13. Solid modeling of architectural design with PLASM language

Alberto Paoluzzi, Claudio Sansoni°*

(*) Dip. di Informatica e Sistemistica Università "La Sapienza"
V. Buonarroti 12, 00185 Roma, Italy
(**) Dip. di Progettazione Arch. e Urb. Università "La Sapienza"
V. Gramsci 53, 00197 Roma, Italy

PLASM (Programming LAnguage for Solid Modeling) is a prototype, high level, user oriented, functional design language currently being developed at the University of Rome "La Sapienza". A PLASM "program" is the symbolic definition of a complex of variational polyhedra depending on some unbound variable, and therefore allows for the description of a whole set of geometric solutions to a design problem. In our view the language should be used, possibly with the assistance of a graphical user interface, both in the first steps of the design process as well in the detailed design. In the paper the guide-lines are shown of the preliminary definition of the syntax of the language. The paper also contains the definition of some new and very powerful solid operators.

Introduction

PLASM is an experimental design language which allows to define and evaluate the expressions of a special "calculus on polyhedra". It is a pure functional language, where expressions contain operators which map some complexes of polyhedra into other complexes of polyhedra . The language is inspired to the style introduced by the Backus's languages FP (Backus 1978, Williams 1982) and FL (Backus et al. 1989) for programming at function level. PLASM allows to symbolically describe parametric architectural shapes, but the language can also be used for other applications, e.g. to describe the robot, the motion and the environment in the off-line programming of robots (Bernardini et al. 1991).

As the objects and expressions are parametric, the language can generate complex "variational" objects, as well perform the automatic mapping between design representations at different detail levels. For example, PLASM can be used to define the 2D parametric layout of a building, as well to automatically generate a 3D solid model. In particular, the language introduces some powerful new operators for *product*, *power*, *offset*, etc., of complexes of polyhedra. A key concept is the introduction of an extension of the *traversal* operation (typical of PHIGS-like graphical systems), which allows to fully describe structured solid objects without specifying all low level geometry details. We note also that the PLASM operators are *dimension independent*, since they are defined for polyhedra of generic dimension (and not only for 2D or 3D objects). The operators are *abstract*, as their evaluation does not depend from the representation of arguments.

PLASM is planned to be used both by application programmers, which can define new high level operators by writing language *scripts*, and by designers, which can evaluate predefined functions in library packages or interactively instanciate language constructs with the aid of a user-oriented graphical interface, which must allow for bidirectional echoing between a language window and a graphical window.

In an imperative language a program is a sequence of instructions which implement a mapping between input and output data. To program design methods and algorithms in an imperative language may be, in the general case, very hard to do, so that problem-oriented languages are needed for design. Various languages (Steele 1975; Borning 1981, 1986; Leler 1985, 1988) were developed to solve design problems in engineering areas quite far from architecture (e.g. in the design of mechanisms or electrical networks), and work by solving a set of simultaneous constraints. In such languages a "program" is simply a description of functionality and a sequence of constraints (expressed algebraically).

PLASM's aim is to give support for the description of geometrical solutions to shape design problems (in general by producing "solid" models) and to allow for automatic transformation between design descriptions at different detail levels. A "program" is in this case a parametric shape description, given as a set of definitions (with or without some unbound variable). When the program is applied to specific values of parameters, then a geometric model of the designed shape is produced. A natural choice was to adopt an applicative style of programming, and obtain from the right side of each definition the result of the evaluation of an expression of a special "calculus" over polyhedra.

The development of PLASM is one of the research lines of the CAD group at DIS. After having developed the polyhedral solid modeler MINERVA (Paoluzzi et al. 1989) we are currently working to methods for the representation and manipulation of polyhedra of generic dimension (Ferrucci et al. 1991). Such an approach allows in fact to treat in an unified manner several geometric problems (Bernardini et al.).

The paper is organized as follows. First, some main approaches to the use of computers in design are recalled and compared with the programming approach proposed here; then some guide-lines about the language syntax and operational semantics are given, showing the applicative style of the language and some low-level predefined functions. A strong emphasis should be given to the fact that most of PLASM functions are reducible to the evaluation of the STRUCT function, i.e. to the traversal of a structure. Some new high level operators on polyhedra are then introduced and some simple examples are discussed, with the aim of giving some flavour about the expressive power of the language. In the conclusion section some critical issues and open problems are outlined.

Geometric modeling and building design

Current systems vs programming approach

The current systems used in architectural design can be classified into two broad categories:

• 2D drafting systems: the main aid they give is the possibility of hierarchically organizing the design into groups and levels. Such systems are the most diffuse in the professional ateliers because they
do not essentially change the classical practice of the architect's work;

• 3D editing and rendering systems: are used to perform 3D projections, produce hidden surface removed views and, more recently, realistic rendering of the design volumes and of the external envelopes, as well to realize electronic walks-through. 3D systems are quite weak from the modeling point of view, as they need a lot of data and of human work in order to define the model to be rendered. So they are used mainly in the final stage of the design process, as any change to the model may become very expensive and hard to do.

The best current systems are highly interactive, and use data structures which are transparent to the user. In our approach we want conversely offer a programming environment to the architect, allowing to symbolically define the design shape, as well to trace any (geometrical) decision which influenced the generation of the shape. The use of a special purpose language makes the user free from emulating the drafting process; furthermore, this may stimulate the architect creativity, as it allows for quickly exploring several design solutions.

The standard top-down approach

A building can be seen as a container of spaces, where the internal partitions determine the configuration of inner spaces, while the external envelope determines the volume as a whole. The designer needs hence to have contemporary control of both empty and full subsets of spaces.

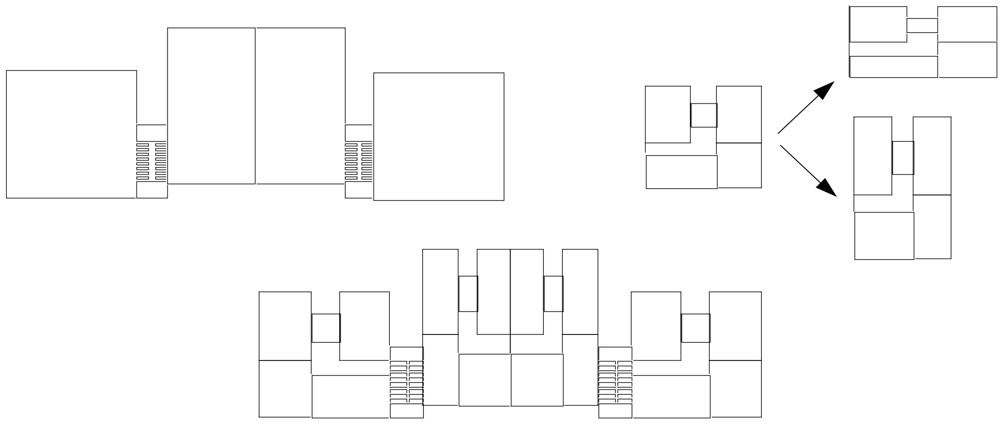

Figure 1. Design development by top-down refinement with an extensive use of affine transformations.

