The Class Constructor in MooZ

Lin Tse Min  Silvio Lemos Meira
ltm@di.ufpe.br  srlm@di.ufpe.br

Departamento de Informática
Universidade Federal de Pernambuco
Caixa Postal 7851, Cidade Universitária
50732-970 – Recife/PE – Brasil

Abstract

The formal specification of software has been recognized as an important step towards the development of high quality and reliable software and plays a crucial role in reducing development costs.

The beginning of the 90's has seen many proposals of object-oriented extensions to the Z specification language. MooZ, one such extension, was proposed by the Formal Specification Group at DI/UFPE. MooZ includes the basic concepts of object-orientation: abstraction, encapsulation, modularity and hierarchy.

This paper describes a set-theoretic model for the class concept and shows how the semantics for the “kind of” inheritance and polymorphism in the MooZ language are defined. The model is proposed as a conservative extension of the well-known semantics of schema of Z. The type system for MooZ is presented and the model of signature, structures, varieties and environment proposed by Spivey for Z is expanded to give support to the class concept in MooZ.

Keywords: Formal Methods, Object-Orientation, Z, MooZ.
1 Introduction

It is widely recognized that the amount of effort spent in the development and maintenance of software is higher than it could be considered reasonable. One of the facts that leads to this situation is that the communication between the designer and other people involved in the process occurs in an informal or semi-formal way and many kinds of problems take place, as described in [19], making the developed software unreliable. As a consequence, a high amount of "bug fixes" must be done during its life time.

The massive use of software in safety-critical systems requires the use of high-quality and reliable software. The formalization of the software development process is one of the main approaches to the development of more reliable software. The use of formal specification as the first step in the process of software development makes possible the verification of properties, ambiguities and detection of design flaws. Formal specification is used, for example, in the specification of communication protocols (the LOTOS language [13]) and in the UK Ministry of Defence, where the standard 00-55 [25] requires the use of formal methods in the development of military software.

Among the notations available to the formal specification of software, Z [4, 12, 22, 23] is one of the most widely adopted. Its powerful structuring mechanism, the schema calculus, allows the composition of schemas into new ones. The Z notation has been used successfully in description of large systems [5, 20]. However, the modularity mechanism of Z, the schema, is not able to deal with such large descriptions as the amount of schemas defined make structuring and maintenance of specifications a very difficult task [2].

Z lacks a structure for grouping schemas in modules which would improve not only readability and structuring but also maintenance and reusability. In this way, many proposals of extensions to Z have been developed. The concepts of object-orientation [3] are found in great part of such proposals: MooZ, Z++, Object-Z and OOZE, for example. Some notations to write an object-oriented style of specifications in Z without introduction of new paragraphs have also been proposed: Hall's style, Schuman & Pitt and ZERO, to cite some of them. A comparative study of all these object-oriented variations of Z is presented in [24].

In MooZ, the concepts of object-orientation are incorporated to increase modularity, to ease maintenance and to improve reusability of MooZ specifications. Object-oriented concepts of class, object, inheritance, polymorphism and dynamic binding are part of the language. Polymorphism is perhaps the most powerful feature of the object-oriented paradigm. Polymorphism plays an important role in modularity and reusability; without it, inheritance is not very useful [3, pp. 104].

In this paper we describe a formal semantics for the class structure, inheritance and polymorphism mechanism in MooZ as an extension of the set-theoretic semantics of Z, as proposed by Spivey in [22]. Through some changes in the signature, structure, variety and environment model, it is possible to define the meaning of a schema in MooZ so that the class structure and occurrence of polymorphism and dynamic binding are considered. A new environment, known as "global environment", is proposed to record the meaning of each class in a MooZ specification.

The reader should have some minimal knowledge to obtain a satisfactory understanding of the work described in the next sections. We assume some background of Zermelo-Fraenkel's axiomatic set theory, mathematical logic, Z notation [12, 23] and MooZ [17, 18, 24]. It is strongly recommended the reading of [22] for a complete understanding of the changes proposed.

