OOZE: An Object Oriented Z Environment

Antonio J. Alencar

Programming Research Group
University of Oxford
11 Keble Road
Oxford OX1 3QD, UK
E-mail: alencar@prg.ox.ac.uk
Tel: +44 (865) 273 869
FAX: +44 (865) 273 839

Joseph A. Goguen

Programming Research Group
University of Oxford
11 Keble Road
Oxford OX1 3QD, UK
E-mail: goguen@prg.ox.ac.uk
Tel: +44 (865) 272 567
FAX: +44 (865) 273 839

OOZE, which stands for "Object Oriented Z Environment," is a generalized wide spectrum object oriented language that builds on the notation and style of Z. OOZE supports requirements, specifications, interpretable programs, and compilable programs. The OOZE system is based on OBJ3, and provides rapid prototyping and theorem proving facilities over a module database. OOZE modules can be generic, can be organized hierarchically, and can be used for structuring and reusing requirements, specifications, or code. Modules can be linked by views, which assert relationships of refinement. Module interfaces can be precisely specified using theories. Abstract data types, multiple inheritance, complex objects, overloading and dynamic binding are supported. Data types, objects, classes and modules are clearly distinguished from one another, and the entire language has a precise and relatively simple semantics based on order sorted, hidden sorted algebra.

Key Words: Object Orientated, Specification, Requirement, Rapid Prototyping, Algebraic Semantics, Z.

1 Introduction

OOZE, which stands for "Object Oriented Z Environment," is primarily intended for the requirement and specification phases of the system life cycle. It uses the graphical notation and comment convention of Z, formalizes its style, and adapts it to fit the object oriented paradigm, allowing declarations for *classes*, *attributes* and *methods* within modules. Attributes can be class-valued, i.e., *complex objects* are supported. Also, unlike other object oriented adaptations of Z, *objects* (which are instances) are carefully distinguished from *class declarations* (which serve as templates for objects). Objects are also organized into *meta-classes*, for ease of identification and iteration. Multiple inheritance is supported for both classes and modules. Abstract data types, overloading, and exception handling are also supported. OOZE has an interpretable sublanguage that can be used for rapid prototyping, and a compilable sublanguage that can be used for implementation. All of this has a precise semantics that is based upon order sorted algebra.

Formal methods emerged in the mid-seventies as an attempt to add mathematical rigour to the development of computer systems. It is claimed that their use can significantly increase quality by permitting accurate design at a high level of abstraction. Consequently, design errors can be reduced

and confidence in the system behaviour increased. Formal specifications can also provide a reliable basis for documentation, implementation and maintenance.

Despite these attractive properties, formal methods have the reputation of being hard to use, because a reasonable understanding of computer science and discrete mathematics is required, and unfortunately, the average computer professional seems not to meet this requirement. Also, communication with clients can be difficult, because they may not easily understand the syntax and semantics of formal specification languages. Moreover, the time spent on design may increase, and it is not obvious that the extra cost is returned in all cases[1].

The present work describes a programming environment that can take advantage of the attractive properties of formal methods while reducing the burden associated with their use. This environment includes not just a syntax and type checker, but also an interpreter for an executable sublanguage, a theorem prover, and a module database; all of this is based on facilities provided by the OBJ3 [14] system. OOZE can be considered a syntactic variant of FOOPS [11], and indeed, it has the same semantics as FOOPS. However, OOZE is intended to be used in a different way by a different constituency, and it has a very different appearance. Thus, although OOZE looks and acts like a model-based language, it actually has a relatively simple equational semantics. However, unlike most other equational languages, OOZE supports the encapsulation of states, and more generally, is truly object oriented.

By providing animation facilities for rapid prototyping, OOZE helps to improve communication with the client, and makes it easier to master its mathematical basis. As a result, OOZE should reduce the time and cost of formal methods, and also increase confidence in correctness. OOZE can also help with subsequent phases of system development, including design, coding and maintenance.

2 Z

Z is a model-based specification language that has been primarily developed in the Programming Research Group at the Oxford University Computing Laboratory [18, 25], and has been widely used in industry. Z is based on set theory and the first-order predicate calculus, and has been successfully used for applications to distributed computing, transaction processing, operating systems, large information systems, etc. The heart of Z is its use of schemas to describe state spaces and state-changing operations. Schemas are the basis for the incremental presentation of Z specifications. They encourage structuring, and are syntactically integrated with informal prose, to help the reader understand what has been specified. Schemas also provide a graphically elegant way of delimiting the parts of a specification.

Although modularity is widely recognized as helpful in all phases of software development, Z provides only rather weak modularity. Z specifications do not have clearly delimited beginnings or ends, cannot be made generic or instantiated, and lack the ability to import and export. Schemas are not modules, because they do not hide information, and they have no natural meaning that is independent of context. Also, schema scope conventions are subtle, and can cause complex specifications to be misinterpreted. Therefore Z specifications can be hard to read and maintain, and their parts can be hard to reuse in new developments; see [22, 6] for related discussion.

A Z specification of a large and complex system may contain many state and operation schemas, which can be dispersed arbitrarily through the specification, because schemas do not enforce any association of operations with states. All state space schemas and operation schemas are global within the specification, i.e., they can be used in any other state space or operation schema. Therefore the

[1] See [4] for a discussion of the pros and cons of using formal methods.

dependency relations that exist between schemas can only be determined by examining all schemas in a specification. In addition, the schema calculus [25] allows schemas to be combined in a large variety of ways, some of which seem quite unnatural, especially from a logical point of view. Thus, understanding and updating specifications can be very hard in some cases.

The best ideas of Z are in its style. These ideas include its comment conventions, its graphical layout, its use of ordinary set theoretic and logical notation, its conventions for decorating input and output variables, and some uses of schemas.

