled by strings of character, which is also a primitive datatype. So, only sorts *Stack* and *Folder* are candidates to class-promotion. Even if *Elem* were not a primitive datatype and one had decided to promote it into a class, this would never see its state

$$\begin{tikzcd}
& \mathit{Class\text{-}Stack\text{-}Alg} & \\
\mathit{Class\text{-}Stack\text{-}Obj} \arrow[ur] & & \mathit{Class\text{-}Folder\text{-}Alg} \arrow[ul] \\
& & \mathit{Class\text{-}Folder\text{-}Obj} \arrow[u]
\end{tikzcd}$$

Figure 1: *Class hierarchy for the classes generated by the objectification of Folder.*

modified. For this reason, *Elem* shall be kept as a declarative type belonging to the class *Stack*.

One will, then, obtain two classes of objects, named *Class-Stack-Obj* and *ClassFolder-Obj* in Figure 1. Why are there two more classes (*Class-Stack-Alg* and *ClassFolder-Alg*) in the diagram of Figure 1 and why are they hierachically structured as they are? In the actual delivery of every such *-Obj* class, we have decided to keep the original functional API as a "library" class (the corresponding *-Alg* class), so that the newly generated operations (methods) are always defined over the original, homonym functions In other words, the *-Obj* class inherits from the corresponding *-Alg* class. For example, for function *insert* of the folder class to invoke *push* of the stack class, it is necessary that the former be defined as a subclass of the latter.

In this way, such classes act as VDM-SL modules "embedded" in VDM++ via the class mechanism of the latter. In summary, the objectification of the functional specification of *Folder* leads to four classes hierarchically related as illustrated in Figure 1. There are two purely functional classes that expose the hierarchic relationship between the original functional models of *Stack* and *Folder*, and two classes with state and public methods (termed *operations* in the VDM terminology) that provide the object oriented API of reactive "machines" *Stack* and *Folder*. As can be checked by inspecting the VDM++ code fragments below, *-Alg* classes only have functions, not operations. This is why we refer to them as being *(purely) functional*, or *declarative*. (See references [Fitzgerald and Larsen, 1998, Fitzgerald et al., 2005] for more details about the syntax and semantics of the VDM-SL and VDM++ notations.) In detail:

class *Class-Stack-Alg*
types
  public$\mathit{Elem} = \mathbb{N}$;
  public$\mathit{Stack} = \mathit{Elem}^*$

```
functions
public
    pop : Stack → Stack
    pop (s) ≜
      tl s;
public
    top : Stack → Elem
    top (s) ≜
      hd s;
public
    push : Stack × Elem → Stack
    push (s, el) ≜
      [el] ⁀ s;
public
    empty : () → Stack
    empty () ≜
      [];
public
    isEmpty : Stack → 𝔹
    isEmpty (s) ≜
      s = []
end Class-Stack-Alg
```

This class *Class-Stack-Alg* provides its functional *core* to class *Class-Stack-Obj*,

```
class Class-Stack-Obj is subclass of Class-Stack-Alg
instance variables
    public ST-Stack : Stack;
```

which, in turn, will offer the corresponding *object-oriented* API for the outside, dropping all *Stack* parameters from the original signatures as one would expect:

```
operations
public
    OP-pop : () -o-> ()
    OP-pop () ≜
      ST-Stack := pop (ST-Stack);
public
    OP-top : () -o-> Elem
    OP-top () ≜
      return top (ST-Stack);
```

public
$$OP\text{-}push : Elem \xrightarrow{o} ()$$
$$OP\text{-}push\,(el) \;\underline{\triangle}$$
$$\quad ST\text{-}Stack := push\,(ST\text{-}Stack, el);$$
public
$$OP\text{-}empty : () \xrightarrow{o} ()$$
$$OP\text{-}empty\,() \;\underline{\triangle}$$
$$\quad ST\text{-}Stack := empty\,();$$
public
$$OP\text{-}isEmpty : () \xrightarrow{o} \mathbb{B}$$
$$OP\text{-}isEmpty\,() \;\underline{\triangle}$$
return $isEmpty\,(ST\text{-}Stack)$
end $Class\text{-}Stack\text{-}Obj$

As far as *Folder* is concerned, class *Class-Folder-Alg* (Figure 1) inherits the functionality of *Class-Stack-Alg* as a (functional) library module:

class $Class\text{-}Folder\text{-}Alg$ is subclass of $Class\text{-}Stack\text{-}Alg$
types
$$Folder :: s1 : Stack$$
$$\qquad s2 : Stack$$
functions
public
$$new : () \rightarrow Folder$$
$$new\,() \;\underline{\triangle}$$
$$\quad \mathsf{mk}\text{-}Folder\,(empty\,(), empty\,());$$
public
$$insert : Folder \times Elem \rightarrow Folder$$
$$insert\,(f, el) \;\underline{\triangle}$$
$$\quad \mathsf{mk}\text{-}Folder\,(f.s1, push\,(f.s2, el));$$
public
$$remove : Folder \rightarrow Folder$$
$$remove\,(f) \;\underline{\triangle}$$
$$\quad \mathsf{mk}\text{-}Folder\,(f.s1, pop\,(f.s2));$$
public
$$backward : Folder \rightarrow Folder$$
$$backward\,(f) \;\underline{\triangle}$$
let $x = top\,(f.s1)$ in
$$\quad \mathsf{mk}\text{-}Folder\,(pop\,(f.s1), push\,(f.s2, x));$$

public
$\quad forward : Folder \rightarrow Folder$
$\quad forward\,(f) \triangleq$
$\qquad$ let $x = top\,(f.s2)$ in
$\qquad$ mk-$Folder\,(push\,(f.s1, x), pop\,(f.s2))$
end *Class-Folder-Alg*

Finally, class *Class-Folder-Obj* inherits the functionality of *Class-Folder-Alg* and *offers it* in an imperative way:

class *Class-Folder-Obj* is subclass of *Class-Folder-Alg*
instance variables
$\quad$ public $ST\text{-}Folder : Folder;$

operations
public
$\quad OP\text{-}new : () \xrightarrow{o} ()$
$\quad OP\text{-}new\,() \triangleq$
$\qquad ST\text{-}Folder := new\,();$
public
$\quad OP\text{-}insert : Elem \xrightarrow{o} ()$
$\quad OP\text{-}insert\,(e) \triangleq$
$\qquad ST\text{-}Folder := insert\,(ST\text{-}Folder, e);$
public
$\quad OP\text{-}remove : () \xrightarrow{o} ()$
$\quad OP\text{-}remove\,() \triangleq$
$\qquad ST\text{-}Folder := remove\,(ST\text{-}Folder);$
public
$\quad OP\text{-}backward : () \xrightarrow{o} ()$
$\quad OP\text{-}backward\,() \triangleq$
$\qquad ST\text{-}Folder := backward\,(ST\text{-}Folder);$
public
$\quad OP\text{-}forward : () \xrightarrow{o} ()$
$\quad OP\text{-}forward\,() \triangleq$
$\qquad ST\text{-}Folder := forward\,(ST\text{-}Folder)$
end *Class-Folder-Obj*

Figure 2 is an attempt to depict the process of building this class hierarchy. We proceed to the formalization of the intuitions behind this process in the section which follows.

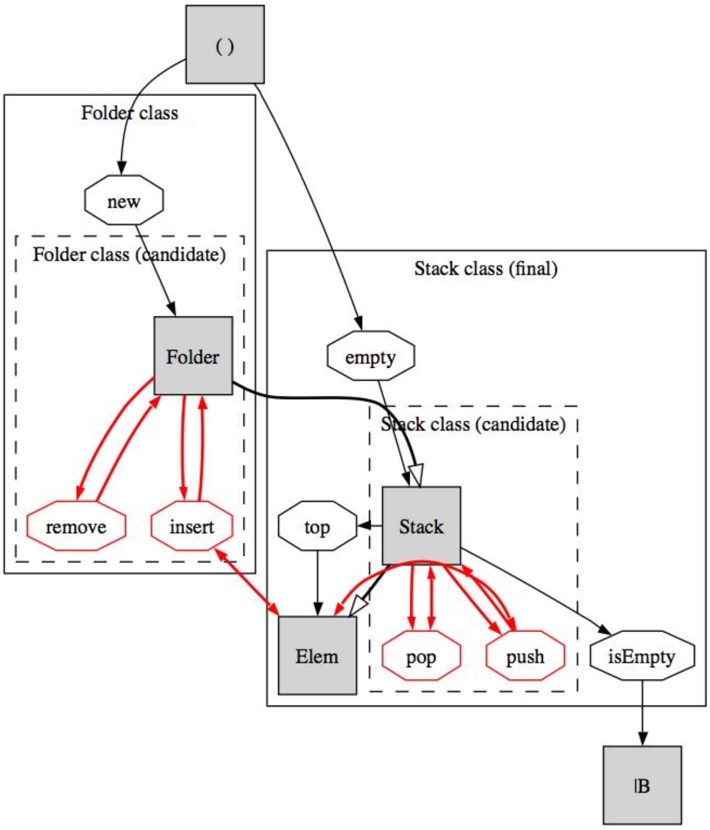


Figure 2: *Overview of objectification example. Filled boxes depict VDM types; bold arrows with open arrow heads depict type dependences; open octagons are functions; attribute-changing functions (fold algebras) are in grey; dashed clusters are candidates for promotion to class; solid line clusters are obtained classes.*

3 Objectification criteria for class generation

3.1 Class inference

A VDM-SL functional model can be divided in two main blocks — *abstract data type specification* and *function specification* [2]. Altogether, these form an *abstract model* $M = (S, M_S, M_\Omega)$ where

[2] We are deliberately ignoring the definition of *constant values* [Fitzgerald and Larsen, 1998] which are not relevant for our purposes.