In the first design stages it is common to choose a top-down approach and to work by successive refinements. For example, when the object as a whole is already defined as a set of possibly interpenetrating volumes, and external constraints are satisfied, it is possible to define the internal subdivisions by hierarchically partitioning the parent volumes and by using an adimensional geometry approach (see Figure 1). In this case, the designer attention is dedicated to the shape as a whole: the need is evident for tools able to manipulate the shape by affine transformations and set operations, without explicitly requiring the complete specification of the geometry of the internal partitions. March and Steadman in (March et al. 1971) have shown since the early seventies how the affine transformations are often applied to the architectural design. Such approach can be easily modeled by allowing the design language to contain a structure definition mechanism analogous to that given by the PHIGS graphics standard (PHIGS 1987).

Variational and rule-based approach

The use of rules in architecture is very ancient; the rules which control the ratios between the parts of temples as a function of the column diameter are an eloquent example (Chitham 1985). In the same way, in the modern architecture age, Le Corbusier used his "Modulor" (Le Corbusier 1965) and based shape ratios on the golden section and Fibonacci's series in order to obtain a harmonic control of proportions.

A parametric representation, also denoted as "primitive instancing" in (Requicha 1980), is defined as a procedure which generates a shape instance depending on the values of a predefined set of parameters. Often it is necessary to modify a previously defined object, where the change must affect (recursively) all the adjacent objects. A solution to this problem is given by variational geometry (Gossard 1988), originating in the mechanical design domain, by assuming geometrical constraints between object parts. The explicit definition of the shape, usual in current graphical systems, needs the enumeration of the (either absolute or relative) position of objects vertices. Conversely, the variational approach is implicit, as it contains constraints and not (only) coordinates. So, a variational definition makes the objects have some explicit degrees of freedom. This representation is a very powerful design aid, as a parametric definition allows to easily explore the consequences on the whole object that are induced by a change in a part.

"Design by constraint" is an important rule-based design methodology (Barford 1987) where whole classes of design solutions are implicitly defined. The front view and the structural grid of a building are two examples of shapes where the final configuration can be defined with dimensional rules. It is possible to introduce in a variational PLASM definition many "rules of composition" both as conditional constructs IF:<cond,expr1,expr2> and as relational or logical or type constraints that the objects or the parameters must satisfy, and which express positional or dimensional relationships between the parts of a structured object.

In the following we use the term "variational" object to denote an expression where either affine transformations or component objects or both may depend on some parameters, and where explicit constraints between parameters may be given and, in this case, are satisfied by any object instance. As will be shown in the paper, the symbolic representation contained in a PLASM script can be considered a variational machinery any time it contains a set of unbound parameters (at left side of definitions).

The PLASM approach

PLASM (Programming LAnguage for Solid Modeling) is an experimental language for the symbolic description of parametric polyhedral complexes. The language domain is restricted to linear polyhedra, as curved objects can always be approximated by piecewise-linear polyhedra. Several operators are specialized for the architectural design domain. Using predefined operators the user can develop his own higher-level architectural operators. For example, it is possible to implement several compositions of symmetry and series, as well as operators which result, when applied to specific data and evaluated, in specific parts of the building fabric (like the staircases or the structural grid).

The geometry of a PLASM object is described by instancing predefined operators, either contained in external libraries or user-defined. A complex object is defined by building a new definition. This can be done in several ways, e.g.: (a) by specifying a 1D complex (which is a graph containing vertices and edges); (b) by specifying a 2D parametric complex (which is the plane partition induced by a 1D complex); (c) by evaluating the boundary of the cells of the 3D complex generated by the extrusion of a 2D complex; and (d) by defining a structure containing operators, elementary objects and affine transformations of substructures.

With PLASM it is possible: to work in a traditional way by developing plan and section drawings (and then to automatically generate a 3D model of the building, by using some specialized operators, as will be described in the paper); to directly assembly 3D solid components; to integrate the two techniques depending on the problem and on the chosen design style.

In our view, PLASM may constitute the kernel of specialized macro-languages for different areas of architectural design (housing, hospitals, architecture of interiors, etc.), as well it might be used to produce personalized packages where recurrent choices and stylemes of the designer are registered. We notice that such an approach is coherent with the tendency to define special purpose languages in areas where application programs were used before. The reason is that such languages make the user able to build specific applications in a much more powerful and flexible way. Examples of this trend are the Postscript language for the description of high quality text pages, TeX (Knuth 84) and LaTeX (Lamport 1986) for electronic typesetting, Mathematica (Wolfram 88) for symbolic computation, Renderman (Pixar 89) for realistic rendering, and many others.

Summing up, we believe that the programming approach with PLASM language may give a very flexible and powerful aid to the design development. In particular, both a bottom-up approach, based on component assembly, and a top-down approach, based on model refinement, as well any kind of mixed strategies of design development could be freely chosen by the designer.

Syntactical notes

A PLASM *expression* may contain *operators*, *sequences*, *atoms* and parenthesis (to make explicit or modify the order of evaluation of the expression). A PLASM *script* is a sequence of *definitions* separated by commas. A definition may concern expressions or operators. Both contain a left and a right part separated by a symbol "=". The right part of a definition always contains an expression. A definition may also contain an optional script, which has the definition as scope.

Atomic objects belong to simple types. Simple types are the Booleans BOOL, the reals REAL, the integers INT, the affine transformations TRANSF and the polyhedra POL. Sequences are ordered multisets of PLASM objects or expressions, enclosed within pair of angle delimiters "<" and ">". Structures are applications of the STRUCT operator, defined on ordered sequences of transformations and polyhedra:

$$\text{STRUCT: (TRANSF} \cup \text{POL)}^* \rightarrow \text{POL.}$$

The types TRANSF and POL are partitioned in subtypes of objects with a given dimension, denoted by a positive integer. I.e. transformations and polyhedra may be 0,1,2,3,... n-dimensional. The 0D cells of a polyhedron in ^{n} are nD points, which are represented as sequences of n reals. A nD affine transformation in TRANSF^n is a (n+1)(n+1) non singular matrix, internally represented (when the dimensions are given) as a sequence of reals.