In the next section we give a short description of the MooZ notation. In section 3 we present how the semantics of a class in MooZ was given and how we deal with the "kind of" inheritance, polymorphism and dynamic binding concepts. In the last section we present our conclusions, related work and suggestions for further research.
2 MooZ

A specification in MooZ consist of a set of classes and every definition must be introduced in a class. This specification leads to a style in which a system (specification) is decomposed into subsystems with the objective of modularity and reusability.

In MooZ, each class may be considered as a specification of a complete system in Z, whilst a set of classes may be treated as a set of closed specifications in Z. There are two kinds of relationships among classes: using or inheritance relationship.

Theoretically, each class in MooZ defines an abstract data type, a model defining the state components, the initial states and the operations that change the state thus, maintaining a close relation to the method used in plain Z. MooZ is an extension of Z that does not invalidate nor change the meaning of the Z paragraphs. Besides Z paragraphs, MooZ includes four new paragraphs: class, anonymous schema, semantic operation and message.

2.1 The Class Structure in MooZ

The body of a class is composed of a sequence of clauses and their order is enforced. However, they are all optional and are described as follows:

Class \( (\text{Class-Name}) \)

givensets \( (\text{type-names-list}) \)

In some situations a class definition may use sets whose structure are not of interest at this particular abstraction level and so, they do not need to be modeled. The givensets clause may be used to introduce a list of these (given set) names. A given set may serve as an abstraction or genericity mechanism.

superclasses \( (\text{class-references-list}) \)

The superclasses clause in MooZ allows (multiple) inheritance. This clause introduces a list of superclass references. All of the definitions in a class are inherited by its subclasses. Name conflicts among superclasses must be solved by the user, through renaming.

public \( (\text{definition-names-list}) \)

private \( (\text{definition-names-list}) \)

The public and private clauses establish the definitions that are visible and not visible respectively. Each visible definition corresponds to a message that may be sent to an object of the class (object message) or to the class itself (class message).

constants

\( (\text{axiomatic-descriptions-list}) \)
\( (\text{auxiliary-definitions}) \)

The constants clause introduces the global constants that play an important role in the meaning of the class. Constants may be introduced in any clause and the purpose of the constants clause is to set apart the constants that have a special meaning in the class from the ones that are auxiliary and it is used to improve readability.

state

\( (\text{anonymous-schema}) \)

or

\( (\text{constraint}) \)

The state components of a class are those introduced in its superclasses and in its state clause.
initialstates

(schema)
(auxiliary-definitions)
The initialstates clause defines the initial state(s) of the class through initialization operation(s). Each operation specifies a possible initial state.

operations

(definitions)
(auxiliary-definitions)
The operations clause introduces the schemas and semantic operations that define the class operations along with auxiliary definitions, if necessary.

EndClass (Class-Name).

Case studies that illustrate the use of inheritance, polymorphism and dynamic binding can be found in [15, 17, 18, 24].

3 The Meaning of Classes in MooZ

In this section a formal semantics for the class structure in MooZ is presented. The first part describes MooZ's type system and then we modify the model of signature, structure, variety and environment associated to Z to give support to the class, polymorphism and dynamic binding concepts in MooZ.

3.1 Type System

The simpler types in MooZ, also known as basic types, are the names of the primitive or given sets. In almost all situations, it is sufficient to assume that naturals (N) and integers (Z), among others, are built-in primitive types of Z (and MooZ, too). But, as shown in [22, pp. 106–111] these primitive types can be specified in Z, and so in MooZ. Through the type constructors of set, cartesian product, schema and class, it is possible to construct very sophisticated types.

The abstract syntax of types in MooZ extends in a conservative way the type system of Z (see figures 1 and 2).

\[
\text{TYPE} ::= \text{givenT}((\text{NAME})) \mid \text{powerT}(\text{TYPE}) \mid \text{tupleT(seq TYPE)} \mid \text{schemaT}(\text{IDENT} \rightarrow \text{TYPE})
\]

Figure 1: Abstract Syntax of Types in Z

\[
\text{TYPE} ::= \text{givenT}((\text{NAME})) \mid \text{classT}(\text{NAME}) \mid \text{powerT}(\text{TYPE}) \mid \text{tupleT}(\text{seq TYPE}) \mid \text{schemaT}(\text{IDENT} \rightarrow \text{TYPE})
\]