– S is a set of (user-defined) sort symbols and Ω is a set of function symbols.

– $M_S : S \rightarrow \mathsf{T}S$ maps each sort to its definition, where $\mathsf{T}S$ is a polynomial (recursive) type describing the abstract syntax of VDM-SL type constructors. All VDM primitive types such as $\mathbb{N}$ and $\mathbb{B}$ are embedded into $\mathsf{T}S$, as well as S itself via the embedding $sort : S \rightarrow \mathsf{T}S$ [3].

– $M_\Omega : \Omega \rightarrow (\mathsf{T}S)^* \times \mathsf{T}S$ is a (heterogeneous) algebraic signature mapping every function symbol in Ω to its functionality.

Which members of S should be promoted to object classes is a decision which depends on the analysis of M as a whole: they should be *non primitive sorts which come equipped with fold-algebras*, that is, that appear simultaneously as parameter and result of one or more functions. This ensures the existence of updating functions essential to the future object's behaviour, by state transition.

Some notation will ease the formalization of the chosen *promotion criteria*. Everywhere the definition of sort $s \in S$ involves another sort $s' \in S$, we say that s directly depends on s' (or that s' participates in the construction of s) and write $s \triangleleft s'$. In our running example, we have $Folder \triangleleft Stack \triangleleft Elem$. Formally, we define $\triangleleft$ (the *direct dependence* relation) as follows,

$$s \triangleleft s' \equiv s' \in_{\mathsf{T}} (M_S \ s) \tag{2}$$

where *generic membership* $\in_{\mathsf{T}}$ extends traditional set-theoretic membership $\in_{\mathcal{P}}$ to "shape" T. Details about this extension can be found in e.g. [Hoogendijk, 1997, Oliveira and Rodrigues, 2004].

Let $s \in S$ be a sort symbol, P_s be the set of function symbols in Ω that have parameters involving sort s

$$P_s = \{f \in \Omega \mid \ s \in_{\mathsf{T}^*} \pi_1(M_\Omega \ f)\}$$

and R_s be a similar set concerning s as result:

$$R_s = \{f \in \Omega \mid \ \pi_2(M_\Omega \ f) = (sort \ s)\}$$

[3] A polynomial type is, roughly speaking, a type whose definition involves Cartesian product (written $\times$) and disjoint usion (written $+$), out of which other types (such as eg. finite lists, etc) can be built. See reference [Visser, 2003] for details about the polynomial definition of the abstract syntax of a given BNF grammar. In brief, the conversion is synthesised by the following table:

Bnf notation		Polynomial notation
$\alpha \mid \beta$	$\mapsto$	$\alpha + \beta$
$\alpha\beta$	$\mapsto$	$\alpha \times \beta$
α^*	$\mapsto$	α^*
α^+	$\mapsto$	$\alpha \times \alpha^*$
ϵ	$\mapsto$	1

The union of this two sets will be denoted by $\Omega_s = P_s \cup R_s$.

For every pair $s, s' \in S$, let $\Omega_{s \to s'}$ denote the set of all function symbols in Ω which have result sort s' and at least one parameter type involving sort s:

$$\Omega_{s \to s'} = P_s \cap R_{s'} \tag{3}$$

Let A_s denote the set of all fold-algebras over sort s found in Ω. In general, $A_s \subseteq \Omega_{s \to s} \subseteq \Omega_s$. We define predicate *classProm* s, meaning "sort s is class-promotable", as follows:

$$\begin{aligned}&classProm\ s \equiv \\ &\quad A_s \neq \{\} \vee \Omega_s \neq \{\} \wedge \langle \exists\ s' \neq s\ :\ s \lhd s' :\ classProm\ s' \rangle\end{aligned} \tag{4}$$

where $\{\}$ denotes the empty set. This definition rules out primitive types (which are not in S, by construction) and requires at least (a) one s-fold-algebra or (b) a function that has *s as parameter <u>or</u> result, in which case s directly depends on some other class-promotable s'.*

In our running example, *Folder* is class-promotable for two reasons: there are several *Folder*-fold-algebras and *Folder* $\lhd$ *Stack*, *Stack* being class-promotable on its own.

Class-promotable datatypes will serve to define the instance variables that form the state of the respective classes. Every datatype which fails such a promotion will have to be defined in some other way but, under what class? Criteria for assigning such datatypes to classes are needed.

Let C_s denote the class originating from a sort s such that *classProm* s holds. In our strategy, the datatypes to be defined inside class C_s are the following:

$$\begin{aligned}&\{s\} \cup \{s' \mid s \lhd s' \wedge \neg(classProm\ s')\} \\ &\quad \cup \{s' \mid s' \lhd s \wedge \neg(classProm\ s') \wedge \Omega_{s \to s'} \cup \Omega_{s' \to s} \neq \{\}\})\end{aligned}$$

That is to say, besides s itself, class C_s will declare

- the types that construct s and are not themselves promoted to class;
- all types s' that depend on s, are not themselves promoted to class and are such that there exists at least one function whose signature involves s and s', either as parameters or result.

In our example, *Stack* $\lhd$ *Elem* and *Elem* is not class-promotable. Thus *Elem* joins class C_{Stack}. Although *Folder* $\lhd$ *Stack*, *Folder* is class-promotable, so it won't join C_{Stack} .

3.2 Classifying the functionality

So far we have identified which VDM++ classes to generate and how to organize them hierarchically. Our strategy is still lacking criteria for identifying which functions and methods should be created in each class. The set of all functions to be declared in the context of class C_s is as follows:

$$A_s \cup \langle \bigcup s' \in S \ : \ (classProm\ s') \Rightarrow s \lhd s' : \ \Omega_{s' \to t} \cup \Omega_{s \to s'} \rangle$$

This means that every function involving a parameter or result s' different from s which is class-promotable can only be accepted in C_s if s' participates in the definition of s, in which case it is provided higher in the hierarchy.

Every function which is not accepted by any C_s, together with its parameter and result sorts, and all sorts shared by more than one C_s, form a top level library *S-Alg* (default system algebra), from which every *-Alg* class inherits.

Last but not least, we are ready to address criteria for inferring methods for class C_t out of the functions directly available or inherited by C_t under the strategy presented above.

4 Method inference

So far, the objectification process delivers a collection of classes that deploy an API similar to the original *functional* API, now with aditional operations *encapsulated* with the data structures over which they operate. The resulting API, may however be enhanced by combining state-changing functionality with operations which *query* that same state, leading to a more useful "Moore machine".

There are at least two ways of pairing such functions:

(a) Split a state-transformer function with a state-query function in the way suggested in section 2 via the *pop* and *top* example, recall (1). In this combination, the state is observed *before* being updated.

(b) First invoke the operation that changes the state and then that which observes it. As an example, consider cash withdrawing in an ATM machine: the receipt provided along with the money shows the account's balance *after* the subtraction of the required amount.

Let us see an example. Suppose that, upon objectification, the following functions

$$f : t_1 \times t_2 \times ... \times t_n \to t'$$
$$g : t_1 \times t_2 \times ... \times t_n \to t_1$$
$$h : t_1 \to t_1$$

are included in class C_{t_1}. Concerning pair f, g, criterion (a) above will offer the extra VDM++ method which follows

$F\_G : t_2 \times \ldots \times t_n \xrightarrow{o} t'$
$F\_G\,(x_2, \ldots, x_n) \triangleq$
  let $x = f\,(ST\text{-}t_1, x_2, \ldots, x_n)$ in
  ($ST\text{-}t_1 := g\,(ST\text{-}t_1, x_2, \ldots, x_n)$; return x)
pre $pre\text{-}g(ST\text{-}t_1, x_2, \ldots, x_n) \wedge pre\text{-}f(ST\text{-}t_1, x_2, \ldots, x_n)$;

and (b) will offer

$G\_F : t_2 \times \ldots \times t_n \xrightarrow{o} t'$
$G\_F\,(x_2, \ldots, x_n) \triangleq$ (
  $ST\text{-}t_1 := g\,(ST\text{-}t_1, x_2, \ldots, x_n)$; return $f\,(ST\text{-}t_1, x_2, \ldots, x_n)$)
pre $pre\text{-}g(ST\text{-}t_1, \ldots, x_n) \wedge pre\text{-}f(g(ST\text{-}t_1, \ldots, x_n), \ldots, x_n))$;

In attempting to pair f, h criterion (a) is excluded, but (b) still works, as the sequencing of the two operations doesn't require the two functions to have the same parameters:

$H\_F : t_2 \times \ldots \times t_n \xrightarrow{o} t'$
$H\_F\,(x_2, \ldots, x_n) \triangleq$ (
  $ST\text{-}t_1 := h\,(ST\text{-}t_1)$; return $f\,(ST\text{-}t_1, x_2, \ldots, x_n)$)
pre $pre\text{-}h(ST\text{-}t_1) \wedge pre\text{-}f(h(ST\text{-}t_1), x_2, \ldots, x_n))$;

Returning to the *Stack* example, the following methods will be offered for free in the final API for synthesized class *Class-Stack-Obj*:

public
$TopPop : () \xrightarrow{o} Elem$
$TopPop\,() \triangleq$
  let $x = top\,(ST\text{-}stack)$ in
  ($ST\text{-}stack := pop\,(ST\text{-}stack)$;
    return x
)
pre $ST\text{-}stack \neq []$

public
$PopTop : () \xrightarrow{o} Elem$
$PopTop\,() \triangleq$
  ($ST\text{-}stack := pop\,(ST\text{-}stack)$;
    return $top\,(ST\text{-}stack)$
)
pre $ST\text{-}stack \neq [] \wedge pop(ST\text{-}Stack) \neq []$;

5 Objects as Moore machines

The approach described in the previous sections starts from a *functional specification* to generate corresponding *OO classes* by the identification of *loci* of state. This is not, however, the unique way of introducing objects in a design. It rather corresponds to what one may call the *objectify as late as possible* alternative. Note that each OO class always inherits from an algebraic class which is its direct correspondent, every method tracing its origin to a specific functional service.