An operator is a mapping from the product set of parameter domains to the output domain. The language contains a special form for definition of operators, consisting of: (a) the reserved word Def; (b) the operator identifier; (c) an optional expression, usually a sequence of integers, which gives a further specification to the operator, enclosed within single or double braces; (d) the optional sequence of unbound parameters, enclosed within parenthesis; (e) the reserved symbol "="; (f) a PLASM expression used to evaluate the operator; (g) an optional script, enclosed between the reserved words Where and End**;** (h) the optional sequence of default values for parameters, between the reserved words Default and End . The single or double braces around the optional parameters specify the reduction rule that must be used when the operator is applied with more than one optional parameter.

Application rules and combining forms

An operator (a function) is normally invoked in prefix form by applying its identifier to an actual parameter. Application, denoted as expr_1:expr_2, apply the value of expr_1 (a function) to the value of expr_2. Infix form is allowed for binary operators. Actual parameters must usually coincide in order and type with formal parameters given in the right side of the definition of the operator. Some parameters may be not given in the actual parameter sequence (but giving the corresponding comma pairs). In this case the default values of such parameters are used in the evaluation process. Expressions or names of expressions within the current scope can be used in the application.

When a *binary* operator is applied to more than two arguments it is recursively evaluated using the follow equivalence:

op:<x1,..., xm, xn> = op:<op:<x1,..., xm>, xn>

Unary operators (defined as applicable to atoms), when applied to a sequence of arguments, evaluate to the sequence of applications to the elements of the sequence:

op:<x1,..., xn> = <op:x1,..., op:xn>

According to FL, two special combining form are used (a) to apply a sequence of functions to the same argument and (b) to apply a function to every element of a sequence argument:

[op1, op2,..., opn]:x = <op1:x, op2:x,..., opn:x>
a:op:<x1,x2,...,xn> = <op:x1,op:x2,...,op:xn>

The a:op function (apply-to-all) evaluates to error when applied to an atom.

Some operators have been defined by using an optional sequence of integers enclosed within single or double "{" and "}" parenthesis, which usually specify the subset of coordinates affected by the operator. In this case the evaluation process can be regarded as follows:

op{i1,...,in} is equivalent to op{i1}:op{i2,...,in}

if op has ben defined with Def op{{i}}; else if op has been defined with Def op{i}, then the following reduction rule apply:

op{i1,...,in} is equivalent to [op{i1},op{i2},...,op{in}]

The first reduction rule applies, e.g., to the transformation functions T,R,S,H or to the centre function defined in the next sections; the second applies, e.g., to the functions min, max, med. When the definition was Def op{i1,...,in}(y) = Body, the reduction equivalencies expressed in Body are applied. Notice that in the application of a sequence of optional parameters, only the single brace pair is used. In order to make the PLASM expressions easier to read, binary composition of functions is allowed. An infix symbol "∞" must be provided in this case:

op2 ∞ op1:x = op2:(op1:x)

Transformations

The elementary affine transformations, i.e. translation "T", scaling "S", rotation "R" and shearing "H", could be seen as maps (REAL)* • POL^n Δ POL^n, where POL^n denotes the set of polyhedra of dimension n. In PLASM they are instead considered as maps from a space of real parameters to the space of squared invertible matrices. E.g., the translation in the direction x_n, which depends on one real parameter¸ is represented by a squared matrix of unknown dimensions (which are fixed only at traversal time). Such object is syntactically expressed as T{n}:a, where the positive integer n denotes the n-th coordinate and the real a is the transformation parameter. Transformations are applied to polyhedra by using structures, described in the next section. E.g., the translation T{n}:a is applied to the polyhedron P by evaluating the expression STRUCT:<T{n}:a,P>. The effect of the evaluation of such expression is that of summing the real number a to the n-th coordinate of P vertices. Predefined operations on transformations are the binary composition "∞", the unary inverse and the standard identity ID:x = x (which applies to PLASM expression of any type).

Polyhedra

A *polyhedral complex* P is defined as a set of polyhedral cells such that: (a) if p P, then any face of p is in P; (b) if p, q P, then either their intersection is empty, or coincides with both the intersection of their boundaries and the union of a P subset. We denote as $P^{i,n}$ the set of polyhedral complexes of intrinsic dimension i embedded in ^{n}. Intrinsic dimension of a polyhedron is the dimension of the maximal simplices contained in it; the dimension of the embedding space coincides with the number of coordinates of vertices. The type polyhedron "POL" is the set of all polyhedral complexes, together with a set of operators and a set of constants.

Predefined operations on polyhedra are (a) the regularized set operations (operators are overloaded): *union* "+", *intersection* "&", *difference* "-", defined in a dimension independent way (Bernardini et al. 1991); (b) the *extrusion* "EXTR" (Ferrucci et al. 91, Bernardini et al. 1991), defined as

$$\underline{\text{EXTR}}: P^{m,n} \;\square\; P^{m+1,n+1}$$

where $\underline{\text{EXTR}}$\{d\}:P maps to the cartesian product P • [0,d], where [0,d] is an interval of the real axes. The expression with default specification $\underline{\text{EXTR}}$:P instead evaluates to P • [0,1]; (c) the specialized operators described in the next sections. Predefined constants are: the empty polyhedron $\underline{\text{EMPTYP}}$; the standard simplices $\underline{\text{s}}$\{n\}, where $\underline{\text{s}}$\{n\} is defined as the convex combination of the n+1 points

$$<0,0,...,0>,\ <1,0,...,0>,...,\ <0,0,...,1>.$$

In PLASM there is no external (symbolic) representation of polyhedra with special language constructs. Each polyhedron in POL can be generated by evaluating some language expression. Some predefined operators allow the input/output of polyhedra from/to external text files using a suitable format specification, i.e. the "winged representation" described in (Paoluzzi et al. 90; Ferrucci et al.□91). The internal representation of polyhedra in PLASM is planned to be decompositive, where each polyhedral cell is represented as a set of quasi-disjoint convex sets. Other internal representations are possible, e.g. simplicial decompositions either of the boundary (Paoluzzi et al. 1989) or of the interior (Paoluzzi et al. 90) or boundary recursive (Rossignac et al. 89).