Figure 2: Abstract Syntax of Types in MooZ

The formal definition presented in figure 2 corresponds to the following more informal notation:

\[
X \triangleq \text{givenT} X \text{ or } \text{classT} X
\]
\[
\mathbb{P}a \triangleq \text{powerT} a
\]
\[
a_1 \times \ldots \times a_n \triangleq \text{tupleT} (a_1, \ldots, a_n)
\]
\[
\{x_1 : a_1; \ldots; x_n : a_n\} \triangleq \text{schemaT} \{x_1 \mapsto a_1, \ldots, x_n \mapsto a_n\}
\]

We do not make an explicit distinction between givenT X and classT X. To the client (in an instantiation or using relationship) the structure of the class used in the definition of type has no
interest and it may be regarded as a primitive (or basic) type. The main difference between an
element $e_1$ of a primitive type $givenT X_1$ and an element (or object) $e_2$ of type $classT X_2$ is that
the object $e_2$ has pre-defined operations (or methods) defined in the class $X_2$ while the element $e_1$
does not have any pre-defined operations.

### 3.2 Signature, Structure and Variety

As in Z, the signature of a schema records all the necessary names to the definition of the schema
variable's type. A new component $class$ is now needed to record the names of classes used in
definition of types:

```
SCHEMASIG
classes : NAME
vars : NAME

given : NAME

(type $\in$ (vars $\rightarrow$ Type(given $\cup$ classes))) $\land$ (given $\cap$ classes = $\emptyset$)
```

The axioms require the type assigned to a variable to be formed from the previously defined given
classes and set names.

In the same way, a signature of a class records the names of classes and given sets used in the
definition of types, the names of its superclasses, constants, state variables and other definitions
introduced in the class:

```
CLASSSIG
classes, given : NAME
vars : NAME

opers : WORD $\rightarrow$ SCHEMASIG

parents : WORD
defs : WORD $\rightarrow$ PAR

consts, vars : NAME

abrev : WORD $\rightarrow$ ABREVDEF

state : SCHEMASIG

(state. classes $\subseteq$ classes)
(state. given $\subseteq$ given)
(state. vars = consts $\cup$ vars)
dom abrev $\cap$ word([ baseids((classes $\cup$ given $\cup$ consts $\cup$ vars))]) = $\emptyset$
dom opers $\cup$ dom abrev $\subseteq$ dom defs

Tag 0 ( | paint ( | word([ baseids((classes $\cup$ given $\cup$ vars)) ])) ) blank ] = classes
Tag 0 ( | paint ( | word([ baseids(given) ])) ) blank ] = given
Tag 0 ( | paint ( | word([ baseids(consts) ])) ) blank ] = consts
Tag 0 ( | paint ( | word([ baseids(vars) ])) ) blank ] = vars

$\forall m :$ dom opers $\bullet$

((opers m).classes $\subseteq$ classes) $\land$ (basenames ((opers m).given) $\subseteq$ given)
word ( | baseids((opers m).vars) $\triangleright$ {pre} ) = word ( | baseids(vars) )
word ( | baseids((opers m).vars) $\triangleright$ {post} ) = word ( | baseids(vars) )
```

The axioms guarantee that:

1. It is not possible to define variables and constants that have the same name;
2. The names of classes, given sets, state variables and constants are undecorated (a blank deco-
   ration means absence of decoration) and tagged at level 0;
3. To each name in the dictionary of schemas' opers:
(a) the names tagged at level 0 must be part of the global names;
(b) all of the identifiers decorated with _pre_ or _post_ must be the name of a state variable.
All of these variables are present in the signature of an operation, even if they are not
explicitly introduced, since they are implicitly declared in a \( \Sigma \)-list.