Between this conservative approach and the somewhat rasher *objectify from scratch* attitude, another alternative is to start just with the *functional specifications* which are intended to generate the basic objects in the design. But what does *basic* mean here? A suitable criteria would classify as *basic* modules declaring operations whose semantics do not result entirely from a combination of imported operation's calls. Intuitively *Class-Stack-Alg* is basic whereas *Class-Folder-Alg* is not. Once such basic algebras are objectified, complex objects will arise by *instance creation* and *object composition*.

This approach requires the introduction of (at least conceptual) mechanisms for *instance creation* and an algebra for *object composition*. Instance creation requires grouping all methods into a single coalgebra and a *naming* mechanism. For example, instances of class *Class-Stack-Obj* are given a name Stack and a semantics specified by a pair of functions which aggregate together *attributes* (*OP-top* and *OP-isEmpty*) and *methods* (*OP-pop*, *OP-push* and *OP-empty*):

$$
\begin{aligned}
o_{\mathsf{Stack}} &: \mathit{Stack} \longrightarrow \mathit{Elem} \times \mathbb{B} \\
a_{\mathsf{Stack}} &: \mathit{Stack} \times (\mathbf{1} + \mathit{Elem} + \mathbf{1}) \longrightarrow \mathit{Stack}
\end{aligned}
$$

where *Stack* is the sort of its internal state space (see *Class-Stack-Obj*) and $\mathbf{1} + \mathit{Elem} + \mathbf{1}$ are the types of the input parameters of the object methods mentioned above[4]. Note that attributes are aggregated by a *split* construction[5]

$$o_{\mathsf{Stack}} = \langle \mathit{OP\text{-}top}, \mathit{OP\text{-}isEmpty} \rangle$$

whereas methods are composed in an additive context through an *either* precomposed with the right distributivity law:

$$a_{\mathsf{Stack}} = [\mathit{OP\text{-}empty}, \mathit{OP\text{-}push}, \mathit{OP\text{-}pop}] \cdot \mathsf{dr}$$

[4] $\mathbf{1}$ denotes the singleton set whose unique element is, by convention, represented by $*$, and $+$ stands for datatype sum, *i.e.*, disjoint union of sets.

[5] A *split* is the canonical function to a cartesian product, *cf.*, equation (1). Dually an *either* $[f, g] : A + B \longrightarrow C$ is the canonical function from a coproduct or disjoint union of sets: either f or g is applied to argument $x : A + B$, depending on x coming from A or B.

These two functions correspond, respectively, to the *observer* and *action* part of a Moore machine: the split of o_{Stack} and (curried) a_{Stack}

$$\langle o_{\mathsf{Stack}}, \overline{a}_{\mathsf{Stack}} \rangle : \mathit{Stack} \longrightarrow (\mathit{Elem} \times \mathbb{B}) \times \mathit{Stack}^{\mathbf{1}+\mathit{Elem}+\mathbf{1}}$$

yields a coalgebra for functor $\mathsf{T} = O \times \mathsf{Id}^I$, for $O = \mathit{Elem} \times \mathbb{B}$ and $I = \mathbf{1} + \mathit{Elem} + \mathbf{1}$. I and O will be referred to in the sequel as the object *interface* types.

Modelling the generated objects by Moore machines provides immediately a proof principle (bisimulation) for proving observational equivalence. But we need more:

- First of all, the objectification process generates (typically) a number of different classes due to the possible identification of several *state-promotable sorts*. This corresponds to starting, on the functional side, with *heterogenous* (*i.e.*, multisorted) models. This entails the need for (well-behaved) ways of composing (concurrently) object instances of the generated classes. Such is the purpose of combinators $\boxplus$, $\boxtimes$ and $\circledast$ to be introduced in next section. In short, data *heterogeneity* at the functional level leads to *concurrent composition* of object instances.

- Furthermore, it is often the case that functions in the original specification involve more than one *state-promotable sort*. Such *cross-referencing*, which has no special meaning at the functional level, is realized by some form of *interaction* between the corresponding object instances.

- Finally, declarative specifications need not be purely functional, but may allow themselves to follow a particular *behavioural pattern* (such as, for example *partiality* or *non determinism*), which moreover is intended to be preserved by composition.

The way we propose to take such behavioural effect into account is by introducing in the coalgebra signature a strong monad B [6] leading to the following re-definition of funtor T:

$$\mathsf{T} \;=\; O \times \mathsf{B}^I \tag{5}$$

[6] A *strong monad* is a monad $\langle \mathsf{B}, \eta, \mu \rangle$ where B is a strong functor[Kock, 1972]. B being strong means there exist natural transformations $\tau_r^{\mathsf{T}} : \mathsf{T} \times \text{-} \Longrightarrow \mathsf{T}(\mathsf{Id} \times \text{-})$ and $\tau_l^{\mathsf{T}} : \text{-} \times \mathsf{T} \Longrightarrow \mathsf{T}(\text{-} \times \mathsf{Id})$, called the right and left strength, respectively, subject to certain conditions. Their effect is to *distribute* the free variable values in the context "-" along functor B. Strength τ_r, followed by τ_l maps $\mathsf{B}I \times \mathsf{B}J$ to $\mathsf{BB}(I \times J)$, which can, then, be flattened to $\mathsf{B}(I \times J)$ via μ. In most cases, however, the *order* of application is relevant for the outcome. The Kleisli composition of the right with the left strength, gives rise to a natural transformation whose component on objects I and J is given by $\delta_r \;=\; \tau_{r_{I,J}} \bullet \tau_{l_{\mathsf{B}I,J}}$ Dually, $\delta_l = \tau_{l_{I,J}} \bullet \tau_{r_{I,\mathsf{B}J}}$. Such transformations specify how the monad distributes over product and, therefore, represent a sort of sequential composition of B-computations.

Generalized Moore machines are formally represented as coalgebras for this functor, given by the split of two functions

$$\langle o_p, \overline{a_p} \rangle : U_p \longrightarrow O \times (\mathsf{B}\ U_p)^I$$

where $o_p : U_p \longrightarrow O$ is the *attribute* and $a_p : U_p \times I \longrightarrow \mathsf{B}U_p$ is a state update function corresponding to the aggregation of methods. Typically, O instantiates to a Cartesian product $\prod_{x \in X} O_x$ of different, but simultaneously available, observers, whereas I takes the form of a sum $\sum_{y \in Y} I_y$ of (state update, non interfering) operations. If Y is regarded as a set of action names, I_y denotes the type of the argument of operation y.

Such coalgebras need to be *pointed*, *i.e.*, an initial state value $u_p \in U_p$ has to be provided, corresponding to the default state value assigned at object creation. This acts as a *seed* for object behaviour. In algebraic specifications this is often ommitted, meaning that any value of U_p could be chosen; sometimes, however, they are given by constant specifications (*e.g.*, the empty list for the stack example) or a suitable predicate. Then,

Definition 1. For interface types I and O, a Moore machine (generalized wrt effect B) is specified as a (pointed) coalgebra

$$p \quad = \quad \langle \gamma_p : U_p \longrightarrow \mathbf{2}, \langle o_p : U_p \longrightarrow O, \overline{a_p} : U_p \longrightarrow \mathsf{B}U_p^I \rangle \rangle \tag{6}$$

where γ_p is referred to as the *seed predicate*.

The section which follows introduces the definition of some combinators, all of them parametric on B, and a fragment of the resulting calculus. The combinators include *pipeline*, *i.e.*, a generalization of functional composition over objects with disjoint state spaces running independently; *hook*, *i.e.*, a partial pipeline in which only specified pairs of attributes/methods are joined; *wrapping*, used to costumize object interfaces (for *e.g.*, to hide or to replicate a particular method) and several forms of *parallel* composition.

For the moment, however, let us go back to our *Stack/Folder* example and remark that at least two ways of building a *folder* from two *stacks* pop out from the same algebraic specification. They are shown as two possible paths in the diagram of Figure 3, where different types of arrows stand for alternative engineering procedures.

