Università degli Studi di Roma Tre
Dipartimento di Informatica e Automazione
Via della Vasca Navale, 79 – 00146 Roma, Italy

Object-Oriented Extension of a Functional Design Language.

Glauco Cenciotti, Claudio Morgia and Alberto Paoluzzi

RT-DIA-45-1999 **Dicembre 1999**

ABSTRACT

In this work a first approach to an object-oriented extension of the `PLaSM` functional language is discussed. This extension, denoted as `OO-PLaSM`, is intended to provide a representation for hierarchical networks of specializations of geometric object classes. An object class is here identified with the image set of a generating functions, whose expression *body* is here assumed as the class constructor. The language defines two new types of local functions, denoted as `feature` and `control`, over the classes. Such functions are used to produce modified values and to perform domain controls over the set of values of the class, respectively. `OO-PLaSM` allows the user to define variants of classes very simply. Generating expressions, their parameters, features and controls can be inherited or ridefined in the specialization hierarchy. The introduced extensions let `OO-PLaSM` be compatible with previous version of `PLaSM` (denoted as *classic* `PLaSM`).

1 Introduction

In the following sections the main motivations of this project will be discussed, that is why we want to extend `PLaSM` language with $Object-Oriented$-like expressions and why such an approach is considered to be better than others for describing geometric knowledge.

`PLaSM` is a geometric calculus oriented functional language. Anyway, an $Object-Oriented$ geometric language doesn't exist at the moment. Complex objects description can be actually done by C++ language with geometric libraries support, but its usage is still very difficult and only expert programmers can manage its complexity. Same considerations are valid for *Java3D* APIs. From an user point of view `PLaSM` is characterized for easily usage in building parametric forms, hierarchical assemblage and animations. On the other side, the descriptive power of many computer graphics languages, such as *VRML*, is much inferior to `PLaSM`. Even if some geometric libraries give useful interfaces towards functional languages, like *OpenGL* versus *Ocaml* or *ACIS* with *Scheme/elk*, programming with such environments is still too verbose.

Unlike other approches, `PLaSM` can be used not only for generating geometric objects, but also for defining new representation methods and/or new geometric operations. We can define it as a fully extensible and customizable language. In addiction, `PLaSM` is intrinsically multidimensional so that we can describe vertices and cells attributes (scalar fields, vectorial fields and tensorial fields) in a natural way. It also represents moving manifold and any type of kinematism in a natural and compact way, and it lets the user define and manipulate space-time descriptions.

Our goal, now, is to trace a path to `PLaSM` extension such to introduce geometric knowledge description in it. Human knowledge is characterized to be associative, that is it tries to group different realities by common features. Knowledge is so described by sets of atomic problems, each of which fully solves only few aspects of a more complex reality. In addiction more complex problems can be represented as compositions and/or specializations of subproblems. In this way an $Object-Oriented$ approach seems to faithfully describe knowledge. Our intention is to apply these concepts to geometric design and to realize a geometric functional language being able to help and guide users to build complex geometric objects through the set of elementary objects previously defined in the environment (assumed as the starting knowledge). New knowledge is so added by simply defining new objects and functions, with relative constrains over them, as composition of others or specialization (by derivation) of some of the existing ones.

The proposed OO extension for this language seems to fit quite well the requirements for knowledge based geometric design and fabrication in specialized application dominions. The symbolic description results synthetic and easily readable even for non programmers users. At the same time we want to *open* `PLaSM` to external environments, introducing only one simple primitive, for driving the language towards a much more collaborative and distributed designing environment. In the following we refer to the OO extension of the language as `OO-PLaSM`. The previous version will be often referred to as *classic* `PLaSM`.

In Section 2 the main properties of `OO-PLaSM` (inheritance, variants, external interfaces) are introduced. In Section 3 the *EBNF* syntax of the new primitives is given. Some important aspects about semantics and implementation of new primitives are discussed in Section 4 and Section 5, respectively. In Section 6 an example of `OO-PLaSM` programming is finally shown, and compared with its *classic* `PLaSM` implementation.

2 Design requirements

In this section the main design requirements of the proposed language extension are summarized and discussed. They respectively concern the partitioning of top-level `PLaSM` functions into four subsets and some design specification for object constructors, inheritance, variant definition and external interfaces.

2.1 Function partitioning

Top-level `OO-PLaSM` functions are partitioned into four disjoint subsets, and in particular into basic operations (specified with keyword `DEF`), classes (with keyword `CLASS`), internal operations (with keyword `FEATURE`) and external operations (with keyword `CONTROL`).

A *class* is defined as the *image set* of a function of `CLASS` type. In particular, a class is generated by a special type function, usually with formal parameters, which produces labeled values, called the *objects* of that class. In the following paragraphs we will use the word *class* for denoting both the generating function of the class and the set of generated objects. For sake of brevity, a further specification will be omitted when the meaning is clarified by the context.