In each structure that satisfies the schema’s axiom part, there is a function that gives the dynamic

type of each variable of the schema which type is defined by a class:

\[
\begin{aligned}
gset &: \text{NAME} \rightarrow W \\
dtype &: \text{NAME} \times \text{NAME} \rightarrow \text{NAME} \\
\forall x : \text{dom } dtype \land (dtype \ x \in \text{dom } gset) & \land (\exists v : \text{dom } val; \ c : \text{dom } gset\ | \ x = (v, c))
\end{aligned}
\]

In _SCHEMAGRAPH_, it is not possible to ensure that the domain of the function _dtype_ is composed
solely of the variables whose type is defined by a class. This condition is enforced by the function
_struct_ defined as follows:

\[
\begin{aligned}
\text{struct} &: \text{SCHEMASIG} \rightarrow \mathbb{P} \text{SCHEMAGRAPH} \\
\text{struct} &= \lambda \Sigma : \text{SCHEMASIG} \to \\
\{ s : \text{SCHEMAGRAPH} | \\
& (\text{dom } s.gset = \Sigma . \text{classes} \cup \Sigma . \text{given}) \land (\text{dom } s.val = \Sigma . \text{vars}) \\
& \forall v : \Sigma . \text{vars} \ | (\neg \exists C : \Sigma . \text{classes} \land \Sigma . \text{type } v = \text{class} T (C)) \land \\
& (s . \text{val } v \in \text{Carrier } s . \text{gset } (\Sigma . \text{type } v) \\
& \forall v : \Sigma . \text{vars} \ | (\exists C : \Sigma . \text{classes} \land \Sigma . \text{type } v = \text{class} \ T (C)) \land \\
& ((v, C) \in \text{dom } s . \text{dtype} ) \land (s . \text{val } v \in \text{Carrier } s . \text{gset } \text{class} \ T (s . \text{dtype } (v, C)) \\
& \forall x : \text{dom } s . \text{dtype} \land \\
& (\exists v : \Sigma . \text{vars}; \ C : \Sigma . \text{classes} \land \Sigma . \text{type } v = \text{class} \ T (C) \land \ x = (v, C))
\end{aligned}
\]

The auxiliary function _restrict_ must be changed to deal not only with the new signature of a schema
but also with the presence of polymorphism in _MooZ_.

\[
\begin{aligned}
\text{restrict} &: \text{SCHEMASIG} \rightarrow \text{SCHEMAGRAPH} \rightarrow \text{SCHEMAGRAPH} \\
\text{restrict} \ \Sigma \ M \cong \\
\mu M' : \text{struct } (\Sigma) \ | \\
& (M'.gset = (\Sigma . \text{classes} \cup \Sigma . \text{given}) \triangle M . gset) \land (M'.val = (\Sigma . \text{vars}) \triangle M . val) \\
\forall c : \text{dom } M'.gset; \ v : \text{dom } M'.val \ | (v, c) \in \text{dom } M . \text{dtype} \land \\
& ((v, c) \in \text{dom } M'.\text{dtype}) \land (M . \text{dtype}(v, c) = M'.\text{dtype}(v, c))
\end{aligned}
\]

The axiom in (1) guarantees that the dynamic type of each variable in _M_ is the same as in _M_.
As expected, the meaning of a schema in _MooZ_ is the same as in _Z_.

\[
\begin{aligned}
\text{sig} &: \text{SCHEMASIG} \\
\text{models} &: \mathbb{P} \text{SCHEMAGRAPH} \\
\text{models} \subseteq \text{struct } (\text{sig})
\end{aligned}
\]

The meaning of a class in _MooZ_ is similar to that of an environment in _Z_, since to make sense
a class must record all names necessary to the definitions introduced in it. In this way, we call the
variety of a class a “local environment”: 
The meaning of a class recorded in the variety of a class is also almost the same as in $Z$ (we must deal with identifiers of class):

\[
\text{SCHEMAMEANING local : SCHEMAMANIECY}
\]

\[
fparam : \text{seq IDENT}
\]

\[
fparam^{-1} \in (\text{localids(local.sig.given)} \rightarrow \mathbb{N})
\]

\[
\text{local.sig.classes} \cup \text{local.sig.given} \cup \text{local.sig.vars} \subseteq \text{basenames(local.sig)} \cup \text{localnames(local.sig)}
\]

\[
\text{local.sig} \in \text{dom basis}
\]

As in $Z$, the semantic domain $GENDEFMEANING$ for generic definition is the same as that for schema: $GENDEFMEANING \equiv \text{SCHEMAMEANING}$.