– Following the path on the right, we proceed by importing *Class-Stack-Alg* into another algebraic specification (*Class-Folder-Alg*) which is then objectified. A *folder* is then an instance FolderO of this class obtained by direct objectification of the corresponding algebra. This corresponds to the VDM objectification exercise presented in the first part of this paper.

$$\begin{tikzcd}
 & \mathit{Class\text{-}Stack\text{-}Alg} \arrow[dl, Rightarrow, "\mathit{Objectify}"'] \arrow[dr, dotted, "\mathit{Import}"] & \\
\mathit{Class\text{-}Stack\text{-}Obj} \arrow[d, "\mathit{CreateInstance}"'] & & \mathit{Class\text{-}Folder\text{-}Alg} \arrow[d, Rightarrow, "\mathit{Objectify}"] \\
\mathsf{L.Stack}, \mathsf{R.Stack} \arrow[d, rightsquigarrow, "\mathit{ObjectCompose}"'] & & \mathit{Class\text{-}Folder\text{-}Obj} \arrow[d, "\mathit{CreateInstance}"] \\
\mathsf{FolderC} & & \mathsf{FolderO}
\end{tikzcd}$$

Figure 3: *Alternative paths to objectification.*

– The alternative path on the left brings objectification earlier to the process and produces a OO class of *stacks* (*Class-Stack-Obj*). A *folder* is built by creating two instances L.Stack and R.Stack of this class, modelling respectively the left and right piles in the folder. They are then appropriately combined, resorting to a calculus to be presented in the sequel.

An interesting question concerns the relationship between objects FolderO (after *folder by objectification*) and FolderC (after *folder by composition*). Actually, as shown in sub-section 6.2, they correspond to *bisimilar* Moore machines. In the general case, however, one may act as a *refinement* [Meng and Barbosa, 2004] of the other. Before dwelling in these issues, let us review briefly the underlying object calculus.

6 A Calculus of (generalised) Moore machines

6.1 Object combinators

The algebra of generalised Moore machines is expressive enough to capture several object instance composition patterns. In the sequel we introduce a number of combinators and (a fragment of the) calculus which enables compositional reasoning about object-oriented designs. Such laws are typically equations stated in terms of observational equivalence [7].

For this purpose we first need a notion of morphism. Fortunately the general theory of coalgebras provides such a notion: a morphism relating two coalgebras

[7] Refinement laws, witnessing object simulation, have been studied in [Meng and Barbosa, 2004], although in the broader context of coalgebraic modelling.

is just a function between their state spaces with commutes with the coalgebra dynamics. For Moore machines $h : p \longrightarrow q$ is a morphism from p to q iff

$$(O \times \mathsf{B}h^{I}) \cdot p \;=\; q \cdot h \tag{7}$$

which can be re-written into

$$o_p = o_q \cdot h \;\wedge\; \mathsf{B}h \cdot a_p = a_q \cdot (h \times \mathsf{id}) \tag{8}$$

Additionally the following preservation condition for seeds should hold

$$\gamma_p \;=\; \gamma_q \cdot h \tag{9}$$

A notion of a morphism entails for free a notion of equivalence. Again a general result in the coalgebra theory asserts that the existence of a morphism between two Moore machines is enough to prove they are *bisimilar*. This provides a proof technique for the Moore calculus: proofs are presented in the *Bird-Merteens* pointfree calculation style [Bird and Moor, 1997] popularised in the functional programming community, and therefore do not rely on the explicit construction of bisimulations (as done in mainstream coalgebra literature).

Our first combinator is *pipelining*: two objects p and q are placed side by side, connecting the output attribute of p to the input of q action. Formally,

Definition 2. Pipeline $p\,;\,q : I \longrightarrow O$ is given [8] by

$$p\,;\,q \;=\; \langle \gamma_{p;q} : U_p \times U_q \longrightarrow \mathbb{B}, \langle o_{p;q}, \overline{a}_{p;q} \rangle \rangle \tag{10}$$

$$
\begin{aligned}
o_{p;q} &= U_p \times U_q \xrightarrow{\pi_2} U_q \xrightarrow{o_q} O \\
a_{p;q} &= U_p \times U_q \times I \xrightarrow{\mathsf{xr}} U_p \times I \times U_q \xrightarrow{a_p \times \mathsf{id}} \mathsf{B}\,U_p \times U_q \xrightarrow{\tau_r} \mathsf{B}(U_p \times U_q) \\
&\quad \xrightarrow{\mathsf{B}\langle \mathsf{id}, o_p \cdot \pi_1 \rangle} \mathsf{B}(U_p \times U_q \times K) \xrightarrow{\mathsf{Ba}} \mathsf{B}(U_p \times (U_q \times K)) \\
&\quad \xrightarrow{\mathsf{B}(\mathsf{id} \times a_q)} \mathsf{B}(U_p \times \mathsf{B}\,U_q) \xrightarrow{\mathsf{B}\tau_l} \mathsf{BB}(U_p \times U_q) \xrightarrow{\mu} \mathsf{B}(U_p \times U_q) \\
o_{p;q} &= \wedge \cdot (\gamma_p \times \gamma_q)
\end{aligned}
$$

Clearly, sequential composition is associative, *i.e.*, for p, q and r suitably typed

$$(p\,;\,q)\,;\,r \;\sim\; p\,;\,(q\,;\,r) \tag{11}$$

To investigate the existence of units for $;$ requires a prior introduction of a mechanism for representing functions as objects which is done through a sort of '*mirror* mechanism':

[8] The definition makes use of a few natural isomorphisms, namely $\mathsf{a} : (A \times B) \times C \longrightarrow A \times (B \times C)$ (associativity), $\mathsf{s} : A \times B \longrightarrow B \times A$ (commutativity) and $\mathsf{xr} : A \times B \times C \longrightarrow A \times C \times B$ (defined by $\mathsf{xr} = \mathsf{a}^{\circ} \cdot (\mathsf{id} \times \mathsf{s}) \cdot \mathsf{a}$.

Definition 3. A function $f : A \longrightarrow B$ is represented by

$$\ulcorner f \urcorner \;=\; \langle b \in B, \langle \mathsf{id}_B, \overline{\eta_B \cdot f \cdot \pi_2} \rangle\rangle$$

i.e., by a coalgebra $B \longrightarrow B \times (\mathsf{B}B)^A$.

Note that, in general, the representation of f is not unique. Even worse, the possible representations are not bisimilar. We shall come back to this soon. For the moment, consider

$$\mathsf{copy}_K \;=\; \ulcorner \mathsf{id}_K \urcorner$$

Is this a unit for sequential composition? Let $p : I \longrightarrow O$ be a component. We want to discuss whether equations

$$\mathsf{copy}_I \,;\, p \;\sim\; p \tag{12}$$

$$p \,;\, \mathsf{copy}_O \;\sim\; p \tag{13}$$

hold. The obvious choice of morphisms to witness bisimulations in equations (12) and (13) is, respectively, $\pi_2 : I \times U_p \longrightarrow U_p$ and $\pi_1 : U_p \times O \longrightarrow U_p$. It is easy to prove that these morphisms commute with the action part of the respective objects. For the observers, however, one has:

$$o_{\mathsf{copy}_I ; p} \;=\; o_p \cdot \pi_2$$

but

$$o_{p;\mathsf{copy}_O} \;=\; o_{\mathsf{copy}_O} \cdot \pi_2 \;\neq\; o_p \cdot \pi_1$$

Anyway, should both $p \,;\, \mathsf{copy}_O$ and p be observed after action takes place, the result would be the same, because the 'expected' value would be already stored in the state of copy_O. This leads to the following definition of a *next morphism.*

Definition 4. A *next morphism* from $p : I \longrightarrow O$ to $q : I \longrightarrow O$ is a function $h : U_p \longrightarrow U_q$ such that

$$\mathsf{B}o_p \cdot a_p \;= \mathsf{B}o_q \cdot a_q \cdot (h \times \mathsf{id}) \tag{14}$$

and

$$\mathsf{B}h \cdot a_p \;=\; a_q \cdot (h \times \mathsf{id}) \tag{15}$$

Two components p and q are *next bisimilar*, written as $p \overset{\bullet}{\sim} q$ iff there is a seed preserving next morphism $h : p \longrightarrow q$ relating them.

Lemma 5. *The composition of next morphisms is a next morphism. Moreover, every morphism is also a next morphism.*

Proof. Let $h : p \longrightarrow q$ and $k : q \longrightarrow r$ be next morphisms. Then,

$$\begin{array}{cl} & a_r \cdot ((k \cdot h) \times \mathsf{id}) \\ = & \{ \ \times \text{ functor and } k \text{ is a next morphism} \ \} \\ & \mathsf{B}k \cdot a_q \cdot (h \times \mathsf{id}) \\ = & \{ \ \mathsf{B} \text{ functor and } h \text{ is a next morphism} \ \} \\ & \mathsf{B}(k \cdot h) \cdot a_p \end{array}$$

and

$$\begin{array}{cl} & \mathsf{B}o_p \cdot a_p \\ = & \{ \ h \text{ is a next morphism} \ \} \\ & \mathsf{B}o_q \cdot a_q \cdot (h \times \mathsf{id}) \\ = & \{ \ k \text{ is a next morphism} \ \} \\ & \mathsf{B}o_r \cdot a_r \cdot (k \times \mathsf{id}) \cdot (h \times \mathsf{id}) \\ = & \{ \ \times \text{ functor} \ \} \\ & \mathsf{B}o_r \cdot a_r \cdot ((k \cdot h) \times \mathsf{id}) \end{array}$$

We shall now prove that every morphism is a next morphism: if $h : p \longrightarrow q$ is a morphism it already meets (15). It furthermore satisfies (14) because

$$\begin{array}{cl} & \mathsf{B}o_p \cdot a_p \\ = & \{ \ h \text{ is a morphism (attribute condition) and } \mathsf{B} \text{ functor} \ \} \\ & \mathsf{B}o_q \cdot \mathsf{B}h \cdot a_p \\ = & \{ \ h \text{ is a morphism (action condition)} \ \} \\ & \mathsf{B}o_q \cdot a_q \cdot (h \times \mathsf{id}) \end{array}$$

□

We may thus conclude that $p \,;\, \mathsf{copy}_O \stackrel{\bullet}{\sim} p$ holds. Therefore, Moore machines and next morphisms form a category. Instead of going deep into the theorectical details (the reader is referred to [Barbosa, 2001]) we prefer to unveil a bit more of the envisaged calculus. The result which follows, whose proof is included to illustrate the pointfree reasoning style mentioned above [9], shows that pipelining is actually the object extension of functional composition.