Basic operations are functions that are defined over basic types. *Features* and *Controls* are unary restricted function that are defined as internal or external to a class, respectively. The different kind of functions are pictorially displayed in Figure 1.

2.2 Constructors

A class constructor is the `CLASS` function itself, which uses the generating form coded in the function body. This expression can include both the parameters declared in the class or in some of its superclasses as well as the external symbols declared in the current functional environment. A functional dependancy network will be consistently maintained by the `OO-PLaSM` interpreter, as *classic* `PLaSM` already does.

2.3 Inheritance

Controls and features can take full advantage of hierarchical class inheritance. So you can apply a feature or a control defined over a class that is a superclass of the one you are using. The interpreter behaviour is quite similar to the one given in other OO languages. When the invoked function (method) is not available in the current class it will look for the function with the same name defined in superior classes, choosing the one related to the nearest superclass in the hierarchy. If it cannot find such a function it will return an *exception* value.

Inheritance cannot be multiple, since each class can have at most one direct superclass in `OO-PLaSM`. In other words, the full class hierarchy is either a *tree* or a *forest*, but not a more general DAG (direct acyclic graph). At the moment it seems not useful to allow for multiple inheritance in the definition of mechanical objects or assemblies. Allowing for multiple inheritance would greatly complicate the language implementation without paying with increased expressive power in the design domain.

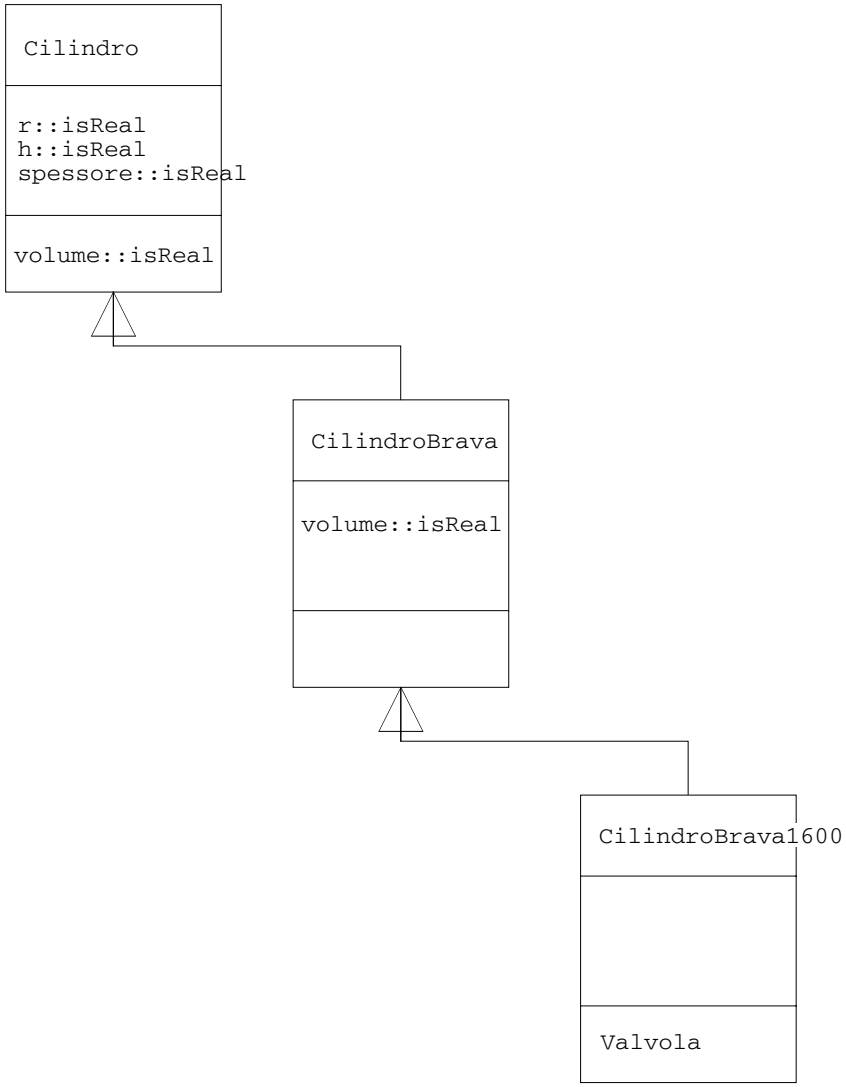

Figure 1: Different kind of functions as introduced in OO-PLaSM

2.4 Variants

These are `CLASS` type functions defined without any generating expression, that is without a *body* expression. The classes defined in this way inherit the *body* from their nearest superclass. Such classes are called *variant* classes. The use of a variant is justified by the will of the user to redefine either some or all the `FEATURE`s and/or the `CONTROL`s related to some subset of values of a specific class, without loosing the old ones. Notice that the *initial classes*, i.e. the ones without any superclass, *must* have a generating expression.