Some predefined operators

Basic functions

Following FL, we have that:

$$\underline{\text{IF}}:<p,f,g>:x = \begin{cases} f:x \text{ if } p:x = \text{true}, \\ g:x \text{ if } p:x = \text{false}; \end{cases}$$

$\underline{\text{s1}}$, $\underline{\text{s2}}$,..., $\underline{\text{sn}}$ select the first, second and n-th element of a sequence; $\underline{\text{SEL}}$ applied to a sequence of integers evaluates to the sequence of corresponding selector functions: $\underline{\text{SEL}}$:<$\underline{1}$..$\underline{n}$> = <s1,s2, ...,sn>, where the primitive infix function .. is the integer interval constructor

$$<i1,in> = <i1,i1+1,...,in-1,in>.$$

NOP is a function which has no effect, i.e. returns nothing, for any argument; ID:x = x, for any x, is the standard identity function. K is the constant function, defined as K:x:y = x for any y; CARD gives the integer cardinality of a sequence. CONS is the usual sequence constructor, TAIL removes the first element of a sequence:

CONS:<x,<x1,...,xn>> = <x,x1,...,xn>
TAIL:<x1,...,xn> = <x2,...,xn>

MKSEQ and MKPOL are sequence and polyhedra constructors, respectively. The first, when applied to an atom of type POL (polyhedral complex), returns the sequence of its higher order cells; the second, when applied to a sequence of quasi-disjoint polyhedra of the same order, returns the corresponding complex as an atom.

Structure & traversal

Structure is our denotation for the expression STRUCT:expr, where STRUCT is a predefined function and expr is a sequence of structures, transformations and polyhedra. The evaluation of a STRUCT function is called *traversal* of a structure. The result of a structure traversal is a polyhedral complex, possibly non connected or even empty. Each polyhedral component of the sequence is called *substructure*. The STRUCT operator allows to define hierarchical objects. The domain of such operator is very close to the set of PHIGS structures, with two main differences. First, all the substructures must have the same dimension; e.g., to mix polylines with polygons is not allowed. Second, an implicit Boolean operation (progressive difference) is introduced between the substructures. In particular, the default evaluation of the polyhedron P = STRUCT:<x1,x2,...,xm,xn>, where x1,..,xn are polyhedral expressions, is

P = STRUCT:<x1,x2,...,xm,xn> =
= MKPOL:<x1,-:<x2,x1>,...,-:<xn,xm,...,x2,x1>>

where the minus function denotes a Boolean difference operation. When some transformations Ti,Tj,...,Tk are contained in the sequence to which the STRUCT function is applied, the following reduction rules apply:

STRUCT:<Ti,xp,...,Tj,xq,...,Tk,xr,...,xs> =
= STRUCT:<Ti,xp,..,STRUCT:<Tj,xq,..,STRUCT:<Tk,xr,..,xs>>>
STRUCT:<Tk,xr,...,xs> =
= STRUCT:<STRUCT:<Tk,xr>,...,STRUCT:<Tk,xs>>

Finally we have, with xr,xs POL and Tk in TRANSF:

STRUCT:<Tk,xr> = xs; STRUCT:<Tk> = NOP; STRUCT:<xr> = xr;

Notice that this evaluation mechanism *has the same semantics than the traversal of PHIGS structures* or than other popular handling mechanisms of local coordinates in similar graphics systems. Commands like pushmatrix or popmatrix in the Silicon

Graphics GL library are e.g. implementable in PLASM simply by opening or closing a STRUCT application.

Structures have a central role in the PLASM language. In particular:
(a) Structures do not allow the interpenetration of solids. Hence, if used smartly, they allow the user to define complex geometrical objects without explicitly defining the low-level geometrical details, as it can be seen in the Figure 2, which contains the object resulting from a STRUCT function evaluation, after the execution of implicit differences between substructures. Notice that a different order in the sequencing of substructures may result in a very different evaluated object (see Figures 2b, 2c). This characteristic can be very useful to the designer;
(b) They give the language the same expressive power of PHIGS, as (i) the parts of an objects can be defined in local modeling coordinates; (ii) the object parts can be instanced many times in different locations by simply inserting their name after proper transformation matrices (the traversal can also be implemented as a Depth-First-Search, by maintaining the matrices in a stack and computing a current transformation matrix, which is applied to polyhedra during a structure traversal);
(c) both the previous properties result in a very natural and powerful way of modelling the building objects, which are always strongly structured and with some amount of shape iteration.

Some examples of use of structures are given in the following sections. Other examples can be found in (Paoluzzi 1990).

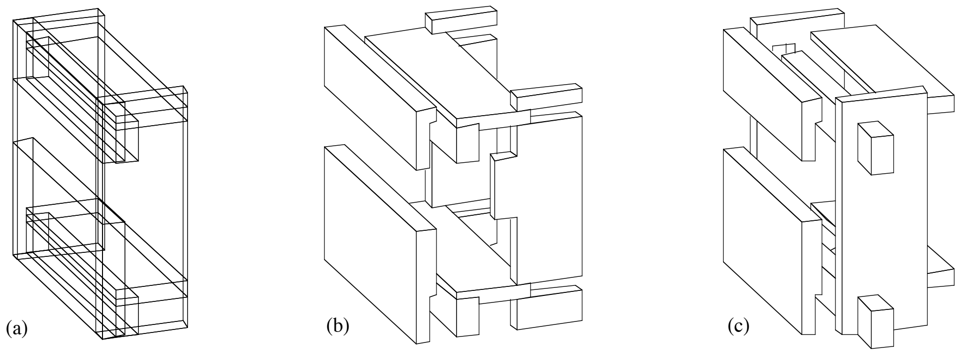

Figure 2. An evaluated structure defined by four objects, each one containing two parallelopipeds. (a) Structure definition as a sequence of isothetic (interpenetrating) parallelopipeds; (b) PLASM evaluated expression, assuming that each substructure is given in world coordinates STRUCT:<Beams,Roofs, Enclosures,Partitions>; (c) PLASM evaluated expression STRUCT:<Beams,Partitions,Enclosures,Roofs>.

Sizing and positioning

In this subsection some primitive and non primitive operators are discussed. The latter are of general usefulness, and therefore will be stored in PLASM permanent packages. The applications

MAX{i1..in}:x, MIN{i1..in}:x, MED{i1..in}:x

evaluate to the sequence of the maximum, minimum or medium coordinates of the argument x. The default forms MAX:x, MIN:x and MED:x evaluate to the full sequence of coordinates of the specified x point. The function SIZE, which returns the sequence of measures along a specified subset of coordinates (remember the application rules with optional integer parameters), is defined as

Def SIZE{i} = -:[MAX{i},MIN{i}]

The function BOX is defined as follows, where the result is an object of type POL:

Def BOX{i1..in} = Struct:[T{i1..in}:MIN{i1..in},*:a":SIZE{i1..in}]

where a"<x1,...,xn> = <"x1,"x2,...,"xn> generates a sequence of 1D polyhedra corresponding to some edges of the containment box. The " constructor and the "*" product function are described in the next sections. When BOX is applied to a polyhedral expression, i.e. we have BOX{i1..in}:x, the minimal nD parallelopiped, oriented as the reference frame, is returned which contains the projection of x in the subspace of coordinates x1,...,xn. The application