3 OOZE

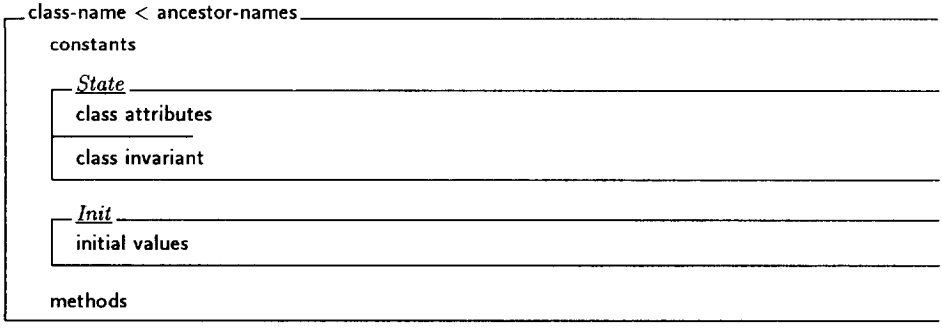
OOZE formalizes and enforces these attractive ideas, and combines them with other good ideas in software engineering, such as object orientation, modularization and algebraic semantics, yielding an environment that strongly supports reusability and the subordination of complexity. A summary of differences from Z appears in Section 7.

While numbers and other data elements do not change, objects have attributes that may change. For example, the natural number 7 is fixed and eternal; it does not change. But the amount of money in one's bank account may vary with time. Failure to make this distinction can cause confusions with undesirable practical consequences. For example, in Smalltalk72 [16], addition is not commutative; e.g., $2 + 2.0$ is 4 whereas $2.0 + 2$ is 4.0, because in each case the second number is sent as a message to the first, and so a different addition method is used.

3.1 Objects

By convention, a Z schema with non-decorated variables represents the state space of some system component, and is followed by another schema defining the initial values for those variables, and by other schemas that define operations on these variables. But these conventions are not enforced, and it is very easy to write specifications that violate them.

OOZE groups the state space schema, the initial state schema and operation schemas into a single unit[2]. Only the operations defined in the unit are allowed to act upon the objects of that class. Syntactically, such a unit is an open-sided named box in which the features of the class are defined, with the following general form,



where **class-name** is the name of the class. The symbol < indicates that the class being defined is a subclass of one or more previously defined classes, named in **ancestor-names**. The **constants** are fixed values which cannot be changed by any method, and are the same for all instances of the class. The

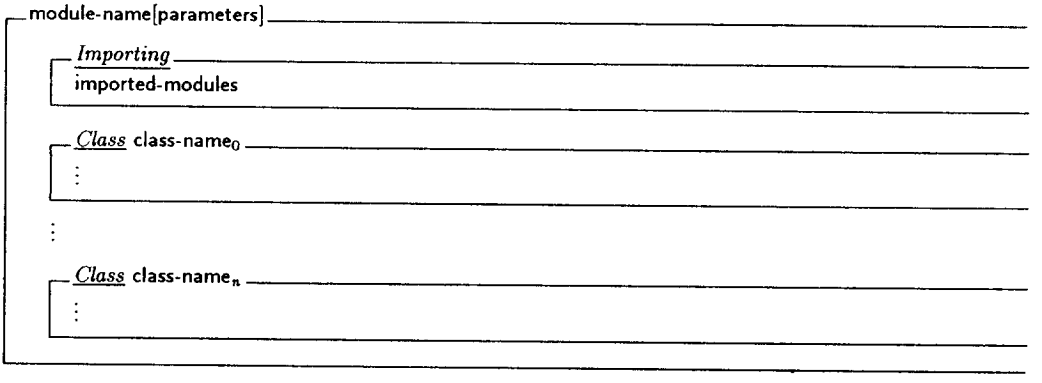
[2]This structure is similar to that used in Object-Z [6], but its semantics is quite different.

class attributes are variables that can take values either in another class or in a data type. The class invariant is a predicate that constrains the values that the attributes can take; it must hold for all objects of the class, before and after the execution of methods and in the initial state. *Init* gives the initial values that attributes take. methods are given in schemas that define operations involving one or more attributes of the same class, and possibly input or output variables; these define the relationship between the state of an object before and after the execution of a method. (Differences between the syntax and semantics of schemas in OOZE and Z are discussed in Section 3.8).

3.2 Encapsulation of Classes in Modules

To build a large system, we must overcome the difficulties of manipulating huge amounts of information. Explicit mechanisms for modularization are needed to support this in a natural way, allowing the system to be defined in small separate pieces that are easy to read and manage, and that can be easily combined [8]. Indeed, such mechanisms may significantly reduce the cost associated with system development, and are becoming common not only in languages for programming, but also in languages for prototyping and specification. There is no widely accepted agreement on an ideal mechanism, and several interesting alternatives have been proposed, including Clear [3], OBJ [14], Ada [1], SML [17], and the extension of VDM proposed by Bear [2].

In OOZE, classes of objects are encapsulated in modules, which can be generic and can contain any number of classes:



Here module-name names the module; parameters is a list of formal names with their corresponding requirements on the actual parameters that instantiate the module; imported-modules lists the imported modules, and class-name$_0$, ..., class-name$_n$ are the classes defined in the module. Note the clear distinction between module importation and class inheritance. The former has to do with the scope of declarations; for example, a class cannot be used unless the module that declares it is imported. Note that module importation is *transitive*, so that if A imports B and B imports C, then everything in C is also available in A. See [8, 9] for more detailed discussions of the module concepts used in OOZE, including importation and genericity; they are evolved from those introduced in Clear [3] and implemented in OBJ [14], and are given a precise semantics using the theory of institutions.

In many object oriented languages, including Eiffel [20], Smalltalk [21] and Object-Z [6], modules and classes are identified, so that only one class can be encapsulated. Because of this, cases where several classes have interdependent representations are not easily captured. For example, consider a class `Private-Teachers` and a class `Independent-Students`, where each class has only one attribute, with a value involving the other class: teachers keep a list of their students, and students keep a list of their teachers. Because these two classes are interdependent, it is impossible to determine which

should be defined first. If no order is established, then the object hierarchy is not properly enforced. A straightforward solution is to introduce both classes in one module. In OOZE, data, objects, classes and modules are distinct entities, carefully distinguished both syntactically and semantically. (This discussion follows [15]).

When encapsulating classes in modules, it is not rare to end up with modules with just one class. Moreover, the module and the class usually have the same name. As a result, for the sake of simplicity and conciseness, the class name and enclosing box for its space state and methods can be omitted in OOZE, as in Section 3.1.