### 3.3 Environment

A MooZ class is not a "closed" entity as it may be used in two ways: instantiation or inheritance. Thereafter it is necessary to record the name and definition of each class previously specified. An environment, $ENV$, has been defined in MooZ to handle classes and make their occurrences consistent in other classes:

\[
\text{ENV classes : WORD } \rightarrow \text{CLASSMEANING}
\]

\[
\forall c : \text{dom classes} \mid \#(\text{classes c}).fparam \neq 0 \bullet
\]

\[
(\text{classes c}) \in \{ CM : \text{CLASSMEANING} \}
\]

\[
CM\.local.sig \text{classSubSIG (classes c)}.local.sig
\]

\[
(\forall M : (\text{classes c}).local.models; \text{aparam} : \text{seq W} \mid
\#	ext{aparam} = \#CM.fparam \bullet
\]

\[
\exists M' : CM\.local.models \bullet
\]

\[
\text{restrict (basis (CM\.local.state.sig)) M} = \text{restrict (basis (CM\.local.state.sig)) M'}
\]

\[
M'.gset \circ (\text{Tag 0} \circ CM\.fparam = \text{aparam})
\]

A class that introduces given set names corresponds to a generic definition, and so, the uniqueness condition in axiom (1) must be guaranteed because an axiomatic description introduced (in the class) may constrain the elements allowed for the given set.\(^1\)

\(^1\)This problem is described fully in [22, pp. 83–89] and [16, pp. 76–78].
The meaning of a class recorded in the environment is defined as follows:

\[ \text{CLASSMEANING} \]

\[
\begin{align*}
\text{local} & : \text{CLASSVARIETY} \\
\text{fparam} & : \text{seq IDENT} \\
\text{fparam}^{-1} & \in ((\text{Tag} \ 0)^{-1} (\text{basenames(local.sig.state.given)})) \rightarrow \text{N}
\end{align*}
\]

The sequence \text{fparam} records the order of the class’ formal generic parameters (or given set identifiers).

### 3.4 Polymorphism in MooZ

Besides inclusion polymorphism, \textit{MooZ} adopts the overloading and the parametric form of polymorphism present in \textit{Z} [26]. To understand how the entities previously defined were used to define inclusion polymorphism in \textit{MooZ}, we present here the semantic function for declarations (of polymorphic identifiers) in \textit{MooZ}, whose abstract syntax is given as follows:

\[
\text{DECL} ::= \text{IDENT} : \text{TERM} \\
| \text{SDES} \\
| \text{DECL} ; \text{DECL}
\]

\[
\text{decl} : \text{ENV} \rightarrow \text{WORD} \rightarrow \text{CLASSVARIETY} \rightarrow \text{LEVEL} \rightarrow \text{DECL} \rightarrow \text{SCHEMAVARIETY}
\]