Lemma 6. *Let $g : I \longrightarrow K$ and $f : K \longrightarrow O$ be functions. Then,*

$$\ulcorner f \cdot g \urcorner \sim \ulcorner g \urcorner \,;\, \ulcorner f \urcorner \tag{16}$$

Proof. Equation (16) will be proved by checking that $\pi_2 : K \times O \longrightarrow O$ is a morphism from $\ulcorner g \urcorner ; \ulcorner f \urcorner$ to $\ulcorner f \cdot g \urcorner$. For the observers part note that

$$o_{\ulcorner g \urcorner ; \ulcorner f \urcorner} = \pi_2 \cdot \mathsf{id}_O = o_{\ulcorner f \cdot g \urcorner}$$

[9] This proof, as any other one in the calculus, is parametric on monad B. Such a genericity would be very difficult to express in a conventional pointwise proof style.

holds. On the other hand,

$$
\begin{array}{ll}
& \mathsf{B}\pi_2 \cdot a_{\ulcorner g \urcorner ; \ulcorner f \urcorner} \\
= & \quad \{ \ ; \text{ and function lifting definitions} \ \} \\
& \mathsf{B}\pi_2 \cdot \mu \cdot \mathsf{B}\tau_l \cdot \mathsf{B}(\mathsf{id} \times (\eta \cdot f \cdot \pi_2)) \cdot \mathsf{Ba} \cdot \mathsf{B}\langle \mathsf{id}, \pi_1 \rangle \cdot \tau_r \cdot ((\eta \cdot g \cdot \pi_2) \times \mathsf{id}) \cdot \mathsf{xr} \\
= & \quad \{ \ \eta \text{ is a strong natural transformation: } \tau_l \cdot (\mathsf{id} \times \eta) = \eta \ \} \\
& \mathsf{B}\pi_2 \cdot \mu \cdot \eta \cdot \mathsf{B}(\mathsf{id} \times (f \cdot \pi_2)) \cdot \mathsf{Ba} \cdot \mathsf{B}\langle \mathsf{id}, \pi_1 \rangle \cdot \tau_r \cdot ((\eta \cdot g \cdot \pi_2) \times \mathsf{id}) \cdot \mathsf{xr} \\
= & \quad \{ \ \text{monad axioms: } \mu \cdot \eta = \mathsf{id} \ \} \\
& \mathsf{B}\pi_2 \cdot \mathsf{B}(\mathsf{id} \times (f \cdot \pi_2)) \cdot \mathsf{Ba} \cdot \mathsf{B}\langle \mathsf{id}, \pi_1 \rangle \cdot \tau_r \cdot ((\eta \cdot g \cdot \pi_2) \times \mathsf{id}) \cdot \mathsf{xr} \\
= & \quad \{ \ \text{routine: } (\mathsf{id} \times \pi_2) \cdot \mathsf{a} = \pi_1 \times \mathsf{id} \ \} \\
& \mathsf{B}\pi_2 \cdot \mathsf{B}(\mathsf{id} \times f) \cdot \mathsf{B}(\pi_1 \times \mathsf{id}) \cdot \mathsf{B}\langle \mathsf{id}, \pi_1 \rangle \cdot \tau_r \cdot ((\eta \cdot g \cdot \pi_2) \times \mathsf{id}) \cdot \mathsf{xr} \\
= & \quad \{ \ \times \text{ absorption: } (f \times g) \cdot \langle h, k \rangle = \langle f \cdot h, g \cdot k \rangle \ \} \\
& \mathsf{B}\pi_2 \cdot \mathsf{B}\langle \pi_1, f \cdot \pi_1 \rangle \cdot \tau_r \cdot ((\eta \cdot g \cdot \pi_2) \times \mathsf{id}) \cdot \mathsf{xr} \\
= & \quad \{ \ \times \text{ cancellation: } \pi_1 \cdot \langle f, g \rangle = f \text{ and } \pi_2 \cdot \langle f, g \rangle = g \ \} \\
& \mathsf{B}f \cdot \mathsf{B}\pi_1 \cdot \tau_r \cdot ((\eta \cdot g \cdot \pi_2) \times \mathsf{id}) \cdot \mathsf{xr} \\
= & \quad \{ \ \eta \text{ is a strong natural transformation: } \tau_r \cdot (\eta \times \mathsf{id}) = \eta \ \} \\
& \mathsf{B}f \cdot \mathsf{B}\pi_1 \cdot \eta \cdot (g \times \mathsf{id}) \cdot (\pi_2 \times \mathsf{id}) \cdot \mathsf{xr} \\
= & \quad \{ \ \eta \text{ is natural: } \eta \cdot f = \mathsf{B}f \cdot \eta \ \} \\
& \eta \cdot f \cdot \pi_1 \cdot (g \times \mathsf{id}) \cdot (\pi_2 \times \mathsf{id}) \cdot \mathsf{xr} \\
= & \quad \{ \ \times \text{ cancellation} \ \} \\
& \eta \cdot f \cdot g \cdot \pi_1 \cdot (\pi_2 \times \mathsf{id}) \cdot \mathsf{xr} \\
= & \quad \{ \ \text{routine: } \pi_1 \cdot (\pi_2 \times \mathsf{id}) \cdot \mathsf{xr} = \pi_2 \cdot (\pi_2 \times \mathsf{id}) \ \} \\
& \eta \cdot f \cdot g \cdot \pi_2 \cdot (\pi_2 \times \mathsf{id}) \\
= & \quad \{ \ \text{function lifting definition} \ \} \\
& \ulcorner f \cdot g \urcorner \cdot (\pi_2 \times \mathsf{id})
\end{array}
$$

□

Directly connected to function lifting is the *wrapping* combinator which encapsulates objects's pre- and post-composition with lifted functions. Formally,

Definition 7. For an object $p : I \longrightarrow O$ and functions $f : I' \longrightarrow I$, $g : O \longrightarrow O'$, define

$$\text{-}[f, g] : \mathsf{Cp}(I, O) \longrightarrow \mathsf{Cp}(I', O')$$

mapping $\langle \gamma_p, \langle o_p, \overline{a}_p \rangle \rangle$ into $\langle \gamma_p, \langle o_{p[f,g]}, \overline{a}_{p[f,g]} \rangle \rangle$, where

$$
\begin{aligned}
o_{p[f,g]} &= U_p \xrightarrow{o_p} O \xrightarrow{g} O' \\
a_{p[f,g]} &= U_p \times I' \xrightarrow{\mathsf{id} \times f} U_p \times I \xrightarrow{a_p} \mathsf{B}U
\end{aligned}
$$

The following result can be established in terms of *next bisimilarity*:

Lemma 8. *For any machine* $p : I \longrightarrow O$ *and functions* $f : I' \longrightarrow I$, $g : O \longrightarrow O'$, $f' : J \longrightarrow I'$ *and* $g' : O' \longrightarrow R$,

$$p[f, g] \overset{\bullet}{\sim} \ulcorner f \urcorner \,;\, p \,;\, \ulcorner g \urcorner \tag{17}$$

$$p[f, g][f', g'] \overset{\bullet}{\sim} p[f \cdot f', g' \cdot g] \tag{18}$$

Object aggregation is captured by two tensor products: $\boxtimes$, which corresponds to *synchronous product*, and $\boxplus$, which captures a notion of *choice* between the available methods. The composite object $p \boxtimes q$ executes simultaneously both actions and offers the product of both attributes. Formally,

Definition 9. Let $p : I \longrightarrow O$ and $q : J \longrightarrow R$. Then,

$$p \boxtimes q = \langle \gamma_{p \boxtimes q}, \langle o_{p \boxtimes q}, \overline{a}_{p \boxtimes q} \rangle \rangle$$

where

$$\gamma_{p \boxtimes q} = U_p \times U_q \xrightarrow{\gamma_p \times \gamma_q} \mathbf{2} \times \mathbf{2} \xrightarrow{\wedge} \mathbf{2}$$

$$o_{p \boxtimes q} = U_p \times U_q \xrightarrow{o_p \times o_q} O \times R$$

and

$$a_{p \boxtimes q} = U_p \times U_q \times (I \times J) \xrightarrow{\cong} U_p \times I \times (U_q \times J) \xrightarrow{a_p \times a_q} \mathsf{B} U_p \times \mathsf{B} U_q \xrightarrow{\delta_l} \mathsf{B}(U_p \times U_q)$$

On the other hand, composition $p \boxplus q$, although also yielding the product of attributes, behaves either as p or q depending on input. Formally,

Definition 10. The *choice* combinator is defined as

$$p \boxplus q = \langle \gamma_{p \boxplus q}, \langle o_{p \boxplus q}, \overline{a}_{p \boxplus q} \rangle \rangle$$

where

$$\gamma_{p \boxplus q} = U_p \times U_q \xrightarrow{\gamma_p \times \gamma_q} \mathbf{2} \times \mathbf{2} \xrightarrow{\wedge} \mathbf{2}$$

$$o_{p \boxplus q} = U_p \times U_q \xrightarrow{o_p \times o_q} O \times R$$

$$\begin{aligned} a_{p \boxplus q} = {} & U_p \times U_q \times (I + J) \xrightarrow{\cong} U_p \times I \times U_q + U_p \times (U_q \times J) \\ & \xrightarrow{a_p \times \mathsf{id} + \mathsf{id} \times a_q} \mathsf{B} U_p \times U_q + U_p \times \mathsf{B} U_q \\ & \xrightarrow{\tau_r + \tau_l} \mathsf{B}(U_p \times U_q) + \mathsf{B}(U_p \times U_q) \xrightarrow{\nabla} \mathsf{B}(U_p \times U_q) \end{aligned}$$

where $\nabla = [\mathsf{id}, \mathsf{id}]$ is the codiagonal function.