3.2.1 Arrays

Consider the following code for arrays of real numbers[3]:

Array

State

$array : \mathbf{Z} \twoheadrightarrow \mathbf{R}$ [hidden]
$lower\_bound : \mathbf{Z}$
$upper\_bound : \mathbf{Z}$

Init

$b_1?, b_2? : \mathbf{Z}$

$\forall j : \min\{b_1?, b_2?\} \mathinner{..} \max\{b_1?, b_2?\} \bullet array'\ j = 0$
$lower\_bound' = \min\{b_1?, b_2?\}$
$upper\_bound' = \max\{b_1?, b_2?\}$

Store

$\Delta array$
$j? : lower\_bound \mathinner{..} upper\_bound$
$x? : \mathbf{R}$

$array' = array \oplus \{j? \mapsto x?\}$

Get

$j? : lower\_bound \mathinner{..} upper\_bound$
$x! : \mathbf{R}$

$x! = array\ j?$

Max_of

$x! : \mathbf{R}$

$x! = \max(\mathrm{ran}\ array)$

Following the notation of Z, we let $\mathbf{Z}$, $\mathbf{R}$ and $\mathbf{N}$ denote the integers, reals and naturals respectively. Also, $f : A \twoheadrightarrow B$ indicates that f is a finite partial function from A to B, while "ran f" denotes the range of f, and $x..y$ denotes the interval $\{n \mid x \leq n \leq y\}$. Finally, $f \oplus \{a \mapsto b\}$ denotes the partial function equal to f except that it takes value b on argument a. The notation "$\forall x : X \bullet \ldots$" for quantification over a variable x of sort X also comes from Z.

[3]The functions max and min are built in.

Arrays have attributes *array*, *lower_bound* and *upper_bound*. The *array* attribute is *hidden*, i.e., it is an internal state that is not visible outside the current module. This class does not inherit from any others, and it has no invariant. Only the methods *Init*, *Store*, *Get* and *Max_of* can be used on Arrays; these respectively create an array, store a value in an array, retrieve a stored value, and return the maximum value in an array.

Values of attributes before method application are indicated by undashed variables, and by dashed (') variables afterwards. Method inputs are indicated by variables with an interrogation mark (?), and outputs by variables with an exclamation mark (!). Δ heads a list of attributes whose values may be changed by the method; attributes absent from the list are unchanged, i.e., the dashed attribute value equals the undashed one. The absence of a Δ list means that no attribute value can be changed by the method[4]. Unlike Z, these are not mere conventions; they are part of the definition of OOZE, and are enforced by the implementation.

3.3 Parameters and Theories

It can be very useful to define precisely the properties that the actual parameters to a parameterized module must satisfy in order for it to work correctly; in OOZE these properties are given in a *theory*; theories are a second kind of module in OOZE, with the same syntactic form. In particular, theories can be parameterized and can use and inherit other modules. A theory is introduced in an open-sided box, and its name is preceded by the key word *Theory*.

Theories declare properties and provide a convenient way to document module interfaces. Understandability and correctness for reusability are improved by this feature. For example, the following theory requires that an actual parameter provide a totally ordered set with a given element:

Theory *TotalOrder*

$[X]$

$v : X$

$\_ \sqsubset \_ : X \leftrightarrow X$

$\forall x, y, z : X \bullet$

$\neg (x \sqsubset x)$

$(x \sqsubset y) \wedge (y \sqsubset z) \Rightarrow (x \sqsubset z)$

$(x \sqsubset y) \vee (x = y) \vee (y \sqsubset x)$

Here the notation $[X]$ indicates that X is a set newly introduced for this specification.

The formal parameters of an OOZE module are given after its name in a list, along with the requirements that they must satisfy. The actual parameters of generic modules are not sets, constants and functions, but rather modules. The motivation for this is that the items that naturally occur in modules are usually closely related, so that it is natural to consider them together rather than separately. Moreover, by allowing parameters to be modules, OOZE incorporates the powerful mechanisms of parameterized programming [8]. In the syntax below, $\mathsf{P}_0, \mathsf{P}_1, \cdots, \mathsf{P}_n$ are the formal module names, while $\mathsf{T}_0, \mathsf{T}_1, \cdots, \mathsf{T}_n$ are theory names:

module-name$[\mathsf{P}_0 :: \mathsf{T}_0, \mathsf{P}_1 :: \mathsf{T}_1, \cdots, \mathsf{P}_n :: \mathsf{T}_n]$

$\vdots$

$\vdots$

[4] *Init* is an exception to this rule; its signature has no Δ list, and only dashed variables are available.

3.3.1 A Parameterized Array Module

The following parameterized version *NewArray* of *Array* is much more flexible, in that it can be instantiated to define arrays with different types and ranges:

NewArray[P :: *TotalOrder*]

State

$array : \mathbb{Z} \nrightarrow X$ [hidden]
$lower\_bound : \mathbb{Z}$
$upper\_bound : \mathbb{Z}$

$Max : \mathsf{F}_1 X \to X$

$\forall S : \mathsf{F}_1 X;\ x, y : S \bullet$
$\quad Max(S) = x \Leftrightarrow y \sqsubset x \vee y = x$

Init

$b_1?, b_2? : \mathbb{Z}$

$\forall j : \min\{b_1?, b_2?\} \,..\, \max\{b_1?, b_2?\} \bullet array'\ j = v$
$lower\_bound = \min\{b_1?, b_2?\}$
$upper\_bound = \max\{b_1?, b_2?\}$

Store

$\Delta array$
$j? : lower\_bound \,..\, upper\_bound$
$x? : X$

$array' = array \oplus \{j? \mapsto x?\}$

Get

$j? : lower\_bound \,..\, upper\_bound$
$x! : X$

$x! = array\ j?$

Max_of

$x! : X$

$x! = Max(\mathrm{ran}\ array)$

Here $\mathsf{F}_1 X$ denotes the set of all non-empty finite subsets of X, and $Max(\{a_o, a_1, \cdots, a_n\})$ denotes the maximum element of a non-empty totally ordered finite set.

3.4 Views for Parameterized Modules

A view can be used to say how a given module satisfies a given theory. A view is a mapping from the features (sets, methods, functions, constants, etc.) of the source module to the features of the target module, preserving subtype relations and the rank of methods and functions. Views are needed because an actual parameter may satisfy a given theory in more than one way. For example, the set of natural numbers is totally ordered with the relation $<$ or with $>$; these correspond to two distinct views. Views that are used for instantiating parameterized modules have the following general form,

$\mathsf{M}\{\mathsf{t}_0 \mapsto \mathsf{m}_0, \mathsf{t}_1 \mapsto \mathsf{m}_1, \cdots, \mathsf{t}_n \mapsto \mathsf{m}_n\}$,

in which M is the name of the parameterized module, while $t_0, t_1, \cdots, t_n$ are names of features in the source theory, and $m_0, m_1, \cdots, m_n$ are names of features in M.