CENTRE:x

apply a translation to the argument, in such a way that it results referred to an origin located at the centroid of its containment box. This function can be defined as:

Def CENTRE{{i}} = STRUCT:[T{i}:-MED{i}, ID]

The application of the binary operator ALIGN{<i,MAX,MIN>}:<x1,x2> is equivalent to the traversal of a structure constituted by the two argument expressions, with a translation inbetween, which relocates x2 with respect to x1, according to the location directive given in the optional argument sequence. For the function to work properly, x1 and x2 must be embedded in the same space (same number n of coordinates), and i must range between 1 and n. Each alignment directive must have the form <i,Pos1,Pos2>, with Pos1,Pos2 Œ {MIN,MAX,MED}, and specify the relative location of Obj1,Obj2 along the i-th coordinate direction, according to the following definition:

Def ALIGN {{<i,Pos1,Pos2>}} (Obj1:isPol,Obj2:isPol) =
STRUCT:<Obj1, T{i}:(Pos1{i}:Obj1 - Pos2{i}:Obj2), Obj2>

The expressive power of the function ALIGN in architectural composition may be very high, especially when used within hierarchical expressions. The Figure 3 shows the result of application of this function in a simple building front, made with a wall and a window. Note that with a suitable use of such construct any kind of score can be obtained. Notice also that the function works properly when applied to a sequence argument with more than two elements.

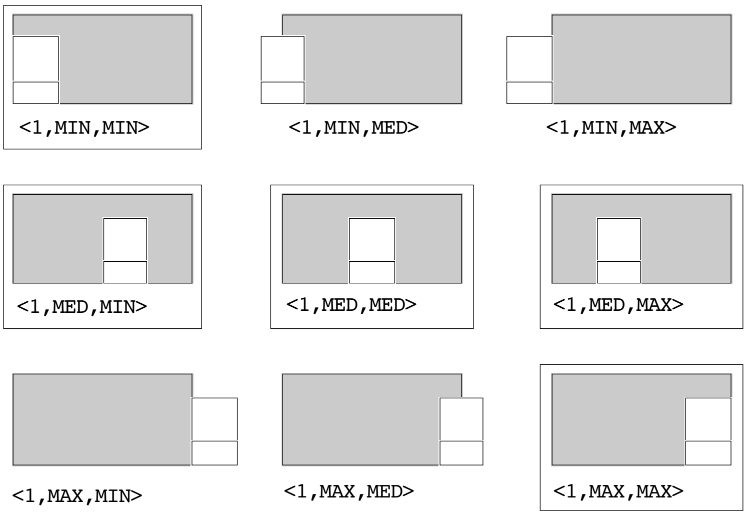

Figure 3. Front object for different values of the alignment directive along the x direction. The most useful are highlighted by enclosing them in a box.

Repeat patterns

Two repeat binary functions #:<n,x>, ##:<n,sequence> (or n#x and n##sequence) can be used with the aim of replicating a PLASM expression or sequence an integer number of times. They can be particularly useful within a " operator instance (discussed in the following) or within structures. The repeat operators establish the following equivalences:

<2 # <1,2>,3> = <<1,2>,<1,2>,3>
<4 # 1> = <1,1,1,1>
<2 ## <1,Fun:x>> = <1,Fun:x,1,Fun:x>

If Step is a polyhedron given in local coordinates, then a stair operator with a variable n number of steps and a variable shape of step can be defined as follows:

```
Def Stair (n:isInt, Step:isPol) =
    STRUCT:<    STRUCT:<S{3}:(3/4), Step>,
    (n-1) ## STRUCT:<T{1,3}:<(3/4)a, (3/4)b>, Step> >
    Where  a = SIZE{1}:Step,  b = SIZE{3}:Step, 1 ≤ n, n ≤ 20  End
    Default  n = 10, Step = MKBOX{1..3}:<28,80,16>  End
```

isInt and isPol are two predefined predicates, used for type-checking of unbound objects. MKBOX (with 3 optional integer parameters) is used as the constructor of a 3D box polyhedron. Default values for n and Step are given in the definition, to be used when the parameters are not instanced or evaluate to error or exception. Default values are also used to perform the graphical echo of function definition. The resulting polyhedral complex is shown in Figure 4.

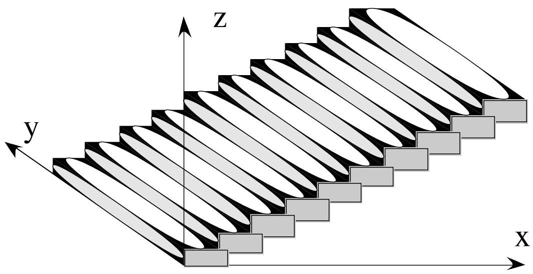

Figure 4. The complex resulting from the evaluation of both the expressions Stair:<10,Step> or Stair.

High level operators on polyhedra

Boundary

The boundary operator @ is a mapping:

$$@: P^{n,m} \Delta P^{n-1,m}, \quad 1 \leq n \leq m.$$

E.g., the boundary of a 3D polyhedron is a complex of 2D polyhedra whose support space (union of cells) coincides with the set of points which do not belong neither to the interior nor to the exterior of the polyhedron; the boundary of a 2D polyhedron is a 1D complex, i.e. a set of line segments (see Figure 5b). The boundary of a structure STRUCT:<x1,...,xn> may be obtained by evaluating the expression @:STRUCT:<x1,...,xn>.

Skeletons

The r-skeleton K^r of an n-complex P Œ $P^{n,m}$ $(n \leq m)$ is the maximal regular subcomplex with implicit dimension r. We call *r-extractor* the mapping

$$@r: P^{n,m} \Delta P^{r,m}, \quad 0 \leq r \leq n$$

such that @r:P is the r-skeleton of the polyhedral complex P. So, @0:P denotes the set of P vertices, @1:P the set of vertices and edges, while @2:P is the set of vertices, edges and faces. In Figure 5 is shown the result of the expression -:[@,@1]:Plan which is equivalent to @:Plan - @1:Plan. The function "-" denotes in this context a Boolean difference between sets of points having intrinsic dimension 1D, i.e. less than the dimension of the embedding space (2D). Such objects cannot be considered "solids". A set operation between non-solid objects is computed as the union of the results of the same operation between pairs of "solid" subcomplexes lying in the same subspaces.