Both combinators are associative, commutative whenever B is a commutative monoid and commute with pipeline. The unit of $\boxtimes$ is object $\mathsf{idle} = \ulcorner \mathsf{id}_1 \urcorner$ (the lifting of the identity) whereas the unit of $\boxplus$ is $\mathsf{nil} = \ulcorner !_\emptyset \urcorner$ (the lifting of the unique function from $\emptyset$ to $\mathbf{1}$). As one would expect the unit for $\boxplus$ acts a zero element for $\boxtimes$.

Concurrent composition, in which objects execute either simultaneous or independentely, is captured by yet another tensor $\circledast$ which combines both $\boxtimes$ and $\boxplus$. Formally,

Definition 11. Let $p : I \longrightarrow O$ and $q : J \longrightarrow R$. Then,

$$p \circledast q = \langle \gamma_{p \boxtimes q}, \langle o_{p \boxtimes q}, \overline{a}_{p \boxtimes q} \rangle \rangle$$

where

$$\begin{aligned} \gamma_{p \circledast q} &= \gamma_{p \boxtimes q} \\ o_{p \circledast q} &= o_{p \boxtimes q} \\ a_{p \circledast q} &= U_p \times U_q \times (I + J) + I \times J \xrightarrow{\cong} \\ &\quad U_p \times U_q \times (I + J) \; + \; U_p \times U_q \times (I \times J) \\ &\quad \xrightarrow{a_{p \boxplus q} + a_{p \boxtimes q}} \mathsf{B}(U_p \times U_q) + \mathsf{B}(U_p \times U_q) \xrightarrow{\nabla} \mathsf{B}(U_p \times U_q) \end{aligned}$$

Object interaction is dealt through a family of *hook* combinators which allow a value read in an attribute to be fed back as the argument of a method.

Definition 12. Let $p : I \longrightarrow O$ be a component and assume $I = \sum_{y \in Y} I_y$, $O = \prod_{x \in X} O_x$ such that $I_i = J_j$, for indices i and j. The *hook* combinator on position i, j is defined by

$$p \urcorner_{i,j} = \langle \gamma_p, \langle o_{p \urcorner_{i,j}}, \overline{a}_{p \urcorner_{i,j}} \rangle \rangle$$

where

$$\begin{aligned} o_{p \urcorner_{i,j}} &= o_p \\ a_{p \urcorner_{i,j}} &= U_p \times (I + \mathbf{1}) \xrightarrow{\cong} U_p \times I + U_p \xrightarrow{a_p + \Phi_{i,j}} \mathsf{B}U_p + \mathsf{B}U_p \xrightarrow{\nabla} \mathsf{B}U_p \end{aligned}$$

where

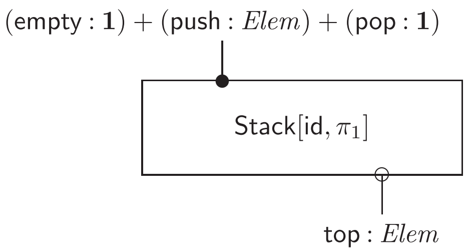
$$\Phi_{i,j} = U_p \xrightarrow{\langle \mathsf{id}, o_p \rangle} U_p \times O \xrightarrow{\mathsf{id} \times \pi_j} U_p \times O_j \xrightarrow{\mathsf{id} \times i_i} U_p \times I \xrightarrow{a_p} \mathsf{B}U_p$$

Reference [Barbosa, 2001] proves a number of laws for these combinators. For example, it is easy to show that pipelining is a special (total) case of hook. The calculus also provides other combinators (and corresponding laws): for example a *delay* combinator δp which replicates the state space of p so that one copy of U_p always maintains the previous state value. It also includes a proviso for representing autonomous *internal* (hidden) activity within the calculus.

Due to space limitations we shall now close this glimpse over the calculus and proceed to applying it to build the FolderC object mentioned in Figure 3. Our main target is to compare this with the result of the objectification process applied to the *Folder* declarative specification which leads to FolderO, as discussed in the first part of this paper.

6.2 The Folder example

This subsection discusses how two instances of *Class-Stack-Obj* can be assembled together to yield a Folder object. Our starting point are two copies L.Stack and R.Stack of a *Class-Stack-Obj* instance Stack wrapped by $[\mathsf{id}, \pi_1]$ to hide observer *OP-isEmpty*[10] which plays no role in the folder specification. For readers' convenience object's input/output signatures are shown decorated with the names of the corresponding methods or attributes, as in



Our next step is to aggregate both stack instances concurrently, via $\boxast$, and wrap the composite on both input and output sides, as follows. Let

$$I_{\mathsf{L.Stack}} = \mathsf{L.empty} : \mathbf{1} + \mathsf{L.push} : Elem + \mathsf{L.pop} : \mathbf{1}$$

denote the input interface of instance L.Stack, obtained by prefixing with instance identifier L. all the original methods, and assume a similar convention for R.Stack. According to Def. 11, the input interface of L.Stack $\boxast$ R.Stack is $I_{\mathsf{L.Stack} \boxast \mathsf{R.Stack}} = (I_{\mathsf{L.Stack}} + I_{\mathsf{R.Stack}}) + I_{\mathsf{L.Stack}} \times I_{\mathsf{R.Stack}}$, which makes far more methods available than the ones required for building a folder object [11]. Therefore, embedding

$$\begin{aligned} \mathsf{sel} : (\mathsf{L.empty}, \mathsf{R.empty}) : \mathbf{1} \times \mathbf{1} + \mathsf{R.pop} : \mathbf{1} + \mathsf{R.push} : Elem\, + \\ + (\mathsf{L.pop}, \mathsf{R.push}) : \mathbf{1} \times Elem + (\mathsf{L.push}, \mathsf{R.pop}) : Elem \times \mathbf{1} \\ \longrightarrow I_{\mathsf{L.Stack} \boxast \mathsf{R.Stack}} \end{aligned}$$

[10] In the sequel, method and attribute identifiers *OP-mth* (as in *OP-isEmpty*) will be replaced by the more concise notation mth (as in isEmpty).

[11] For example, joint method (L.new, R.pop) plays no role in the assembly.

selects the relevant methods.

Interaction between the two stacks, as required to implement the folder's forward method, is specified by feeding back attribute L.top as an argument to method R.push. A similar combination of R.top and L.push yields method backward[12]. This is, however, only part of the picture: the correct implementation of forward requires that L.pop is also simultaneously performed. We will resort to the hook combinator in order to connect the joint method $(\mathsf{L.pop}, \mathsf{R.push})$ to attribute L.top, paired with a dummy parameter $* : \mathbf{1}$ which acts as an input for L.pop. A similar scheme applies to the implementation of backward. Such output extensions are performed by wrapping the composite output with isomorphism

$$\mathsf{ext} : \mathsf{L.top} : Elem \times \mathsf{R.top} : Elem \longrightarrow (\mathbf{1} \times Elem) \times (Elem \times \mathbf{1})$$

By puting everything together, and keeping in mind that methods forward and backward are realized by feeding back the first (respectively, second) output factor into the fourth (respectively, fifth) input additive component, one obtains

$$(\mathsf{L.Stack} \boxast \mathsf{R.Stack})[\mathsf{sel}, \mathsf{ext}] \Lsh_{4,1\text{-}5,2}$$

which is of type

$$\mathbf{1} \times \mathbf{1} + Elem + \mathbf{1} + \mathbf{1} \times Elem + Elem \times \mathbf{1} + \mathbf{1} + \mathbf{1} \quad \longrightarrow \quad (\mathbf{1} \times Elem) \times (Elem \times \mathbf{1})$$

Note that the application of hook yields two new methods activated by a trivial input of type $\mathbf{1}$ (which models, say, the pushing of a button), but retains all the already available methods. Therefore a final step is necessary to conform this object with the intended folder signature. This amounts to getting rid of unused methods, by hiding the original stack attributes and relabeling operations. Formally, we obtain

$$\mathsf{FolderC} \;=\; (\mathsf{L.Stack} \boxast \mathsf{R.Stack})[\mathsf{sel}, \mathsf{ext}] \Lsh_{4,1\text{-}5,2} [\mathsf{relb}, \mathsf{hide}]$$

of type

$$\mathsf{new} : \mathbf{1} + \mathsf{insert} : Elem + \mathsf{remove} : \mathbf{1} + \mathsf{forward} : \mathbf{1} + \mathsf{backward} : \mathbf{1} \quad \longrightarrow \quad \mathbf{1}$$

where the relabelling function $\mathsf{relb} = [\triangle, i_2, i_3, i_6, i_7]$ is a case analysis defined as an *either*, and $\mathsf{hide} = !$ is just the (universal) function which collapses any data into the singleton set $\mathbf{1}$.