2.5 Extensibility

The language is easily extended by external form inclusion. The only primitive to introduce is

`EXTERNAL`: *target* : *expr*

where *target* denotes an external software environment, such as *ACIS* or *Scheme* or *Python*, and *expr* denotes a valid symbolic form for the *target* language or environment. Such a primitive will return the value generated by the evaluation of the *expr* expression in the *target* environment. In this way `OO-PLaSM` will interact with external environments, but only if they provide an interpreter and they allow to access to the internal values generated by the evaluation of expressions.

The `EXTERNAL`primitive should be freely combined with generic `PLaSM` expressions, even empty. Consequently the critical point is to hold conversions of different types returned by external environments, in particular for geometric values. Among others, native types to convert will be strings, numbers and number lists, sequences and comunication mechanisms (like sockets, streams and files).

2.6 Interface towards `ACIS` geometric kernel

A further design choice concerned the interoperability of the designed `PLaSM` extension with external kernel modelers. In particular we decided to interface `PLaSM` with the ACIS C++/Scheme geometric kernel. Hence, the `OO-PLaSM` extension will be provided with a hybrid geometric representation such that it will be possible to obtain, on request, either a `HPC` (Hierarchical Polyhedral Complex) based representation or an equivalent `ACIS` internal representation. For this purpose, a new C++ version of $\texttt{Simple}_X^n$kernel, developed at Roma Tre, will be used. This library will contain a new class, named `Ibrid`, whose interface will contain a couple of pointers to the two different representations. Consequently, an `ACIS` operation, for which we want an `OO-PLaSM` interface, have to be implemented as a new method of this class, which will supply also convertion methods between the two representations. In this case the primitive `EXTERNAL` will be the gateway to access to `ACIS` operations, by using the Scheme wrapping mechanism made available by ACIS. Such a loose coupling mechanism between `PLaSM` and ACIS can be made completely transparent to the user by inserting new ad-hoc operators in the basic functional environment of the language.

Table 1: Syntactic conventions

Symbol	Interpretation
UPPER CASE	terminal
italic	non-terminal
|	"or"
$\{\ldots\}$	grouping
$[\ x\ ]$	expression x repeated 0..1 times
$[\ x\ ]^*$	expression x repeated 0..n times
$\{\ x\ \}^y$	expression x repeated 1..n times, with separator y

3 Sintax specification

An Extended BNF specification for the OO-PLaSM extensions is given in this section. This specification is actually incomplete, but it is sufficient to give the syntactic flavour of the new primitives of the extended language. Further changes could be introduced in the next future on the given syntax.

3.1 Used conventions

The conventions for EBNF productions introduced in the following subsections are given in Table 1.

3.2 Operation

We will denote with *operation* a *top-level* function of *classic* PLaSM. The following definition is the standard one given in PLaSM grammar.

DEF *op* [({{*par*}, :: *IsType* };)]* = *body*
[WHERE *local_env* END]

where

par ::= *par-name*
IsType ::= *type-expression* | *class-name*
body ::= *plasm-expression*
local_env ::= { *fun* = *body* }'

This is a simple example of PLaSM operation:

```
DEF op (par_1,...,par_n::IsType_1) (par_{n+1},...,par_{n+m}::IsType_2) ··· = expr
WHERE
  local_1 = expr_1,
  local_2 = expr_2,
  ......
  local_n = expr_n
END;
```

3.3 Class

An OO-PLaSM class was defined as the image set of a generating function of class type. A *class* generating function is like a normal PLaSM function, but produces labeled values (called class *objects*). Its EBNF syntax is as follows:

> CLASS *name* [({{*par*}, :: *IsType* }$^{;}$)]* [ISA *superclass*] = *body*
> [WHERE *local_env* END]

where

name	::=	*plasm-name*
superclass	::=	*IsType*
par	::=	*par-name*
IsType	::=	*type-expression* | *class-name*
body	::=	*plasm-expression*
local_env	::=	{ *fun* = *body* },

A new OO-PLaSM class can hence be defined in this way:

```
CLASS newclass (par_1,...,par_n::IsType_1) (par_{n+1},...,par_{n+m}::IsType_2) ···
  ISA superclass = expr
WHERE
  local_1 = expr_1,
  local_2 = expr_2,
  ......
  local_n = expr_n
END;
```

3.4 Feature

A *feature* is defined as an internal function over a class. By definition, a feature can be applied only to values of the class over which it was defined. Also, by its closure property, a feature function will only return values of the same class.