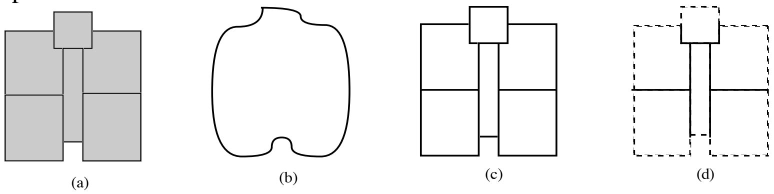

Figure 5. (a) The 2-complex Plan; (b) the 1-complex @:Plan; (c) the 1-complex @1:Plan; (d) the complex -:[@,@1]:Plan (solid lines).

Generation of 1D polyhedra

The operator " is used to define, easily (using local coordinates) and compactly, 1D polyhedra embedded in the 1D space. Such polyhedra are then used as sets of linear measures by other PLASM operators. Such "dimensioning" operator " is defined as a mapping:

": (¨ - {0})* → P ^{1,1}.

A special form of the syntax allows to write "x instead that ":x. Negative parameters are used to introduce empty intervals in the evaluated complex. The semantics of such operator is, once again, a reduction to a structure traversal :

Def " = STRUCT:a:(IF:<isPos, 1##[STRUCT:[S{1},K:s{1}],T{1}], T{1}:->)

We remember that s{1} is the standard 1D simplex, i.e. the line interval [0,1], so that STRUCT:[S{1}, K:s{1}]:x, with x Œ REAL, evaluates to the 1D polyhedron which covers the interval [0,x] of the real line. In Figure 6 two 1D polyhedra generated by applying " to two sequences of real expressions are shown. Some more examples of such operator are given later in the paper.

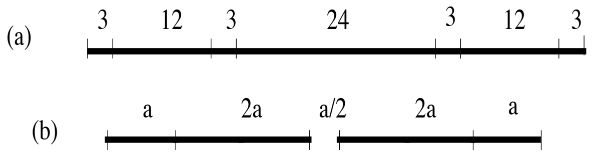

Figure 6. 1D polyhedra generated by "quoting" sequences of real expressions: (a) "[s2,3*s1,s2, 6*s1,s2,3*s1,s2]:<4,3>; (b) "[ID,2*ID,-ID/2,2*ID,ID]:a

Product of 1D polyhedra (grid generation)

The *product* operator produces, starting from a sequence of n polyhedral complexes in POL^0 ª POL^1, a complex of intrinsic dimension r ≤ n, where r is the sum of intrinsic dimension of the arguments:

*:<x1,x2,...,xn> = x Œ POL^r , x1,x2,...,xn Œ POL^0 ª POL^1

It is allowed to write the application of this operator in infix form, as the followinq equation holds

:<x1,x2,...,xm,xn> = <:<x1,x2,...,xm> ** xn>

where ** denotes the power operation described in the next section (The true reason is that the mathematical definitions of the two operations coincide). The cells of the output complex of the expression X1 * X2 are generated by computing cartesian products between any pair of cells of the argument complexes:

X1 * X2 = {x| x = x1 • x2, x1 Œ X1, x2 Œ X2} e.g., the line segments of a 3D grid with cubic cells of side a can be generated as GridEdges:a, where the definition of the function GridEdges follows (see Figure 7).

Def GridEdges = @1:*[x,y,z]
    Where x = "[4 # ID], y = "[5 # ID], z = "[4 # ID] End

Similarly, the arrangement of polygons shown in the exploded view of Figure 7c is obtained as

Def GridFaces = @2:*[x,y,z]
    Where x = "[2 # 2*ID], y = "[3*ID], z = "[3 # ID] End

It can be useful to trace the evaluation process of the expression GridFaces:a

GridFaces : a =
    = @2:*:[x,y,z]:a =
    = @2:*:<x:a,y:a,z:a> =
    = @2:*:<"[2 # 2*ID]:a,"[3*ID]:a,"[3 # ID]:a> =
    = @2:*:<"[2*ID,2*ID]:a,"[3*ID]:a,"[ID,ID,ID]:a> =
    = @2:*:<"<2*ID:a,2*ID:a>,"<3*ID:a>, "<ID:a,ID:a,ID:a>> =
    = @2:*:<"<2a,2a>,"<3a>, "<a,a,a>> =
    = @2:*:<STRUCT:<STRUCT:<S{1}:2a,s{1}>,T{1}:2a,
            STRUCT:<S{1}:2a,s{1}>,T{1}:2a>,
        STRUCT:<STRUCT:<S{1}:3a,s{1}>,T{1}:3a>,
        STRUCT:<STRUCT:<S{1}:a,s{1}>,T{1}:a,
            STRUCT:<S{1}:a,s{1}>,T{1}:a,
            STRUCT:<S{1}:a,s{1}>,T{1}:a>> =
    = @2:*:<STRUCT:<L_{2a},T{1}:2a,L_{2a},T{1}:2a>,
        STRUCT:<L_{3a},T{1}:3a>,
        STRUCT:<L_a,T{1}:a,L_a,T{1}:a,L_a,T{1}:a>> =
    = @2:*:<STRUCT:<L_{2a},STRUCT:<T{1}:2a,L_{2a}>>,
        STRUCT:<L_{3a}>,
        STRUCT:<L_a,STRUCT:<T{1}:a,L_a,STRUCT:<T{1}:a,L_a>>>> =
    = @2:*:<P_1^1, P_2^1, P_3^1> = @2:P_{123}^3 = P_{123}^2

where: L_{2a}, L_{3a}, L_a respectively represent (in local coordinates) the 1D segments of length 2a,3a,a; P_1^1,P_2^1,P_3^1 are complexes of 1D polyhedra which contain 2,1 and 3 relocated instances of L_{2a},L_{3a},L_a, respectively; P_{123}^3 is the 3D complex with solid parallelopiped cells generated by the product of P_1^1,P_2^1,P_3^1; and finally P_{123}^2 is the 2-skeleton of P_{123}^3 (see Figure 7c).

A combined use of the operators " and * is shown in the following. The 2-complex Plan contains four squares of side a, two squares of smaller side b, and a rectangle with sides b and 2a-2b = 2(a-b). The four bigger squares can be defined as "<a,-b,a>*"<a,a>. The central part of the object is given, in world coordinates, as

"<-a,b>*"<-b/2,b,2(a-b),b>.