Data types in OOZE are defined in modules similar to those used for classes, but they have initial order sorted algebra semantics[5]. For example, the module *Nat* gives the natural numbers, with the carrier N. Using the *NewArray* class and a view, we can get a class of arrays of natural numbers, with an operation that returns the maximum element, as follows:

NewArray [*Nat* $\{X \mapsto \mathsf{N}, v \mapsto 0, \sqsubset \mapsto <\}$]

Because we already know that *TotalOrder* is the source theory for this view, its name is not needed. It is common to use an "obvious" view, and then it can be distracting to have to present it in detail. Some conventions for simplifying views ease this problem. For example, any pair of the form $S \mapsto S$ can be omitted[6].

3.5 Applying Methods

A basic principle of object oriented programming is that only the methods defined with a class may directly act upon its objects. Because attributes, method inputs and method outputs can be object valued, methods defined in other classes may be needed to define methods that manipulate the attributes of such complex objects. In OOZE, the following syntax indicates that a certain method acts on a certain object,

object•method($p_0, p_1, \cdots, p_n$)

where object is an object name, method is the name of a method on the class to which object belongs, and $p_0, p_1, \cdots, p_n$ are parameters whose types must agree with those of the corresponding formal parameters. Actual parameters are associated with formal parameters according to the order in which the latter are declared. For example, see the methods *Store* and *Get* in the *Matrix* class in Section 3.7 below. The method *Init* is an exception, because it acts on no objects, and its syntax is simplified to the form *Init*($p_0, p_1, \cdots, p_n$).

To actually create the object A of class *NewArray* from its "template" in the *NewArray* module, it is necessary to first instantiate the module with an actual value for P, say the natural numbers with 0 and $<$, yielding a *NewArrayNat* class, and then to apply the built in method *Init* with values for the parameters *lower_bound* and *upper_bound*,

A•*Init*(1, 10).

OOZE provides a selection function for each visible attribute of a class, indicated by a dot before the attribute name. So if A is the above object of the *NewArrayNat* class, then A•*upper_bound* yields the value 10 for the *upper_bound* of that specific array.

3.6 Overloading

OOZE has a strong but flexible type system. Strong typing is not only useful to catch meaningless expressions, but it also favours the separation of logically and intuitively distinct concepts (such as matrices and arrays) and enhances readability and reusability by documenting such distinctions. Moreover, strong typing supports *overloading*, i.e., attaching more than one meaning to a name. In particular, overloading allows simpler code, because the context can determine which possibility is intended.

[5]The semantics of data types is discussed in Section 6. Many data types including naturals, integers, reals, sequences, and tuples are built in, and can be used anywhere.

[6]Views in OOZE are derived from the OBJ3 [14] implementation of the ideas introduced in Clear [3]; see [8] for further discussion.

Overloaded attributes and methods of a class can be distinguished by the type required in a given context. For example, overloading resolution applies to the methods *Store*, *Get*, *Init* and *Max_of* in the Matrix class below.

3.7 Object-Valued Attributes

We illustrate complex objects using a class *Matrix* with an attribute that takes values in the *NewArray* class. Inheritance applies to all kinds of modules in OOZE and the effect is simply that the descendent module inherits all features of its parents. For example the module *Matrix* below inherits the module *NewArray*:

Matrix[P :: *TotalOrder*]

Importing

NewArray[P]

State

$matrix : \mathbb{Z} \nrightarrow NewArray$ [hidden]
$lower\_bound : \mathbb{Z}$
$upper\_bound : \mathbb{Z}$

Init

$mb_1?, mb_2? : \mathbb{Z}$
$ab_1?, ab_2? : \mathbb{Z}$

$\forall j : \min\{mb_1?, mb_2?\} \,..\, \max\{mb_1?, mb_2?\} \bullet matrix'\ j = Init(ab_1?, ab_2?)$
$lower\_bound = \min\{mb_1?, mb_2?\}$
$upper\_bound = \max\{mb_1?, mb_2?\}$

Store

$\Delta matrix$
$i? : lower\_bound..upper\_bound$
$j? : (matrix\ i?) \cdot lower\_bound \,..\, (matrix\ i?) \cdot upper\_bound$
$x? : X$

$matrix' = matrix \oplus \{i? \mapsto (matrix\ i?) \cdot Store(j?, x?)\}$

Get

$i? : lower\_bound \,..\, upper\_bound$
$j? : (matrix\ i?) \cdot lower\_bound \,..\, (matrix\ i?) \cdot upper\_bound$
$x! : X$

$x! = (matrix\ i?) \cdot Get(j?)$

Max_of

$i? : lower\_bound \,..\, upper\_bound$
$x! : X$

$x! = (matrix\ i?) \cdot Max\_of$

3.8 Method Schemas

In both OOZE and Z, schemas are used to define some key aspects of systems, but OOZE schema syntax and semantics have been designed to enhance readability without loss of expressiveness.

In OOZE, the values of variables before and after method application are related by conditional equations. The *if* clause can be considered a pre-condition. The respective equations are required to hold if the condition, expressed by a predicate, is *true*. If the *if* clause is omitted, then the equations must hold in any circumstance. Method schemas have the following general form:

schema_name

declarations

equations
if predicate

equations
if predicate

⋮

In the *RegularAccount* class in Section 3.9.1 below, conditional equations say that after the *Debit* operation, the values of the attributes *bal* and *hist* are changed if the value of *bal* is greater than or equal to $m?$. On the other hand, the *Credit* method changes the value of *bal* and *hist* in any circumstance, because there is no *if* clause.

In OOZE, exceptional and non-exceptional behaviour for methods are defined in distinct schemas having the same name, with the exception schema name preceded by the key word *Error*. Moreover, the input variables appearing in the error schema must be among those appearing in the non-error schema. By this device, users need not understand both situations at once. Therefore code is simplified, readability is enhanced, and complexity is subordinated. The *Debit* method for the *RegularAccount* class illustrates this feature.

If the object that a method is supposed to act upon is omitted, then the method being used and the method being defined are regarded as sharing the same object, as in the *Interest* method in the *SavingsAccount* class below.