> FEATURE *name* [({{*par*}, :: *IsType* }$^{;}$)]* ON *class* = *body*
> [WHERE *local_env* END]

where

name	::=	*plasm-name*
class	::=	*class-name*
par	::=	*par-name*
IsType	::=	*type-expression* | *class-name*
body	::=	*plasm-expression*
local_env	::=	{ *fun* = *body* },

An example of *feature* may be:

```
FEATURE class_feature (params::IsType) ON class = expr
WHERE
  local_1 = expr_1,
  local_2 = expr_2,
  ......
  local_n = expr_n
END;
```

3.5 Control

A *control* is an OO-PLaSM function defined over a class. It can be applied only to values of the class, but unlike a feature it returns values of different classes.

```
CONTROL name :: IsType [ ( {{par}, :: IsType }; ) ]* ON class
    = body
[ WHERE local_env END ]
```

where

name	::=	*plasm-name*
class	::=	*class-name*
par	::=	*par-name*
IsType	::=	*type-expression* | *class-name*
body	::=	*plasm-expression*
local_env	::=	{ *fun* = *body* },

For example, a parameterized `control` function is defined as follows, where *IsAnyType* is the type of the value returned by the application of *mycontrol* function on some object of *myclass* class.

```
CONTROL mycontrol::IsAnyType (params::IsType) ON class = expr
WHERE
  local_1 = expr_1,
  local_2 = expr_2,
  ......
  local_n = expr_n
END;
```

4 Semantics problems

The language extension described in this paper seems to match quite well the requirements posed by knowledge-based mechanical design applications to an object-oriented design environment. Anyway, some semantics problems remain open in this approach, whereas conversely the approach itself looks nice (at least to the authors) from some abstract viewpoint. The main open problems are briefly described in the following.

Class closure versus its features Generally speaking, the results of application of a feature f are only weakly closed on the class C the feature f belongs to. As a matter of fact, if a is an element of the C class, it is not usually verified that $f(a)$ can be generated by the C constructor (i.e. the C generating expression). In this case we assume that the result $f(a)$ returned by the function application, is labeled with the same *tag* of a, so that both a and $f(a)$ are forced to belong to the same class.

Feature composition validity Features are unary operators whose resulting values are closed (as specified above) in a class. In this sense they form a non-commutative group with respect to the composition of functions.

Let C be a class and F be the set of its features (both locally defined and inherited from the superclasses). Generally speaking, to apply any feature $f \in G(F)$, where $G(F)$ is the feature group, to any object $c \in C$ is a nonsense operation. Even if the single operation is syntactically and geometrically valid, the result of a generic composition of operations could be semantically wrong or nonsense for the user.

Variant closure versus its features Formally, a variant is a subset of the superclass which contains its generating expression. This subset coincides with the union of the image sets of the functions belonging to the feature group redefined by the variant. Even in this case, not all the functions belonging to the feature group of the variant can be applied to the objects of the variant without generating nonsense values.

Validity control for features Generally speaking, the application of compound features (i.e. the ones obtained by composition of others), even if always allowed, must be approached by the user with particular caution. In particular, it could be very useful to impose some mandatory constraint concerning the application of features. Such application could be always followed by some special validity control (named `ISVALID?` for example) implementing control criteria over the class and returning either the input value or an exception such as `INVALID_VALUE`. Anyway, as it is well known from the solid modeling theory, such kind of general validity checks, in this case also capturing some design intent, are very difficult to implement.

5 Implementation

5.1 Local environments visibility determination

The goal is to implement full local environments visibility within a class hierarchy (*features* and *controls* included). In this way the language is more strictly bounded to an $Object-Oriented$ paradigm, as many other OO imperative languages. Ideally we can consider $features$ and $controls$ like the public methods of classes, but in this case the class interface is defined in a dynamic way (it could be everytime enriched by other methods in the global environment, without having to redefine anything).

A first problem is to manage formal parameters and their visbility. We can consider different strategies of implementation, each of which involves advantages and disadvantages.

Let's consider the following OO-PLaSM example, where two classes are defined and the second is derived from the first:

```
CLASS base_obj (par1,par2::IsReal) =
    f(par1,local1,local2)
  WHERE
    local1= g1(par2),
    local2= g2
  END;

CLASS der_obj1 (par3::IsReal) ISA base_obj =
    base_obj:<local3,par2>
  WHERE
    local3= h1(par2)
  END;
```

Note that in this example the second class uses directly the symbol par_2 defined in its superclass. In this case we couldn't decide for a single OO-PLaSM form to obtain a der_obj_1 class object without defining a complete *scope* rule for par_2 parameter. In particular two are the possible strategies:

1. der_obj_1:<15>;

or

2. der_obj_1:<15>:<ANY_VALUE,8>;

Analyzing the two above expressions we have undefined values in the first case (that is not all the parameters are assigned during valuation) and information redundancy together with *generic* values in the second (that is some parameter is assigned more than once and ANY_VALUE expressions are used to solve them). A first approach to the problem has outlined three different implementation strategies, showed below starting from the one more quickly affordable and going on till to the more complex and evolved one.