The whole parametric definition may be given, using local modeling coordinates, as:

<u>Def</u> Plan (a:<u>isReal</u>, b:<u>isReal</u>) =
<u>STRUCT</u>: <"<a,-b,a>*"<a,a>, <u>T</u>{1,2}:<a,b/2>, "<b> *"<b,2(a-b),b>>
<u>Where</u> 0 ≤ b, b ≤ a <u>End</u>
<u>Default</u> b = 24, a = 48 E<u>nd</u>

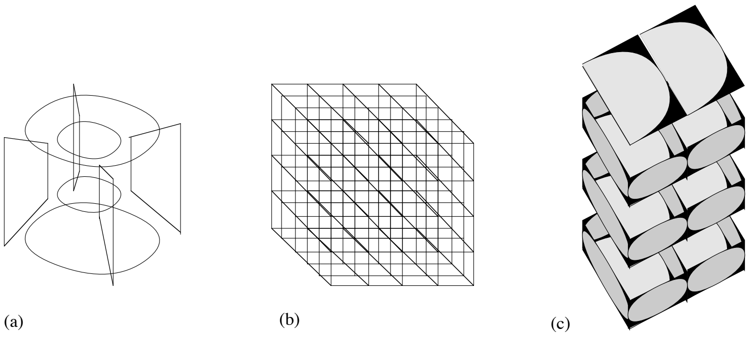

Figure 7. (a) Perspective view of the edges @1:*:<4 # "1> of the 4D standard hypercube, which can also be described as @1:("1*"1*"1*"1); (b) axonometric view of the object GridEdges:1 defined in the text; (c) exploded view of the arrangement of polygons in $^{..3}$ obtained by evaluating the expression GridFaces:a

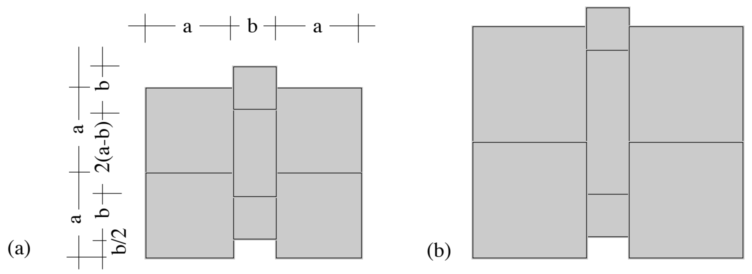

Figure 8. Two instances of the parametric complex Plan, corresponding to different values of the parameter a.

Power of polyhedra

The power of polyhedra is defined as a mapping from pairs <Basis,Exponent> of polyhedra to polyhedra. The exponent argument must have either dimension 0 or 1. The power operator "**" extrudes the polyhedron Basis Œ P ^{n,m} into $^{..m+1}$ if the polyhedron Exponent is in P ^{1,1}, otherways simply embeds Basis into $^{..m+1}$. In particular, if Basis and Exponent are two polyhedral complexes with proper dimensions, and b • e is the point set obtained as cartesian product of two polyhedral cells, we can write:

$$**:\ P^{n,m} \bullet P^{i,1}\ \Delta\ P^{n+i,m+1} \quad i\ Œ\ \{0,1\},$$

Basis ** Exponent = {p|p = b • e, b Œ Basis, e Œ Exponent}

In Figure 9 two examples of power application with basis Plan are given, showing both a multiple embedding of the 2D object Plan in 3D (depending on the real cells of the 0D exponent) and the generation of a multiple instance of a solid (3D) version of Plan (depending on the cells of the 1D exponent). A more complex and interesting example is shown in Figure 10, where the power operator is combined with the 2D skeleton extractor function.

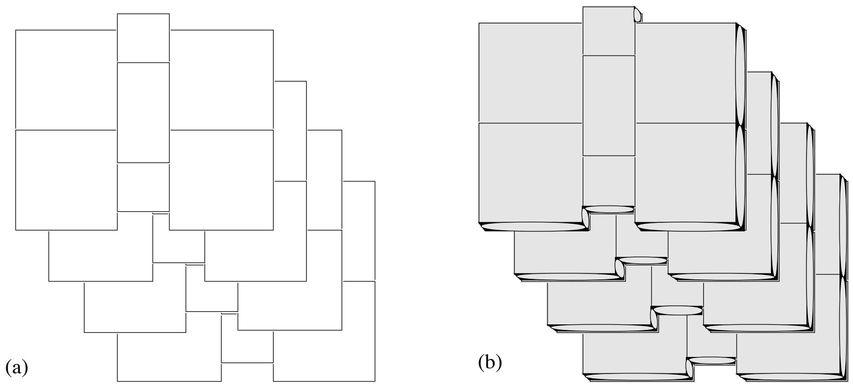

Figure 9. (a) The polyhedron Plan ** (@0:"<h,h,h>). (b) The polyhedron Plan**"<w,-(h -w),w,-(h - w),w,-(h - w),w>; the result is a solid complex whose connected components have width w and are located at distance h-w.

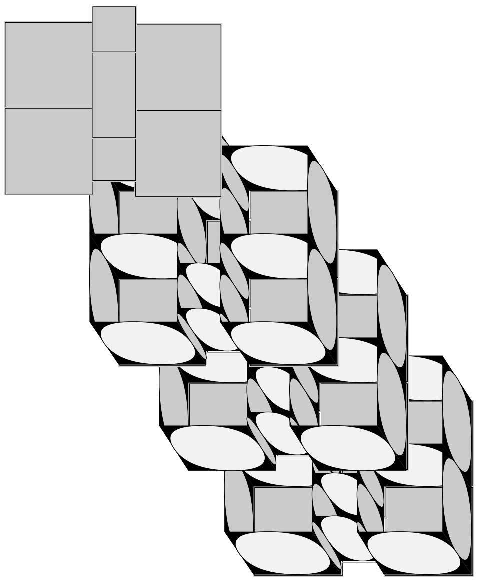

Figure 10. A graphical representation of the 2-complex in ^{3} generated by evaluating the expression @2:(Plan ** "<h,h,h>), where h is the distance between adjacent floors. The function can be abstracted with respect to h and to the number n of floors: @2:**:[K:Plan,":#:[s1,s2]]:<n,h>. A more useful abstraction is @2:**:[s3,":#:[s1,s2]]:<n,h,Plan>, and a new operator Def Block (x1:isInt,x2:isReal,x3:isPol) = @2:**:[x3, ":#:[x1,x2]] can be defined, such that the model in the figure can be generated as Block:<3,h,Plan>. Obviously, for different values of Plan or h a completely different 3-floor model may be obtained.

An extended form of the power operator, which applies parametric transformations to the cells of the resulting complex, is defined in (Paoluzzi 1990). An example of generation of a circular stair using two nested instances of the (extended) power operator is also given there.