3.9 Class Inheritance

OOZE supports multiple inheritance for classes. When one class inherits from another, the attributes defined in the ancestor are added to those of the descendent. If the descendent class declares no attributes, it is assumed that ancestor and descendent have the same attributes. Class invariants defined in ancestors must hold in the descendant, and may also be strengthened. Initial values defined in the descendant take precedence over those of ancestors. All this is illustrated by the *SavingsAccount* $<$ *RegularAccount* declaration below.

3.9.1 Bank Accounts

The *BankAccount* module to follow is parameterized, with parameter requirements defined by the *DateMoneyAndRate* theory below. Note that R_0^+ denotes the positive real numbers union $\{0\}$.

Theory DateMoneyAndRate

$[DATE, MONEY, RATE]$

$MONEY \subset R_0^+$

$RATE \subset R_0^+$

$\forall m : MONEY \bullet \lfloor 100 * m \rfloor = 100 * m$

Every object of the *RegularAccount* class has a balance and a history which records credit and debit transactions; these appear in the attributes *bal* and *hist*. *today* is a built in function that returns the current date. As in Z, $\frown$ denotes sequence concatenation, and $\langle(a, b)\rangle$ is the singleton sequence consisting of the ordered pair (a, b).

BankAccount[P :: *DateMoneyAndRate*]

Class RegularAccount

State

$bal : MONEY$

$hist : seq\ DATE \times MONEY$

Init

$bal' = 0$

$hist' = \langle today, 0\rangle$

Credit

$\Delta\ bal, hist$

$m? : MONEY$

$bal' = bal + m?$

$hist' = hist \frown \langle(today, m?)\rangle$

Debit

$\Delta\ bal, hist$

$m? : MONEY$

$bal' = bal - m?$

$hist' = hist \frown \langle(today, -m?)\rangle$

$\underline{if}\ bal \geq m?$

Error Debit

$m? : MONEY$

$error! : Report$

$error! = Overdrawn$

$\underline{if}\ bal < m?$

Class SavingsAccount < *RegularAccount*

State

$rate : RATE$

Interest

$\Delta\ bal, hist$

$bal' = Credit(rate * bal).bal$

$hist' = Credit(rate * bal).hist$

Note that because there is no *Init* method for *SavingsAccount*, the *Init* method for its superclass *RegularAccount* will be used; however, a value must be given for the attribute *rate* whenever a new

SavingsAccount is created, because there is no default value. The parameterized module *BankAccount* can be instantiated with *DateMoneyAndRate* modules to yield banks operating under various conventions for currency and date. Also, using a notational short cut, the specification of the method *Interest* could have been simplified to $Interest \equiv Credit(bal * rate)$.

3.10 Animation

In order for an OOZE module to be executable, its axioms must have a special form,

m

ΔL
$p_1 : T_1$
$\vdots$
$p_n : T_n$

$a_1' = e_1$
$a_2' = e_2$
$\vdots$
if P

where $a_1, a_2, ...$ are attributes changed by the method m and listed in L, where $p_1, ..., p_n$ are parameters of m with types $T_1, ..., T_n$, where $e_1, e_2, ...$ are expressions in $p_1, ..., p_n$ and the state of the object, including $a_1, a_2, ...$, and where P is a predicate in the same variables. The *if* clause is optional, and there could even be more than one such clause, each giving a set of conditional equations. These equations have a declarative interpretation, and in fact are referentially transparent. They define a method by its effects on attributes. Alternatively, methods can be defined as compositions of other already defined methods. The operations for composing methods include sequential and parallel composition.

Animation is useful for rapid prototyping during the requirement and early specification phases. If the attributes associative or commutative are used for any operation, then the resulting program cannot be compiled, but only interpreted (Section 5 contains an example using the attribute associative). Also, some forms of abstract data type definitions can only be interpreted.

4 Requirements

Usually the process of building a system starts from some very high level requirements. In this initial stage, it is quite common that attributes and methods are not completely determined, and so these initial definitions are satisfied by a large class of models, some of which may not fit the client's expectations; also, animation cannot in general be provided at this early stage. For example, the initial requirements for a bank might simply say that it should be possible to debit and credit accounts, and that these methods should respectively decrease and increase the balance. One can easily imagine models of this theory that are not among those really intended by the bankers, such as a deposit method that always increases the balance by one million currency units.

OOZE uses theories to express requirements, and also for the initial stages of specification. In many typical applications, an OOZE text will evolve until theories are only used to specify properties that parameters should satisfy (however, theories can also be retained to document the earlier stages of the development cycle). At this point, it is possible to use the specification itself as a rapid prototype. Let us consider the following theory of bank accounts[7]:

Theory *LooseAccount*

$[DATE, MONEY]$

$MONEY \subset R_0^+$

State

$bal : MONEY$

$hist : seq\ DATE \times MONEY$

$bal = \sum_{i=1}^{\#hist} second(hist(i))$

Init

$bal' = 0$

$hist' = \langle today, 0 \rangle$

Credit

$\Delta\ bal, hist$

$m? : MONEY$

$bal' \geq bal$

Debit

$\Delta\ bal, hist$

$m? : MONEY$

$bal' \leq bal$

Although the attributes *bal* and *hist* are related by a state invariant, their value after the execution of the methods *Debit* and *Credit* is not completely determined. For example it is possible to credit or debit more money to an account than the argument indicates, and it is also possible to put arbitrary values in the history.

When creating large systems, it is important to relate the different stages of the development process. For example, the *BankAccount* module in Section 3.9.1 satisfies the requirements of the *LooseAccount* theory, and it might represent a more recent stage in the evolution of the same system. In OOZE, this satisfaction relationship is described by a view. Although views were previously used to describe how an actual parameter satisfies a theory, they can also express refinement relationships between any kind of module. However, in this context a more comprehensive notation is needed. The view *Account* that follows asserts how the class *RegularAccount* described in the module *BankAccounts* satisfies the theory *LooseAccount*[8]:

[7]The function *second* extracts the second component of a pair [25].

[8]Note that according with the conventions established in Section 3.2, the theory *LooseAccount* encapsulates the description of a class also named *LooseAccount*.