All the strategies are based on the following derivation scheme: a derived class C_d of a superclass C is a partial closure of C, that is it could be C with new added parameters (and new added relations) indipendent from the old ones (defined in the superclass), or is a specialized version of C (that is some of C parameters are bounded by particular relations introduced in C_d, or by new C_d parameters). In the first case we say that C_d is an *enriched* version of C, while in the secon is a C *specialization*.

Private local environment Most immediate solution is to realize a mechanism in which all local symbols are considered *private*, that is they can be used only by class and *features/controls* which defined them. So the only visibility allowed is limited to the *controls/features* defined in the class itself.

In this way we can bypass all the problems related to information redundancy, undefined values and open functions. This kind of implementation implies a more verbose style for the language, withuot however penalizing its expression power.

In the proposed example, the definition of the derived class der_obj_1 would raise an error, because the used par_2 symbol is actually defined in its superclass so it doesn't belong to its scope. In this case neither the version 1. nor the 2. would be valid.

One advantage of this solutions is that it can be implemented apporting few changes to te existent `PLaSM` parser and introducing few primitives for managing the new `CLASS`, `FEATURE` and `CONTROL` keywords (see point 1 in section 5.2).

Weakly protected local environment In a further step, we can export only locally defined symbols and hold in class scope formal parameters. Following the analogy with imperative languages, we could consider a *private* internal state for parameters and a *protected* internal state for local symbols.

Note that only those symbols which don't depend on formal parameters could be exported towards derived classes, without producing open functions. We could define a set containing all the symbols with a protected internal state and one with all other symbols, formal parameters included, which have a private internal state. Misusing λ-calculus terminology, we could name symbols in the first set as $free - symbols$. Only these will be visible in derived classes (*features* and *controls* included), while all the others will be accessible only for class, and related *features*/*controls*, which defined them.

Even this time, the previous example would raise an error during definition, because der_obj_1 class cannot use par_2, which is defined in $base\_obj$. Neither the expression 1. nor the 2. are still valid. However, let's consider the following example:

```
CLASS obj (par1::IsReal) =
    expr
  WHERE
    local1 = f(par1),
    local2 = local_expr2
  END;
```

Now only $local_2$ would be exported to derived classes, while $local_1$ (that depends on par_1) can be used only in the class which defined it (and in its *features*/*controls*).

Implementing such an approach is quite expensive for the functional interpreter, both in a realization point of view (the parser needs more changes), and in a execution point of view. For the last, the interpreter should everytime collect a table containing the local symbol list, dividing them in $free$, so exportable, symbols and not-$free$ symbols. However, with this approach ANY_VALUE usage and information redundancy are still avoided, and the language itself would allow a partial symbol visibility within a class hierarchy, in particular for all those symbols that are classified as $free$.

Protected local environment A more evolved solution could assign both to parameters and to local symbols a *protected* internal state, threating all the symbols defined within a class in the same way. In this way all of them would be visible for all the *features* and *controls* related to the class, and also for all the class belonging to the hierarchy, including their *features* and *controls*.

This time the proposed example would be considered valid by the interpreter, because a derived class like der_obj_1 can use the parameter par_2 defined in its superclass. However,

in order to obtain a good generation of object belonging to der_obj_1 class, further mechanisms for holding particular kinds of values have to be introduced, to avoid information redundancy.

The simplest solution solves this redundancy by introducing undefined values, named ANY_VALUE (as seen in expression 2.), that stand for a generic expression whose real value is not requested. In the proposed example, der_obj_1 closure defines a function of superior class *base_obj* (that is der_obj_1 is an enrichment of *base_obj*), which will be instantiated applying a cuople of values. However, in the derived class a new bound among some of the superclass parameters is introduced, and the same parameters are so partially assigned. To satisfy that signature we have to give a couple of values to the resulting expression, but only the second is meaningful.

Such a problem raises all the times that a derived class uses and assigns only partially a superclass parameter. In our case *base_obj* has actually only one parameter, that is a couple of values, but only one of those values are assigned in the derived class (that, in this way, uses only half of the superclass parameter). This situation causes redundancy to be avoided. Undefined values usage delegates to the programmer all the checks to avoid such situations, giving however a secure tool for holding more critical ones.