\[
\begin{align*}
\mu \text{cv'} : \text{CLASSVARIETY}; \ tt : \text{TERMMEANING} & | \\
\text{cv'.sig.classes} & \supseteq \text{cv.sig.classes} \\
\text{cv'.sig.classes} & \subseteq \text{Tag} \ 0 \ (\text{paint} \ w \ \text{blank}) \ (\text{dom} \ \text{p.classes}) \cup \ w \ (\text{blank}) \\
\text{cv'.sig.given} & = \text{cv.sig.given} \land \text{cv'.sig.parents} = \text{cv.sig.parents} \\
\text{cv'.sigconsts} & = \text{cv.sigconsts} \land \text{cv'.sig.vars} = \text{cv.sig.vars} \\
\text{cv'.sdict} & = \text{cv.sdict} \land \text{cv'.gdict} = \text{cv.gdict} \\
\text{cv.models} & = \text{restrict cv.state.sig} \ (\text{cv'.models}) \\
\text{Tag} \ 0 \ (\text{paint} \ w \ \text{blank}) & \in \text{cv'.sig.classes} \Rightarrow \\
\forall \ M' : \text{cv'.models} & \bullet \\
M'.\text{gset} \ (\text{Tag} \ 0 \ (\text{paint} \ w \ \text{blank})) & = \text{representation} \ (\text{cv'}) \\
\forall \ M' : \text{cv'.models} & \bullet \\
\forall \ c' : \text{cv'.sig.classes} & | \ c' \neq \text{Tag} \ 0 \ (\text{paint} \ w \ \text{blank}) \bullet \\
M'.\text{gset} \ (c') & = \text{representation} \ ((\text{p.classes} \ \text{word}((\text{Tag} \ 0)^{-1} c')).\text{local}) \\
\forall \ x : \text{dom} \ M'.\text{dtype} & \bullet \\
\exists \ v : \text{cv'.sig.vars}; \ c : \text{cv'.sig.classes} & | \ x = (v, c) \\
(1) \downarrow \\
M'.\text{dtype}(v, c) & \in \\
(\text{Tag} \ 0 \ (\text{paint} \ w \ \text{subclasses} \ \text{word}((\text{Tag} \ 0)^{-1} c)), \ w \ (\text{blank})) \cup \ (c) \\
(2) \downarrow \\
\forall \ c' : (\text{Tag} \ 0 \ (\text{paint} \ w \ \text{subclasses} \ \text{word}((\text{Tag} \ 0)^{-1} c)), \ w \ (\text{blank})) \cup \ (c) \bullet \\
\exists \ M' : \text{cv'.models} & \bullet (M'.\text{dtype}(v, c) = c') \land (M'.\text{val}(v) \in M'.\text{gset}(c'))
\end{align*}
\]

\[
\text{tt} \equiv \text{term} \ \rho \ \text{w} \ \text{cv'} \ k \ [t] \bullet \\
\mu \ a : \text{TYPE} | \ tt.type = \text{powerT} \ a \bullet \\
\text{new_var} \ (\text{cv'.state}, \text{Tag} \ k \ \text{id}, a, \ tt.eval)
\]
In the predicate that follows (1), we make sure that the dynamic type of a variable may be its static type or a type defined by one of the subclasses of its static class. In (2) we guarantee that whatever dynamic type a variable may assume, its value is always an element of the class that defines its (dynamic) type.

The function \( \text{representation} \) is used to get a representation of a class in the world of sets \( W \), and it is defined as follows:

\[
\text{representation} : \ \text{CLASSVARIETY} \rightarrow W
\]

\[
\text{representation} (cv) \ni \mu X : W \mid
\]

\[
\forall M : cv.models \ni \exists f : \text{IDENT} \rightarrow W \mid (\text{dom} f = (\text{Tag} 0)^{-1}(\text{dom} M.\text{val})) \land (\forall z : \text{dom} f \ni f z = M.\text{val} (\text{Tag} 0 z)) \rightleftharpoons \text{sproduct} (f) \in X
\]

\[
\forall x : W \ni x \in X \ni \exists M : cv.models; f : \text{IDENT} \rightarrow W \mid (\text{dom} f = (\text{Tag} 0)^{-1}(\text{dom} M.\text{val})) \land (\forall z : \text{dom} f \ni f z = M.\text{val} (\text{Tag} 0 z)) \rightleftharpoons \text{sproduct} (f) = x
\]

The function \( \text{subclasses} \) plays an important role in our definition. It is used to specify the concept of polymorphism in \( \text{MooZ} \), preventing "dynamic" type checking errors and it is defined as follows:

\[
\text{subclasses} : \ \text{WORD} \times \text{ENV} \rightarrow \mathcal{P} \text{WORD}
\]

\[
\text{subclasses} = \lambda w : \text{WORD}; p : \text{ENV} \mid w \in \text{dom} p.\text{classes} \ni \{ c : \text{dom} p.\text{classes} \ni w \in (p.\text{classes} c).\text{parent} \}
\]

\[
(p.\text{classes} c).\text{local}.\text{sig} \text{classSubSIG} (p.\text{classes} w).\text{local}.\text{sig}
\]