View Account

LooseAccount $\rightarrow$ *BankAccount*

$[MONEY \mapsto MONEY, DATE \mapsto DATE]$

Class

LooseAccount $\mapsto$ *RegularAccount*

Attributes

$bal \mapsto bal, hist \mapsto hist$

Methods

$Credit \mapsto Credit, Debit \mapsto Debit$

OOZE texts are hierarchically organized into modules, and a complete OOZE system can be encapsulated in one final module. This allows the use of views to express satisfaction (i.e., refinement) relations between different levels of development of the same system. Views between such conglomerates can also be constructed from individual views between component modules. This makes it possible to build views between conglomerates by combining views between their components which are easier to understand and modify. In this way, each phase of the software life cycle can be precisely and quickly documented. Multiple implementations of a single specification or requirement can also be accommodated within a single OOZE text, encapsulated then in different modules.

5 Data Types

OOZE provides a large library of basic "built in" data types, intended to be rich enough for the vast majority of applications. For the most part, these are modelled after Z. However, it is important to provide a way for defining new data types in case those available in the library do not fit current needs. Data types in OOZE are defined in open-sided boxes in which the module name is preceded by the key word *Data*. Data modules can be parameterized and can import other data types. For example, consider the following definition of the parameterized data type *Seq* that defines sequences along with some of their basic operations; its parameter requirement is defined by the *Triv* theory below.

Theory Triv

$[X]$

Thus, any module P that satisfies *Triv* must have a set X.

Data $Seq[P :: Triv]$

$[Seq, Seq_1]$

$X \subset Seq_1 \subset Seq$

$\langle\rangle : Seq$
$\_ \frown \_ : Seq \times Seq \to Seq \quad [\text{assoc}, \text{id} : \langle\rangle]$
$\_ \frown \_ : Seq_1 \times Seq \to Seq_1 \quad [\text{assoc}]$
$head : Seq_1 \to X$
$tail : Seq_1 \to Seq$

$\forall S : Seq;\ x : X \bullet$
$head(x \frown S) = x$
$tail(s \frown S) = S$

$\#\_ : Seq \to \mathbb{N}$
$rev : Seq \to Seq$

$\forall S : Seq;\ x : X \bullet$
$\#\langle\rangle = 0$
$\#(x \frown S) = 1 + \#S$
$rev(\langle\rangle) = \langle\rangle$
$rev(S \frown x) = x \frown rev(S)$

Here Seq has just one constant, namely $\langle\rangle$, the empty sequence. Seq_1 is the set of all non-empty sequences. $X \subset Seq_1 \subset Seq$ indicates that an element of set X is a sequence and that a non-empty sequence is also a sequence. The operations $\frown$, $\#$ and *rev* respectively denote concactenation, lenght and reverse, while *head* and *tail* have their expected meanings. The key words "assoc" and "id:" indicate that the operations are associative and have an identity. The constant introduced after "id:" is an identity element for that operation. This is a *specification* for an abstract data type whose enriched version is actually built into OOZE. Its conventions are those of initial algebra semantics, as discussed in Section 6 below, rather than those of set theory.

Other basic data types that define naturals, integers, rationals, tuples, etc., along with their respective operations can be defined in a similar way[9].

6 Semantics of OOZE

Although object and data elements are distinguished in OOZE, they share the important common feature of inheritance. At the data level inheritance is subtype inclusion, while at the object/class level, inheritance is subclass inclusion. For example, *Dog* is a subclass of *Canine*, and *Canine* is a subclass of *Mammal*. Similarly, we may say that the naturals are a subset of the integers, and that the integers are a subset of the rationals. It is also important to provide semantics for operations, that is, for functions at the data level (such as addition, division and multiplication for naturals, integers, rationals, etc.), and for methods at the class level (for updating, interrogating and manipulating objects).

[9]See [14] for an introduction to specifying data types in a similar context. It is perhaps worth remarking that the reals cannot be defined using ordinary initial algebra semantics; however, floating point numbers can be defined this way, and other techniques can be used for the "real" reals if they are really desired.

Order Sorted Algebra (*OSA*) [12] gives a powerful theory of inheritance for both data and objects, as well as for overloading functions and methods. OSA also has an operational semantics that can be used to animate specifications under certain conditions [10]. See Section 6.1.

It is also important to give a precise semantics for the large grain programming features of OOZE. Although the details of this are too complex for this paper, they have already been worked out in developing semantics for Clear, OBJ and FOOPS. The essential ideas are that modules are theories (i.e., sets of sentences) over the order sorted, hidden sorted, equational institution [15], and that the calculus of such modules is given by colimits. Views are theory morphisms, and a generic module is a theory inclusion. Furthermore, modules that define structures and operations are distinguished from those that define requirements by whether or not they involve data constraints. Data constraints are also used to define abstract data types, which are distinguished from classes in that their sorts are not hidden. Data constraints generalize initiality.

OOZE builds on FOOPS[10], an object oriented specification, programming and database language conceived by Goguen and Meseguer [11]. Both OOZE and FOOPS take OSA as a basis for their semantics, and specifically FOOPS is used to animate OOZE. FOOPS, in turn, is implemented by a translation into an enrichment of OBJ3 [24]. Both the OOZE and FOOPS implementations are still under construction, and are expected to see completion during 1991.

6.1 Data Types

The basic syntactic unit of the functional part of OOZE is the *Data* module, which defines abstract data types, including their constructor and selector functions. Such modules can be understood on the basis of two different semantics, one denotational and the other operational. The former is based on OSA and the latter on order sorted term rewriting [10] (see [19] for a survey of term rewriting). Following OBJ3, built in functional modules can also be implemented directly in the underlying Lisp system; for example, OBJ3 numbers are implemented in this way.

Consider the specification of sequence of natural numbers obtained by instantiating the data type *Seq* introduced in Section 5 with the module *Nat* which gives the natural numbers, i.e. *Seq* [*Nat* $\{X \mapsto \mathbb{N}\}$]. The basic idea of the term rewriting operational semantics of OOZE is to apply the given axioms to *ground terms*, i.e., terms without variables, as left-to-right rewrite rules, and progressively transform them until a form is reached where no further axioms can be applied; this form is called a *normal* form. Let us take a specific ground term as an example, and let the symbol $\Rightarrow$ indicate that a rewrite rule has been applied:

```
rev(head( 1 ^ 2 ^ 3 ) ^ tail (4 ^ 5 ^ 6)) ⇒
rev(1 ^ tail (4 ^ 5 ^ 6)) ⇒
rev(1 ^ 5 ^ 6) ⇒
6 ^ rev (1 ^ 5) ⇒
⋮
6 ^ 5 ^ 1
```

Two basic properties of term rewriting systems are termination and confluence. A term rewriting system is *terminating* if there are no infinite rewriting sequences on ground terms, and is *confluent* if any two rewrite sequences of a given ground term can be continued to a common term. A rewriting system that is both terminating and confluent is called *canonical*.