Environment with control agents In a more complex system a set of automatic controls (active or passive) could be implemented, such that they could alert the programmer if some violation is found in the user environment so long defined. In particular we can think to passive controls as a set of programming costraints to be satisfied in all the definitions introduced by the user in `OO-PLaSM` environment (a sort of programming rules declared in the environment itself). These constrains could be used in association with a set of controls based on active agents, which everytime monitor all the evolution of the environment and detect each possible corruption of it. Such a system, however, needs a more deep analisys of constraint satisfaction systems and a remarkable contribution from artificial intelligence. Its implementaton is therefore more faraway to be realized than the others showed above.

5.2 Class type recognition tactics

The correct semantic associated to the new primitives `CLASS`, `FEATURE` and `CONTROL` is based on implicit recognition of the type associated to specific functional classes.

The class type is created using a tag feature applied to the class itself, during the definition process and it's set to be equal to the class name.

For example:

```
CLASS obj1 (parameters) = expr ;
  WHERE
    local_env
  END;
```

The type associated with obj_1 is exactly obj_1 (that is obj_1 is a function of class/type obj_1). Even in this case there are available different implementation strategies.

Textual rewriting approach In a first implementation effort focused on rapid (although not final) `PLaSM` language extension toward `OO-PLaSM`, it can be useful to delay the scope definition of local *protected* environments, solving the type determination problem through the introduction of few new primitives.

Using this approach, most operations are realized using a textual simplification method. Operations are rewritten in terms of simpler ones, using the following rules:

i. It's introduced a new expression `OO-PLaSM` using the terminal symbol `TAG` and applied to a list of three elements where the first contains the type of associated function (`CLASS`, `FEATURE` or `CONTROL`), the second contains the name and the third the *type* associated with the class.

   So, a `TAG` primitive can be expressed using the following syntax rules:

   `TAG:<`*class*|*feature*|*control*`,`*function name*`,`*class type*`>;`

   The new primitive updates the internal state of the `OO-PLaSM` function so defined, which internal representation is augmented in a transparent way respect to the previous implementation `PLaSM`.

ii. Moreover, it is introduced a new primitive called `ISTYPE` that can be applied to an object of a class, analyzing the internal state of the object (created and modified using the `TAG` primitive) and returning the associated type information:

   `ISTYPE:`*class type* $\Rightarrow$ `IsClassType;`

   This primitive is used to expand expressions like `FEATURE` and `CONTROL` into smaller parts.

iii. An expression of type `CLASS` will be expanded as follows:

```
CLASS der_obj (params) ISA base_obj = expr ;
          ⇓
DEF der_obj (params) = expr;
TAG:<'CLASS',der_obj,base_obj>;
```

   In this case, the function `TAG` gets a list of three elements where the first is the string *CLASS*, the second is the class name *der_obj* (that also identifies the type) and the parent class type (if it doesn't exist a null value will be imposed).

iv. A `FEATURE` expression will be expanded as follows:

```
FEATURE obj_feat (params) ON obj = expr;
          ⇓
DEF obj_feat (params)(this::ISTYPE:obj) = expr;
TAG:<'FEATURE',obj_feat,obj>;
```

The `TAG` expression gets three parameters: the *FEATURE* string, the function name and the class type that defines the applicability scope.

Also note the usage of the primitive `ISTYPE` for the generation of the type assigned to the last argument of the `DEF` primitive created (that is the functional class type on which the feature can be applied).

v. A `CONTROL` expression will be expanded as follows:

```
CONTROL obj_cont::IsAnyType (params) ON obj = expr;
          ⇓
DEF obj_cont (params)(this::ISTYPE:obj) = expr;
TAG:<'CONTROL',obj_cont,obj>;
```

where the triple assigned to `TAG` primitive is composed by the *CONTROL* string, the function name and the class type on which it's defined.

Such a structure must be supported by modifying the `DEF` primitive, that now should generate a special *wrapper* to control the type of objects on which the function is applied.

Introspection approach Another way is to use the local class environments.

In this case it would be defined a local simbol, *standard* and with the same name for every class, containing the type informations associated for that class.

So the type can be obtained simply analyzing the value of the local symbol exposed by the particular class, an operation that be done either by assigning a *public* (the only exception to scenarios introduced in the previous section) to the symbol itself or by introducing controls (*standard* and with the same name for every class) that read the value and return it out.

Surely, this strategy is more general but and provides a complete definition and implementation of local environments and associated visibility scopes.

6 Examples

In this section OO-`PLaSM`will be illustrated through some simple example. The idea is to give some hints about the proposed sintax extension and to show the more interesting characteristics.