The relation \( \text{classSubSIG} \) is defined to capture our understanding of (internal) "behavioural compatibility" [27] between a class and its subclasses:

\[
\text{classSubSIG} \ni \ni \ \text{CLASSSIG} \leftrightarrow \text{CLASSSIG}
\]

\[
\Sigma_1 \text{classSubSIG} \Sigma_2 \ni \ni \Sigma_1.\text{state} \text{schemaSubSIG} \Sigma_2.\text{state} \ni \forall \text{oper} : \text{dom} \Sigma_1.\text{opers} \ni \ni (\Sigma_1.\text{opers} \text{oper}) \text{schemaSubSIG} (\Sigma_2.\text{opers} \text{oper})
\]

\[
\text{word}[] \text{localids}((\Sigma_1.\text{opers}).\text{vars}) \ni \{ \text{in} \} = \text{word}[] \text{localids}((\Sigma_2.\text{opers}).\text{vars}) \ni \{ \text{in} \} \ni \text{word}[] \text{localids}((\Sigma_1.\text{opers}).\text{vars}) \ni \{ \text{out} \} = \text{word}[] \text{localids}((\Sigma_2.\text{opers}).\text{vars}) \ni \{ \text{out} \}
\]

A given class \( C' \) is (internal) behaviourally compatible with its superclass \( C \) only if it satisfies the following rules:

1. \( C' \) does not change the visibility of any public method inherited from \( C \);

2. Any method inherited from \( C \) and redefined in \( C' \) must keep its original public interface preserved. A public interface of a schema is composed by all local variables that are decorated with "?" (\( \text{in} \)) or "!" (\( \text{out} \)). The public interface of an axiomatic description or semantic operation is its signature.
A complete description of a full semantics for the MooZ notation is presented in [16]. The properties of Z's semantic model, described in [22, pp. 25,30,36,61–62,76–78], continue to hold for the class type in MooZ and the proofs are presented in [16].

3.5 Dynamic Binding in MooZ

To understand how dynamic (or late) binding occurs in MooZ, the meaning of a term that defines object messages is used. As in Z, two inference systems are used to deduce the meaning of a term: the first system is used to reason about the type part of the meaning (figure 3) and the second is used to define the evaluation function of the term (figure 4). In the evaluation rule we may observe that the instantiated method comes from the class which determines the dynamic type of the object.

$$\rho, w, cv, k \vdash id :: P C$$
word(id) $\in$ cv.state.sig.vars
$$(C' = \text{word } ((\text{Tag } 0)^{-1} C)) \land (C' \in \text{dom}(p\text{.classes}))$$
$$w_1 \in \text{dom}(p\text{.classes } C').\text{local.gdict}$$
$$sv \triangleq \text{instantiate } (\text{cv.state}, k, (p\text{.classes } C').\text{local.gdict } w_1,\text{ }$$
$$\text{map } (\text{term } \rho \ w \ cv \ k) \langle t_1, \ldots, t_n \rangle)$$
$$sv\text{.sig.type } (\text{Tag } k (\text{paint } w_1 \text{ blank})) = a$$

$$\rho, w, cv, k \vdash id \ w_1 \langle t_1, \ldots, t_n \rangle :: a$$

Figure 3: Typing Rule - Object Message

In figure 3, the identifier id is used to obtain the class $C'$ that defines the static type of id and introduces $w_1$, a generic definition. It is important to note that the type of a message sending is static, whereas the type of the object (id) can be dynamic.

$$\rho, w, cv, k \vdash id :: C$$
$C' = \text{word } ((\text{Tag } 0)^{-1} C)$$
$C' \in \text{dom}(p\text{.classes})$
$$w_1 \in \text{dom}(p\text{.classes } C').\text{local.gdict}$$
$$\exists C'': \text{subclasses } (C', \rho) \cup (C') \ |$$
$$\text{M.dtype } (id, C) = \text{Tag } 0 (\text{paint } C'' \text{ blank}) \bullet$$
$$\text{sv } \triangleq \text{instantiate } (\text{cv.state}, k, (p\text{.classes } C'').\text{local.gdict } w_1,\text{ }$$
$$\text{map } (\text{term } \rho \ w \ cv \ k) \langle t_1, \ldots, t_n \rangle)$$
$$\exists M': \text{sv.models } \bullet$$
$$\text{restrict } \text{cv.state.sig } M' = M$$
$$M'.\text{val}(\text{Tag } k (\text{paint } w_1 \text{ blank})) = u$$

$$\rho, w, cv, k, M \vdash id \ w_1 \langle t_1, \ldots, t_n \rangle$$

Figure 4: Evaluation Rule - Object Message

In the evaluation rule (figure 4), the result of the message sending is gotten through instantiation of the generic definition $w_1$ (introduced in the class $C''$ that defines the dynamic type of id).
4 Concluding Remarks

MooZ extends the Z specification language with the concepts of abstraction, encapsulation, modularity and hierarchy, making possible an object-oriented style of writing specifications, improving readability, modularity, reusability and maintenance of specifications.