An OOZE functional module is an equational specification consisting of an order sorted signature Σ, which gives the sort and function symbols, and a partial ordering on the sorts, plus a set $\mathcal{E}$ of

[10]FOOPS stands for Functional and Object Oriented Programming System. It was first introduced in 1987 at SRI International, and is now under development in the Programming Research Group [24].

equations which involve only the symbols in Σ. The class of all algebras that satisfy $\mathcal{E}$ has initial algebras, i.e., algebras that have a unique homomorphism to any algebra that satisfies $\mathcal{E}$. OOZE takes initial algebras as the denotational semantics of its functional modules [13].

If a set $\mathcal{E}$ of equations is confluent and terminating as a term rewriting system, then the set of all its normal forms constitutes an algebra which is initial among the algebras satisfying $\mathcal{E}$ [7], and so the denotational (initial algebra) and operational (term rewriting) semantics agree. Since an abstract data type is an isomorphism class of initial algebras, it follows that an OOZE functional module defines an abstract data type. By contrast, Z and other model based languages, including Object-Z, have *concrete* data types, which have excessive implementation bias.

6.2 Classes and Objects

Many aspects of the data level of OOZE are mirrored at its object level. For example, OSA gives meaning to class inheritance and method overloading. Also, equations are used to define how methods modify attributes. Note that the types for data and the classes for objects form entirely separate hierarchies in OOZE, each with its own partial ordering.

Some axioms about objects appear not to be satisfied by all of their intended models. For example, the implementation of a stack by an array and a pointer does not satisfy `pop(push(S,N)) = S` for all states `S`: If the state before a pop is `pop(push(push(push(empty,1),2),3))` then the state after the pop is `pop(push(S,4))`, which differs from the state before in that the number 4 occupies the position above the pointer, instead of the number 3. On the other hand, `top(pop(push(S,N))) = top(S)` is satisfied.

The solution is that rather than demanding axioms to be strictly satisfied, we only demand that their *visible consequences* are satisfied. This is justified by the fact that objects do not actually appear as such, because only their attributes are visible. While the functional level of OOZE has initial algebra semantics, the object level uses classes of algebras with the same observable behaviour as denotations; these need not be isomorphic to one another. More formally, given an order sorted signature Σ, a set of visible sorts and a set of hidden sorts, then two algebras are *behaviourally equivalent* if the result of evaluating any expression of a visible sort is the same for each of them [15, 9, 11].

Such algebras are *abstract machines* whose states are elements of hidden sorts; each object of a given class is a different copy of such a machine, with its own state, and creating an object produces a new copy of the machine in its initial state, while methods change the state, and attributes observe the state. When all sorts are visible, the concepts of abstract machine and abstract data type are identical. At the object level, OOZE takes behavioural equivalence classes of algebras as its denotational semantics. [11] gives an operational semantics based on the reflection of the object level into the data level; this provides on alternative way to support animation.

6.3 Theories and Views

Modern programming languages have many different kinds of entity, such as arrays, procedures, functions, operations, and records; hence, types are useful to separate and classify entities. The use of types helps to avoid meaningless expressions, and also makes it easier to understand code. In OOZE, theories are used to classify modules and to express requirements. This approach can be applied to both specification and programming languages, and is especially useful for building large systems.

Formally, a *theory* is a pair $(\Sigma, \mathcal{E})$; its denotation is the collection of Σ-algebras that satisfy $\mathcal{E}$. In OOZE, an algebra satisfies a data module if it is an initial algebra of the corresponding theory. At

the object level, behavioural satisfaction is used instead. In both cases, *views* are morphisms from one pair $\langle \Sigma, \mathcal{E} \rangle$ to another pair $\langle \Sigma', \mathcal{E}' \rangle$ such that each equation in the first is (behaviourally) satisfied in the models of the second. See [15] for an introduction to behavioural satisfaction, and [9] for a more comprehensive discussion.

7 Related Work

OOZE is not the first proposal for an object oriented extension of Z. The oldest work in this area seems to be that of Schuman, Pitt and Byers [23]. The semantics of that language is based on set theory, first-order logic, events and histories. There are many differences between OOZE and this proposal, one of which is the absence of a specific syntactic construction for classes. As a result, classes and their associated operations may be dispersed freely throughout a specification, and it may be hard to discover what dependencies exist.

Object-Z, which is being developed at the University of Queensland, Australia [6], is based on *class histories* for each object, which record the operations executed on it[11]. In Object-Z, class histories are not only restricted by class invariants, and operations by pre- and post- conditions, but also by history invariants, which are temporal logic predicates. Although temporal operators make Object-Z unique among the object oriented extensions of Z, they may be hard to animate. For example, predicates of the form *eventually* α *will occur* are not easily checked. Also, maintaining a history for each object would require significant amounts of memory and processing time. Moreover, the semantics of Object-Z seems not to be very precise; however, recent work of Cusak [5] goes some way towards filling this gap.

In comparison with Z and other languages based upon it, OOZE is more abstract, more flexible, and more compact. This is largely because the model-based semantics is too concrete for many purposes, and in particular is not very well matched to some aspects of object orientation. For example, it can be difficult to tell whether a subset relationship is intentional or accidental, and what implications it may have for implementation. Given a *Flag* class whose state is a finite set of natural numbers in a certain range, it may be unclear whether *Flag* must be a subclass of a previously defined class of finite sets of integers. As a result, it will be unclear whether or not flags can be implemented by arrays of bits.

IntSet

$st : \mathbb{F}\, \mathbb{Z}$

Flag

$st : \mathbb{F}\, \mathbb{N}$

$n \in st \Rightarrow 1 \leq n \leq N$

where N is a positive integer constant assumed to be previously defined.

Z and Object-Z do not support classes of objects, but only single instances of each class; that is, they conflate the notions of class and object. In particular, they do not support the creation and deletion of objects. As a result, when there is more than one instance of a class, specifications in Z and Object-Z can be considerably longer.