We are finally in position to discuss the relationship between FolderO and FolderC, which corresponds to two alternative ways of building a folder out of an class component modelling stacks, emphasizing, respectively, *objectification* or *composition.*

---

[12] In a sense, this is the classical 'recipe' of process algebra [Milner, 1999]: interaction amounts to parallel composition with synchronization on shared points (*i.e.*, *actions* in a typical process calculi, *attribute/method pairs* above).

Lemma 13. *Objects* FolderO *and* FolderC*, as specified above, are bisimilar,* i.e.,

$$\mathsf{FolderO} \;\cong\; \mathsf{FolderC}$$

Proof. First notice that the state spaces of both FolderC and FolderO are of type $Stack \times Stack$, by definition of $\circledast$ in the former case, by construction of sort *Folder*, in the latter. Noting that there are no visible attributes involved and assuming both objects are initialized with a pair of empty stacks, the identity on $Stack \times Stack$ provides a Moore machine morphism which is enough to establish bisimilarity. Note that, on the absence of attributes, there is no need to restrict ourselves to next-bisimilarity. To establish state identity as a morphism between coalgebras underlying FolderC and FolderO amounts to prove the equivalence of each pair of corresponding methods. As such methods are aggregated through an *either*, the proof can be made independently for each method involved. We sketch below the proof of equivalence between method forward in FolderC and the original *OP-forward* in FolderO. Let $u \in Stack \times Stack$ represent the current state value.

$$
\begin{array}{cl}
 & a_{\mathsf{FolderC}}\,(u, \mathsf{forward}) \\
= & \quad\{ \text{ wrapping, } \mathsf{relb} \text{ embedding } \} \\
 & a_{(\mathsf{L.Stack} \circledast \mathsf{R.Stack})[\mathsf{sel},\mathsf{ext}]\urcorner_{4,1\text{-}5,2}}\,(u, i_6 *) \\
= & \quad\{ \text{ hook definition } \} \\
 & \Phi_{4,1}\; u \\
= & \quad\{ \text{ unfolding definition of } \Phi_{4,1} \} \\
 & a_{(\mathsf{L.Stack} \circledast \mathsf{R.Stack})[\mathsf{sel},\mathsf{ext}]} \cdot (\mathsf{id} \times i_4) \cdot (\mathsf{id} \times \pi_1) \cdot \langle \mathsf{id}, o_{(\mathsf{L.Stack} \circledast \mathsf{R.Stack})[\mathsf{sel},\mathsf{ext}]} \rangle\; u \\
= & \quad\{ \text{ wrapping definition } \} \\
 & a_{(\mathsf{L.Stack} \circledast \mathsf{R.Stack})[\mathsf{sel},\mathsf{ext}]} \cdot (\mathsf{id} \times i_4) \cdot (\mathsf{id} \times \pi_1) \cdot \langle \mathsf{id}, \mathsf{ext} \cdot o_{(\mathsf{L.Stack} \circledast \mathsf{R.Stack})} \rangle\; u \\
= & \quad\{ \circledast \text{ definition } \} \\
 & a_{(\mathsf{L.Stack} \circledast \mathsf{R.Stack})[\mathsf{sel},\mathsf{ext}]} \cdot (\mathsf{id} \times i_4) \cdot (\mathsf{id} \times \pi_1) \cdot \langle \mathsf{id}, \mathsf{ext} \cdot o_{\mathsf{L.Stack}} \times o_{\mathsf{R.Stack}} \rangle\; u \\
= & \quad\{ \times \text{ absorption } \} \\
 & a_{(\mathsf{L.Stack} \circledast \mathsf{R.Stack})[\mathsf{sel},\mathsf{ext}]} \langle \mathsf{id}, i_4 \cdot \pi_1 \cdot \mathsf{ext} \cdot o_{\mathsf{L.Stack}} \times o_{\mathsf{R.Stack}} \rangle\; u \\
= & \quad\{ \text{ definition of } \mathsf{L.Stack} \text{ and } \mathsf{R.Stack} \text{ attributes, isomorphism } \mathsf{ext} \} \\
 & a_{(\mathsf{L.Stack} \circledast \mathsf{R.Stack})[\mathsf{sel},\mathsf{ext}]} \langle u, (i_4 \cdot \pi_1)\,((*, \mathsf{L.top}(\pi_1\, u))(\mathsf{R.top}(\pi_2\, u), *)) \rangle \\
= & \quad\{ \times \text{ cancellation } \} \\
 & a_{(\mathsf{L.Stack} \circledast \mathsf{R.Stack})[\mathsf{sel},\mathsf{ext}]} \langle u, (i_4\,(*, \mathsf{L.top}(\pi_1\, u)) \rangle \\
= & \quad\{ \mathsf{sel} \text{ embedding, } \circledast \text{ definition } \} \\
 & (\mathsf{L.pop}(\pi_1\, u)), \mathsf{R.push}(\pi_2\, u, \mathsf{L.top}(\pi_1\, u))) \\
= & \quad\{ \textit{Class-Folder-Obj} \text{ definition } \} \\
 & \textit{forward}\; u \\
= & \quad\{ \textit{Class-Folder-Obj} \text{ definition } \} \\
 & \textit{OP-forward}\, *
\end{array}
$$

□

7 Conclusions and future work

This paper addresses the problem of devising suitable class hierarchies within the context of object-oriented system development. The paper consists of two parts: a methodology to lift an algebraic specification of a system into a class hierarchy is presented first, followed by a calculus for modeling and reasoning about a class hierarchy and its dynamical behaviour.

The approach includes *objectification*, a formal specification design technique which inspects the object-oriented potential of a declarative model and brings the *implicit objects* explicit. Criteria which detect such object-orientedness and decide about the structure and contents of the target class hierarchy are formalized and implemented in a tool prototype which embeds VDM-SL into VDM++. Once such *implicit objects* have been generated and framed as generalised Moore machines, reasoning about their composition can be conducted *coinductively* in a modular way.

As a formal modelling technique, objectification brings with it a *declarative-first* approach to formal specification and a separation of concerns: one should model static semantics first, let this model evolve into a minimal OO model and — finally — specify on top of this all complex behavioural aspects of the original problem. In this way one is safe from too convoluted start-up object-models which may compromise design simplicity and elegance — two important aspects of formal modelling — in the long run.

We don't claim completeness and correctness of the proposed objectification criteria. Surely, there is much future work in this respect. For instance, the inspection of function bodies and type definitions will bring about other program analysis techniques, namely *slicing* [Weiser, 1981]. Moreover, some polynomial type patterns can be detected which improve the target class hierarchy in a way which maximizes *record subtyping* [Oliveira, 1997].

Moving on to more immediate improvements, the fact that the current version of the prototype tool only inspects functional (deterministic) fold-algebras is not a shortcoming of the approach: it can be easily extended to relational fold-algebras, which are specified in VDM-SL via *pre/post*-conditions. Moreover, the identification of promotable types should be extended from sort names (in S) to type expressions (in $\mathsf{T}S$) that are not associated to a sort name. Another foreseeable improvement has to do with parameter "flipping" and currying in the generation of new methods such as *TopPop*.

On the other hand, the shift of emphasis from inheritance to object composition paves the way to *component-based* design in a perspective similar to that of [Nierstrasz and Dami, 1995, Jifeng et al., 2003] or [Arbab, 2003]. Going further in this direction would mean, first of all, to split methods' invocation into an input and an output interaction, formalized as specific named *ports* (see [Barbosa and Barbosa, 2004] for preliminary work on that direction). This will enable object (or component) *deployment* as the specification of activation patterns for such ports — for example an object of class *Class-Stack-Obj* could be deployed in a context requiring the activation of a fixed number of *pushes* before a *pop* is allowed to occur. Yet another step will introduce such port names in the coalgebra state data, such that the object interaction abilities will change dynamically, open the way to model some form of *mobility*.

8 Related work

The proposed *objectification* technique is related to similar wwork developed in the area of reverse engineering and software restructuring aiming at re-implementation towards object-oriented code. References, like *e.g.* [Gall and Klösch, 1996], [Gall et al., 1995], [Bellay and Gall, 1998] and [Pinzger et al., 2004] present strategies for converting conventional (non-object-oriented) code into object oriented one, provided that the target object oriented language is an extension of the original procedural language (e.g. C to C++, PASCAL to Object Pascal). The transformation process is developed at a level that is very close to the source code, and aims at modifying it as little as possible. This (manual) transformation is based on the "knowledge" of software engineers about class identification. A methodology for reverse engineering of procedural code, based on a diagrammatic abstraction of the original system is presented in [Cheng and Auernheimer, 1993]. The diagrams thus obtained are a basis for the subsequent formal specification of the system.

References [Gannod and Cheng, 1996] and [Gannod and Cheng, 2001] present a method for C-code reverse engineering. This approach is based on *weakest/strongest preconditions* and is adaptable to any procedural language. In particular it is shown how, at a later stage, classes of objects can be identified using the logic abstraction created earlier, aiming at restructuring the software towards the object oriented paradigm. The methodology considers an object as an abstract data type, objects being identified by a number of guidelines. The objectification techniques put forward in the current paper take decisions similar to the ones in [Cheng and Gannod, 1993].

For maintaining bad documented and unspecified *legacy code* some techniques have been developed for code decompilation and reversal to Z^{++} notation [Bowen et al., 1993]. These techniques have been studied for COBOL and FORTRAN code reversal, and are divided in three stages [Bowen et al., 1993]: *clean*, *specify* and *simplify*. Prior to the last step, through a method of functional abstraction, the code itself is reverse engineered to a first order functional language, in which there are only two constructors: functional composition $f\ a$ and conditional expressions *if* e *then* a *else* b. With this two constructs its possible to represent the program's functionality through a set of equations. In the last step, one *tries* to form hierarchies between classes of objects. Functional equations, obtained in the previous phase, are relaxed to relational equations. The relational equations are than applied as a set of transformations and simplifications.