6.1 Parameterized cylinder in *classic* `PLaSM`

In Figure 6.1 and in Appendix A there are some code fragments in `PLaSM` needed to generate the parameterized cylinder shown in Figure **??**. Three functions are used to the scope, the last generating the object `cilindroBrava1600` parameterized on the external symbol `numero_cilindri`.

```
DEF numero_cilindri = 4;

DEF cilindro (r,h,spessore::IsReal) =
  (Ring:<r, r+spessore> * QUOTE:<h>) TOP
  (Ring:<0, r+spessore> * QUOTE:<spessore>);

DEF cilindroBrava (r,volume::IsReal) = cilindro:< r,h,r/5 >
WHERE
  h = volume / (PI * r * r)
END;

DEF cilindroBrava1600 = cilindroBrava:< 3.1, volume >
WHERE
  volume = 1600 / numero_cilindri
END;

cilindroBrava1600;
```

Figure 2: Classic `PLaSM` code fragment that generates the object `cilindroBrava1600` that depends on the cylinder number and piston displacement.

6.2 Parameterized cylinder with specialized features

In the following it will be shown the generic `OO-PLaSM` implementation of the classes hierarchy as it can be seen in Figure 5. First it will be given some definition equivalent to the figure content, then these domain descriptions will be used to instantiate a specific object belonging to the class `cilindroBrava1600`.

Classes, features and controls hierarchy The specialization hierarchy shown in Figure 5 defines a *starting class* `cylinder` with objects parameterized on ray (`r`), height (`h`) and `thickness`.

The objects resulting from the application of the three function *CLASS* with actual parameters will be *tagged* with the labels "`cilindro`", "`cilindroBrava`" and "`cilindroBrava1600`" respectively.

At line `1` (Figura 3) the `cilindro` function is defined with three formal parameters `r,h,spessore`. If not redefined, these parameters are visible even inside subclasses of `cilindro`, as in the cases shown in section 5.1.

The definition shown in line `2` establish the behaaviour of the function `cilindroBrava`: when it's applied to a list of two real numbers, it generates a function equivalent to (a specialization of) `cilindro` and so it can be used by the same way. The values produced by the function will be generated by a different generating expression, specifically the one shown in line `3`. In *classic* `PLaSM` it's body should appear as

```
cilindro~[s1,s2,s1/K:5]
```

```
1    CLASS cilindro (r,h,spessore::IsReal) = expr;

2    CLASS cilindroBrava (volume,r::IsReal) ISA cilindro =
3      cilindro:< r,h,r/5 >;
4      WHERE
5        h = volume / (PI * r * r)
6      END;

7    CLASS cilindroBrava1600 ISA cilindroBrava =
8      cilindroBrava:<volume,3.5>
9      WHERE
10       volume = 1600 / numero_cilindri,
11     END;

12   FEATURE valvola (r::Isreal)(alpha::Isreal)
13     ON cilindroBrava1600 = expr;

14   CONTROL volume::IsReal ON cilindro = expr
```

Figure 3: Piece of code with classes, features and domain controls definitions

In OO-PLaSM it is possible to use λ style thanks to symbols defined in superclasses. (as in case 3, section 5.1).

It should be noted that although the class (function) `cilindro` contains (generates) geometric objects characterized by every possible rays, heights and thicknesses, the class (function) `cilindroBrava` contains (generates) only objects with thickness set to the fifth part of the ray and with height linked to ray, using a function defined on the specified volume.

In the `cilindro:< r,h,r/5 >` application shown in line 3 the symbol `r` is redefined and inserted into the class signature of `cilindroBrava`. In a scenario where every symbol is defined as *protected*, the symbol `r` should be the one defined inside the nearest superclass (not in the `cilindroBrava` superclass signature). Otherwise the symbol `h` is the one locally defined at line 5, that redefines (makes not visible) those defined at line 1.

At line 7 it's defined the class `cilindroBrava1600`, specialization of `cilindroBrava`. The generating expression

```
cilindroBrava:<volume,3.5>;
```

uses the local symbol `volume`.

Thinking for a while that every symbol is visible inside the hierarchy (with the symbol `r` of `cilindroBrava` directly inherited from the class `cilindro` and so not specified into the signature), the function produced by the evaluation of `cilindroBrava:volume` should be applied on undefined parameters of type 8, `ANY_VALUE` and `ANY_VALUE`, as it can be seen at line 8 of the following piece of code:

```
2    CLASS cilindroBrava (volume::IsReal) ISA cilindro =
3      cilindro:< r,h,r/5 >;
```

```
15    DEF numero_cilindri = 4;
16    DEF my_valv = valvola:expr;
17    DEF valv120 = valvola:120;

18    DEF cilindro_3_valvole =
19      ( my_valv:0
20      ~ valvola:140:(PI/3)
21      ~ valv120:(2 * PI/3) ):cilindroBrava1600

23    volume:cilindro_3_valvole;
```

Figure 4: The script uses the control `volume` on an object of class `cilindroBrava1600` modified by the application of three features `valvola`.

```
 4      WHERE
 5        h = volume / (PI * r * r)
 6      END;

 7    CLASS cilindroBrava1600 ISA cilindroBrava =
 8      cilindroBrava:volume:< 3.5, ANY_VALUE, ANY_VALUE >
 9      WHERE
10        volume = 1600 / numero_cilindri,
11      END;
```

The application on the triple could be used to generate a `cilindro` where only the first parameter has meaning, but the last two can get every value. Because the symbol `numero_cilindri` used at line 10 isn't defined neither locally nor in some superclass, it must be an external symbol. This implies that every instance of `cilindroBrava1600` will depends on it.