This work has some important aspects:

- Even though there are many object-oriented extensions to Z, only the proposals of Z++, OOZE, Object-Z and MooZ had their formal semantics for class defined;

- We have shown that MooZ is a conservative extension of Z as the meaning of the Z's paragraphs are preserved in MooZ;

- Even the best informal description makes room for misunderstanding (ambiguity), mistakes, over specification, to cite but a few, as shown by Meyer in [19]. The formal semantics given herein states clearly how to use and what is the exact meaning of the class paragraph in MooZ;

- We have described a formal semantics to the "kind of" inheritance relationship. It is important to notice that in the proposed semantic model we do not require any operation compatibility between a superclass and its subclasses, we guarantee that a variable of type C, where C is a class, may assume dynamic type C', where C' is a subclass of C, only if C' is (internal) behaviourally compatible with C. [16] presents the semantics of the "is a" inheritance relationship for the MooZ notation.

- The formal semantics of MooZ is essential for the development of tools and methods, since it shows how verification of scope and type should occur in MooZ, and it is a required step towards a definition of a theory of refinement and proof for MooZ.

Alencar observes in [1] that OOZE can be considered a syntactic variant of FOOPS [11], and thus, FOOPS can be used to animate OOZE's specifications. OOZE adopts the semantics of the object-oriented language FOOPS and, thus taking order sorted algebra as a basis for its semantics. Lano91 propose in [14] an algebraic semantics for the Z++ notation.

Object-Z was developed at the Department of Computer Science of The University of Queensland, Australia. Object-Z introduces a “class schema” which captures the class notion in an object-oriented concept. In [8] and [21] a set-theoretic model of classes in Object-Z was defined. The environment of Object-Z's definitions was not described and the meaning of a class is a set of histories, where each history is a sequence of events; the meaning of a class C in MooZ, described in section 3.2, is the set of all possible instances (or objects) of C. The type system for Object-Z is presented in figure 5.

[9] describes a full semantics for oo-Z, another object-oriented extension to Z, based in Object-Z. The class structure in Object-Z is quite the same of an oo-Z class, however, Object-Z contains some features not found in oo-Z. For example, mechanism to rename, redefine and remove operations from a class. On the other hand, oo-Z has operations for combining classes (\(\Lambda, \lor, \cup, \cap\)) that are not found in Object-Z.

From the work described in this paper, it is possible to develop some extensions:

---

2Object-Z adopts the "is a" inheritance relationship.

3An event is an operation together with an instance of the state of that operation. This instance represents the state changes undergone by an object both before and after the operation.
Our goal here was to present a clear and easy way of understanding the semantics of MooZ, and not specifically about the difficulty of developing a proof theory for MooZ. Spivey’s semantics for Z, as pointed by Gardiner (who developed a simpler approach in [10]), presents a number of difficulties for the research into a sound logic for Z. An important piece of research here would be to rewrite MooZ’s semantics in Gardiner’s style, in order to ease the task of devising and proving the soundness of a logic for MooZ. Another approach consist of use the Z Base Standard [4] in the development of a new semantics for MooZ.

2. The presented semantics is not fully abstract; therefore, the definition of a fully abstract semantics [28] for the class structure in MooZ is important as it makes possible to compare structurally incompatible classes through their external behaviour;

3. Another research direction includes the definition of a theory of refinement and proof for MooZ, which may be used in the development of tools to the (semi-)automatic generation of executable programs from MooZ specifications. Some steps toward this subject has been done and it is presented in [6].

With the aim of making MooZ widely available, it is very important to develop tools that support the process of development of specifications in MooZ. One such tool is under development at DI/UFPE.

References