The calculus of Moore machines introduced in section 6 is in debt to previous research of the second and third authors on coalgebraic calculi for software component's models, as documented in references [Barbosa and Oliveira, 2003, Barbosa, 2003, Barbosa et al., 2005]. Such work, however, does not cover the object's model and calculus proposed here.

Acknowledgments

The work reported in this paper has been carried out in the context of the PURe Project *(Program Understanding and Re-engineering: Calculi and Applications)* funded by Fct (the Portuguese Science and Technology Foundation) under contract `POSI/ICHS/ 44304/2002`.

The contributions by Joost Visser to an earlier version of this paper are gratefully acknowledged. The Vdm++ and Vdm-sl AST modules imported by the current version of the prototype tool have been borrowed from Ifad Ltd (Denmark) under a non-disclosure agreement.

References

[Arbab, 2003] Arbab, F. (2003). Abstract behaviour types: a foundation model for components and their composition. In de Boer, F. S., Bonsangue, M., Graf, S., and de Roever, W.-P., editors, *Proc. First International Symposium on Formal Methods for Components and Objects (FMCO'02)*, pages 33–70. Springer Lect. Notes Comp. Sci. (2852).

[Barbosa, 2001] Barbosa, L. (2001). *Components as Coalgebras*. PhD thesis, Universidade do Minho.

[Barbosa, 2003] Barbosa, L. S. (2003). Towards a Calculus of State-based Software Components. *Journal of Universal Computer Science*, 9(8):891–909.

[Barbosa and Oliveira, 2003] Barbosa, L. S. and Oliveira, J. N. (2003). State-based components made generic. In Gumm, H. P., editor, *CMCS'03, Elect. Notes in Theor. Comp. Sci.*, volume 82.1. Elsevier.

[Barbosa et al., 2005] Barbosa, L. S., Sun, M., Aichernig, B. K., and Rodrigues, N. (2005). On the semantics of componentware: a coalgebraic perspective. In He, J. and Liu, Z., editors, *Mathematical Frameworks for Component Software: Models for Analysis and Synthesis*, Series on Component-Based Development. World Scientific.

[Barbosa and Barbosa, 2004] Barbosa, M. and Barbosa, L. (2004). Specifying Software Connectors. In Araki, K. and Liu, Z., editors, *Proc. First International Colloquim on Theoretical Aspects of Computing (ICTAC'04), Guiyang, China*, pages 53–68. Springer Lect. Notes Comp. Sci. (3407).

[Bellay and Gall, 1998] Bellay, B. and Gall, H. (1998). Reverse engineering to recover and describe a system's architecture. In *ESPRIT ARES Workshop*, pages 115–122.

[Bird and Moor, 1997] Bird, R. and Moor, O. (1997). *The Algebra of Programming*. Series in Computer Science. Prentice-Hall International.

[Bowen et al., 1993] Bowen, J., Breuer, P., and Lano, K. (1993). Formal specifications in software maintenance: From code to Z^{++} and back again. *Information and Software Technology*, 35(11/12):679–690.

[Cheng and Auernheimer, 1993] Cheng, B. and Auernheimer, B. (1993). Applying formal methods and object-oriented analysis to existing flight software. In *Proc. of 18th Annual Software Engineering Workshop, Greenbelt, Maryland*, pages 274–282.

[Cheng and Gannod, 1993] Cheng, B. and Gannod, G. (1993). A two-phase approach to reverse engineering using formal methods. In *Formal Methods in Programming and Their Applications*, pages 335–348.

[Coad and Yourdon, 1991] Coad, P. and Yourdon, E. (1991). *Object-Oriented Analysis*. Prentice Hall, 2nd edition.

[Cruz, 2004] Cruz, A. (2004). Objectification of formal specifications. Master's thesis, University of Minho. (in Portuguese).

[Fitzgerald and Larsen, 1998] Fitzgerald, J. and Larsen, P. (1998). *Modelling Systems: Practical Tools and Techniques for Software Development* . Cambridge University Press, 1st edition.

[Fitzgerald et al., 2005] Fitzgerald, J., Larsen, P., Mukherjee, P., Plat, N., and Verhoef, M. (2005). *Validated Designs for Object–oriented Systems*. Springer, New York.

[Gall and Klösch, 1996] Gall, H. and Klösch, R. (1996). Improving reusability of legacy applications through object-oriented re-architecturing. Technical report, DSD, Technical University of Vienna.

[Gall et al., 1995] Gall, H., Klösch, R., and Mittermeir, R. (1995). Architectural transformation of legacy systems. In *Proceedings of the ICSE-17 Workshop on Program Transformation for Software Evolution, Seattle, Washington*.

[Gannod and Cheng, 1996] Gannod, G. and Cheng, B. (1996). Strongest postcondition semantics as the formal basis for reverse engineering. *Journal of automated software engineering*, 3(1/2).

[Gannod and Cheng, 2001] Gannod, G. and Cheng, B. (2001). A suite of tools for facilitating reverse engineering using formal methods. In *Proc. of 9th International Workshop on Program Comprehension (IWPC 2001), Toronto, Canada*, pages 221–232.

[Hoogendijk, 1997] Hoogendijk, P. (1997). *A Generic Theory of Data Types*. PhD thesis, University of Eindhoven, The Netherlands.

[Jacobs and Rutten, 1997] Jacobs, B. and Rutten, J. (1997). A tutorial on (co)algebras and (co)induction. *EATCS Bulletin*, 62:222–159.

[Jifeng et al., 2003] Jifeng, H., Zhiming, L., and Xiaoshan, L. (2003). A contract-oriented approach to component-based programming. In Liu, Z., editor, *Proc. of FACS'03, (*Formal Approaches to Component Software*)*, Pisa.

[Kock, 1972] Kock, A. (1972). Strong functors and monoidal monads. *Archiv für Mathematik*, 23:113–120.

[Lano, 1991] Lano, K. (1991). Z++, an object-oriented extension to Z. In Nicholls, J. E., editor, *Proceedings of the Fifth Annual Z User Meeting on Z User Workshop*, Workshops in Computing Series, pages 151–172. Springer-Verlag Workshops in Computing Series.

[Meng and Barbosa, 2004] Meng, S. and Barbosa, L. (2004). On refinement of generic state-based software components. In Rattray, C., Maharaj, S., and Shankland, C., editors, *10th Int. Conf. Algebraic Methods and Software Technology (AMAST)*, pages 506–520, Stirling. Springer Lect. Notes Comp. Sci. (3116).

[Milner, 1999] Milner, R. (1999). *Communicating and Mobile Processes: the π-Calculus*. Cambridge University Press.

[Moore, 1966] Moore, E. F. (1966). Gedanken experiments on sequential machines. In *Automata Studies*, pages 129–153. Princeton University Press.

[Nierstrasz and Dami, 1995] Nierstrasz, O. and Dami, L. (1995). Component-oriented software technology. In Nierstrasz, O. and Tsichritzis, D., editors, *Object-Oriented Software Composition*, pages 3–28. Prentice-Hall.

[Oliveira, 1997] Oliveira, J. (1997). A calculational approach to reverse specification. Seminar presented at UNU-IIST, Macau, May 13th, 1997, 22 pages.

[Oliveira and Rodrigues, 2004] Oliveira, J. and Rodrigues, C. (2004). Transposing relations: from *Maybe* functions to hash tables. In *Seventh International Conference on Mathematics of Program Construction, 12-14 July, 2004, Stirling, Scotland, UK*, pages 334–356. Springer Lect. Notes Comp. Sci. (3125).

[Pinzger et al., 2004] Pinzger, M., Fischer, M., Jazayeri, M., and Gall, H. (2004). Abstracting module views from source code. In *Proceedings of the International Conference on Software Maintenance*, pages 533–533, Chicago, USA. IEEE Computer Society Press.

[Rutten, 2000] Rutten, J. (2000). Universal coalgebra: A theory of systems. *Theor. Comp. Sci.*, 249(1):3–80.

[Smith, 2000] Smith, G. (2000). *The Object-Z Specification Language*. Advances in Formal Methods. Kluwer Academic Publishers. 160 pages.

[Spivey, 1989] Spivey, J. (1989). *The Z Notation — A Reference Manual*. Series in Computer Science. Prentice-Hall International. C.A. R. Hoare.

[Turi and Rutten, 1998] Turi, D. and Rutten, J. (1998). On the foundations of final coalgebra semantics: non-well-founded sets, partial orders, metric spaces. *Math. Struct. in Comp. Sci.*, 8(5):481–540.

[Vene, 2000] Vene, V. (2000). *Categorical Programming with Inductive and Coinductive Types*. PhD thesis, Faculty of Mathematics, University of Tartu (*Dissertationsa Mathematicae* 23).

[Visser, 2003] Visser, J. (2003). *Generic Traversal over Typed Source Code Representations*. Ph. D. dissertation, University of Amsterdam, Amsterdam, The Netherlands.

[Wadler, 1990] Wadler, P. (1990). Comprehending monads. In *Proceedings of the 1990 ACM Conference on Lisp and Functional Programming, Nice, France*.

[Weiser, 1981] Weiser, M. (1981). Program slicing. In *Fifth International Conference on Software Engineering, San Diego, California*.