Some other significant differences arise from OOZE's powerful module facility. These include: localizing variables and operations to modules, yielding simpler scoping conventions and eliminating the "global variable problem" (which is that all variables have global scope); parameterized (or "generic") modules; module importation; module expressions; a distinction among classes, objects

[11] There are differences between this approach and that of [23].

and modules; a distinction among modules used for interfaces and requirements, modules used for defining classes, and modules used for defining data types; relegating schemas to a minor rôle; eliminating schema combinators; clearly distinguishing between module inheritance and class inheritance; and using views to express refinement, satisfaction and inheritance at the module level. Finally, some significant differences arise from the semantic foundation in order sorted, hidden sorted equational logic. These include: precise and general ways of defining and handling exceptions; operation overloading; precise notions of abstraction and encapsulation; and simple criteria for when a refinement is correct.

8 Conclusion

OOZE is a generalized "wide spectrum" object oriented language with both *loose* specifications and *executable* (compilable or interpretable) programs. These two aspects of the language can be encapsulated in *modules* and may be linked by *views*, which assert refinement relationships. Modules are organized according to an *import hierarchy*, and can also be generic, i.e., *parameterized*. A system of modules, which may be loose and can even have empty bodies, can be used to express the large-grain *design* of a system. A single, very high level module can be used to express overall *requirements*, via a view to a module that encapsulates the whole system, or at earlier stages of development, just its design or specification. Rapid prototypes can be developed and precisely linked to their specifications and requirements by views. The use of loose specifications to define interfaces can be seen as a powerful semantic type system. Theorem proving is supported by the underlying OBJ system. OOZE is truly object oriented, allowing varying numbers of objects to a class, and complex objects (i.e., object valued attributes), as well as multiple inheritance and dynamic binding. The precise semantics based on order sorted algebra supports exception handling and overloaded operations. These characteristics are unique among the proposals for extending Z, and along with its animation and database facilities, make OOZE a very attractive language for developing large systems.

References

[1] American National Standards Institute, Inc. The programming language Ada reference manual. In *Lecture Notes in Computer Science 155*. Springer-Verlag, 1983. ANSI/MIL-STD-1815A-1983.

[2] S. Bear. Structuring for the VDM specification language. In *Lecture Notes in Computer Science 328*. Springer-Verlag, 1988.

[3] R.M. Burstall and Joseph A. Goguen. The semantics of Clear, a specification language. In *Lecture Notes in Computer Science 86*, pages 292–332. Springer-Verlag, 1980.

[4] Dan Craigen. Position paper for FM 89. In *Proceedings of Formal Methods '89, Workshop on Formal Methods*, Halifax, Nova Scotia, Canada, July 1989.

[5] Elspeth Cusack. Inheritance in object oriented Z, November 1990. British Telecom.

[6] David Duke, Roger Duke, Gordon Rose, and Graeme Smith. Object-Z: An object-oriented extension to Z. In *Proceedings of Formal Description Techniques (FORTE'89)*, 1989.

[7] Joseph A. Goguen. How to prove algebraic inductive without induction: With aplications to the correctness of data type representations. In Wolfgang Bibel and Robert Kowalski, editors, *Lecture Notes in Computer Science 87*, pages 356–373. Springer-Verlag, 1980.

[8] Joseph A. Goguen. Parameterized programming. *IEEE Transactions on Software Engineering*, SE-10(5), September 1984.

[9] Joseph A. Goguen. Types as theories. In *Proceedings of Symposium on General Topology and Applications*, Oxford University, June 1990. To appear, Oxford University Press, 1991.

[10] Joseph A. Goguen, Jean-Pierre Jouannaud, and José Meseguer. Operational semantics of order-sorted algebra. In *Lecture Notes in Computer Science 194*. Springer-Verlag, 1985.

[11] Joseph A. Goguen and José Meseguer. Unifying functional, object-oriented and relational programming with logical semantics. Technical Report SRI-CSL-87-7, SRI International - Computer Science Lab, July 1987.

[12] Joseph A. Goguen and José Meseguer. Order-Sorted Algebra I: Equational deduction for multiple inheritance, overloading, exceptions and partial operations. Technical Report SRI-CSL-89-10, SRI International, Computer Science Lab, July 1989.

[13] Joseph A. Goguen, James Thatcher, and Eric Wagner. An initial algebra approach to the specification, correctness and implementation of abstract data types. In Raymond Yeh, editor, *Trends in Programming Methodology IV*, pages 80–149. Prentice Hall, 1978.

[14] Joseph A. Goguen and Timothy Winkler. Introducing OBJ3. Technical Report SRI-CSL-88-9, SRI International -Computer Science Lab, August 1988.

[15] Joseph A. Goguen and David Wolfram. On types and FOOPS. In *Proceedings of Working Conference on Database Semantics*, Windermere, Lake District, United Kingdom, July 1990. (To appear).

[16] Adele Goldberg and Alan Kay. Smalltalk72 instruction manual. Technical report, Learning Research Group, Xerox Palo Alto Research Center, 1976.

[17] Robert Harper, David MacQueen, and Robin Milner. Standard ML. LFCS Report Series ECS-LFCS-86-2, University of Edinburgh, March 1986.

[18] Ian Hayes, editor. *Specification Case Studies*. International Series in Computer Science. Prentice Hall International, 1987.

[19] Gerard Huet and Derek Oppen. Equations and rewrite rules: A survey. In Ronald Book, editor, *Formal Language Theory: Perspective and Open Problems*, pages 349–405. Academic Press, 1980.

[20] Bertrand Meyer. *Object-Oriented Software Construction*. Prentice Hall International, 1988.

[21] Lewis J. Pinson and Richard S. Wiener. *Object Oriented Programming and Smalltalk*. Addison-Wesley, 1988.

[22] Augusto Sampaio and Silvio Meira. Modular extension to Z. In *Lecture Notes in Computer Science 428*. Springer-Verlag, 1990.

[23] S.A. Schuman, David Pitt, and P.J. Byers. Object-oriented process specification. Technical report, University of Surrey, 1989.

[24] Adolfo Socorro. An implementation of FOOPS. Programming Research Group, Oxford University Computing Laboratory, 1990.

[25] J. Michael Spivey. *The Z Notation, A Reference Manual*. International Series in Computer Science. Prentice Hall International, 1989.