Features and object instantiation At line 15 it's assigned a value to the symbol `numero_cilindri`, that formally it's a global 0-ary function or a *constant*. At lines 16 and 17 two different partial functions are generated and named (respectively) `my_valv` and valv120. If these are applied on a real number, they will generate two instances of the parametric feature `valvola`, defined at line 12 on the class `cilindroBrava1600`.

It should be noted the mechanism of using the functions `FEATURE` at lines 19, 20 and 21. By definition, the *features* are *functions* and so they can be composed between themself and with the function `CLASS` on which they're defined or with functions `CLASS` that are specializations.

At line 21 the generating form `cilindroBrava1600` is evaluated and the generated value is passed as input to the feature `valv120:(2 * PI/3)`. It's output is passed to the feature `valvola:140:(PI/3)`, whose output is used by by the feature `my_valv:0`. The global result of the computation will be stored into the constant `cilindro_3_valvole`.

At line 23 the domain control named `volume` is applied on the object `cilindro_3_valvole` belonging to the class `cilindroBrava1600`, on which the control is defined.

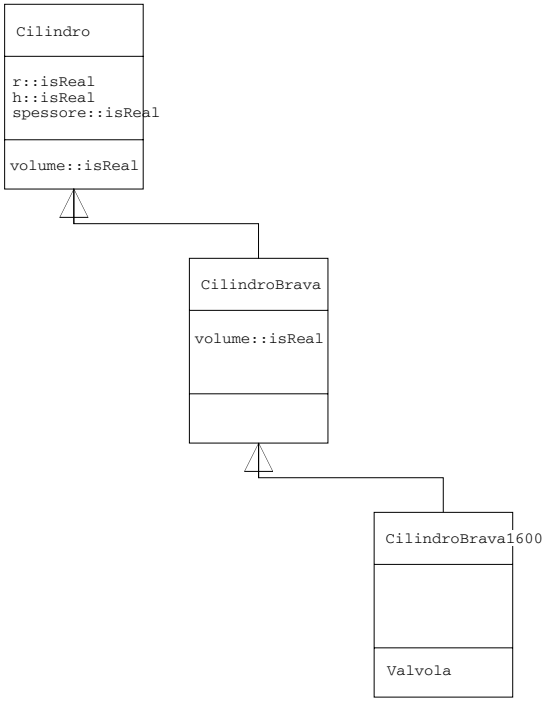

Figure 5: UML diagram of the class hierarchy considered in the example

6.3 Representation by diagram of the classes, features and controls hierarchy

The specialization relationship between classes, together with *features* and *controls*, can be visualized as a UML (Unified Modeling Language) class diagram, as shown in Figure 5 about the example of Section 6.2. According to the UML rules, every class is represented by a rectangle containing three areas. In case of OO-`PLaSM`classes, the three areas respectively corresponds to:

- the function name,
- the formal parameters of the generating function and
- the features and controls defined on the class.

Features and controls differs by the type expression attached to the result value in controls.

Two classes included into the *specialization* relationship (and into the inverse *generalization* relationship) are linked through an arrow starting from the subclass and directed toward the superclass. So every class can inherit methods, controls and generating expressions defined along the path from the starting class (root) of the connected subgraph (tree) to which the class belongs.

```
DEF domain1D (a::isreal)(n::IsInt) = (QUOTE ~ #:n):(a/n) ;

DEF Interval (x1,x2::IsReal)(n::IsIntPos) =
  (T:1:x1 ~ domain1D:(x2 - x1)):n ;

DEF Domain2D (x1,x2::IsReal)(y1,y2::IsReal)(n,m::IsIntPos) =
  Interval:<x1,x2>:n * Interval:<y1,y2>:m ;

DEF Ring (r1,r2::IsReal) =
  ( MAP:[s2 * cos ~ s1, s2 * sin ~ s1]
  ~ Domain2D:<0,2*PI>:<r1,r2> ):  <32,1> ;
```

Figure 6: Geometric toolbox used to generate 2D circles and rings.

APPENDIX

A Geometric toolbox

In Figure 6 are reported `PLaSM` code fragments needed to generate polyhedral approximation of circles and rings 2D. Both are generated by the function `Ring` that applies a specific map to a specific 2D domain decomposition generated by the Cartesian product of 1-dimensional domains.