Intersection of extrusions

The intersection of extrusions, denoted with the symbol "&&", is an operator from sequences, with length at least 2 and at most n, of polyhedral complexes in $P^{n-1,n-1}$ to the polyhedral complexes in $P^{n,n}$

$$\&\&: (P^{n-1,n-1})^m \rightarrow P^{n,n}, \qquad 2 \leq m \leq n$$

The arguments of such operator will be usually called *sections*. The infix form is allowed only when applied to two arguments, as the operator is not associative. The effect of the evaluation of the && operator is a polyhedral complex whose cells are obtained by intersecting, in all the possible ways, the cells of the straight infinite extrusions of the arguments. Such extrusions are computed after having embedded the arguments into coordinate hyperplanes of the embedding space of the result.

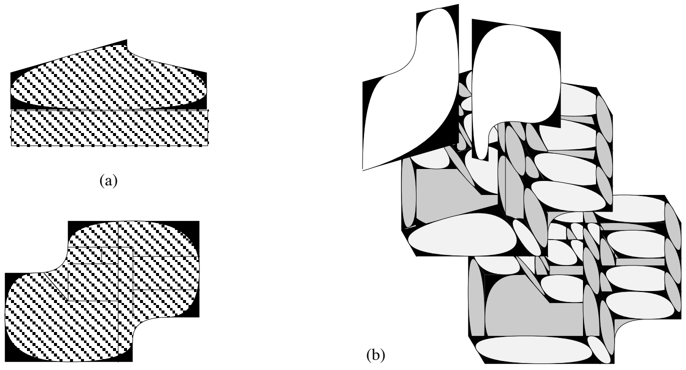

Figure 11. (a) The arguments Section and Plan of a && operation; (b) exploded view of the 2D complex generated by the expression @2:&&{1,3}:<Section,Plan>.

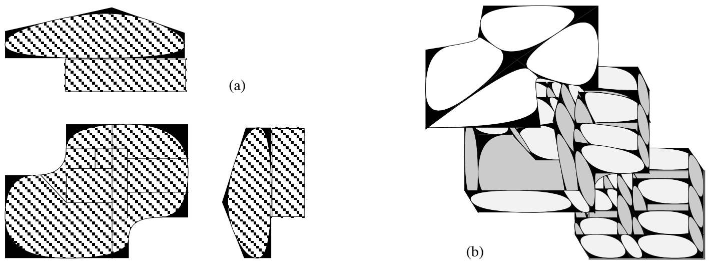

Figure 12. (a) The arguments Section1, Section2 and Plan of a && operation; (b) exploded view of the 2D complex generated by the expression @2:&&:<Section1,Section2,Plan>.

e.g., the expression &&{3,1}:<plan,section>, where plan,section Œ $P^{2,2}$, is evaluated as follows: the plan argument is embedded in the plane x3 = 0 of the 3D space and indefinitely extruded in the x3 direction, while the section argument is embedded in the plane x1 = 0 and indefinitely extruded in the x1 direction. The 3D solid polyhedral complex resulting from the operation is finally computed by intersecting the extruded cells in all the possible ways. The 2-skeleton of such complex, i.e. the set of 2D boundary faces of the 3D cells (and the set of their edges and vertices), is graphically represented in Figure 11.

Lattice of objects

The "Lattice" binary operator, used for relocating a set of instances of the same object, optionally depending on the elements of a Boolean array, is useful to define the structural grid of a building or the set of windows in the building front. The operator is applied as:

LATTICE{BoolArray}:<x1, x2>

and can be written also in a special infix form:

x1 ${BoolArray} x2

where the vertices of x1, x2 Œ POL must have the same number of coordinates. The semantics of this operator is that of generating a set of instances of the x2 argument, each one having as relative origin one of the points (0-cells) in x1. The optional argument BoolArray, nD array of Booleans with a one-to-one mapping with the x1 0-cells, controls the existence of the x2 instances, depending on the truth value of the corresponding BoolArray elements. In more formal terms, we can write that a LATTICE function is equivalent to the structure given in the following, where, as usual, the underlined words represent functions or keywords predefined in the language.

```
Def LATTICE (BoolArray, x1:isPol, x2:isPol) =
    STRUCT:a:(IF:<s1,INSTANCE,NOP>):PAIRS
    Where       INSTANCE = STRUCT:[T{1..CARD:s2}:a-:s2, K:x2]
                PAIRS = a:a:[1 ## SEL:<1..LAST>]:<BoolArray,@0:x1>
                LAST = MIN:<CARD:BoolArray,CARD:@0:x1> End
```

Restricted offset

The "OFFSET" operator "++" generates a polyhedral complex in $P^{n,n}$ starting from a polyhedral argument in $P^{n-1,n}$, called *basis* of the operator, and from a real number, called *width* of the operator:

++: $P^{n-1,n}$ • □ $P^{n,n}$.

The OFFSET operator is used to increase the intrinsic dimension of the argument, i.e. to make it solid. It is also possible to use an extended form

++: $P^{n-1,n}$ • • □ $P^{n,n}$.

where the two real arguments are called *right width* and *left width*, with respect to the internal orientation of the (highest order) cells of the basis polyhedron.

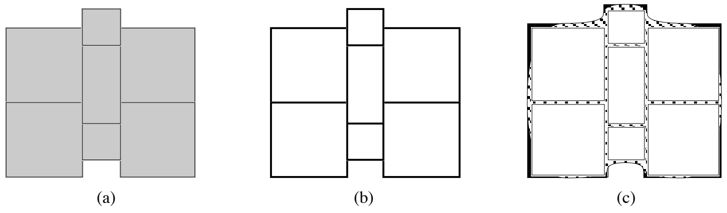

Figure 13. (a) The 2D polyhedral complex Plan; (b) the 1D complex @1:Plan; (c) the polyhedron generated by the expression ++:@1:Plan.

It is very easy to compute the OFFSET function when the base argument is the n-1 skeleton of a solid complex P Œ POL^n. It is possible to show that in this case the function can be computed as a Boolean difference between the object +:MKSEQ:P, where the expression represents the Boolean sum applied to the sequence of P cells, and MKSEQ:P, after having suitably enlarged/shrunk the objects, with a method described in (Paoluzzi 1991). This can be done in a very efficient way when the internal representation of polyhedral cells is a boundary representation.

```
Def OFFSET (P:isPol{n},width:isReal) = -:CONS<
    MoveVert{width/2}:+:MKSEQ:P, MoveVert{-width/2}:MKSEQ:P >
```

Replacement

According to (Mitchell 1991), we notice that in design development by top-down refinement it can be very useful to have a special function to quickly explore the consequences of refinement decisions, without loosing neither the previous definitions, nor the corresponding compiled objects. Such a function may be applied as follows:
REPLACE:<ScriptName, NewScriptName, OldExpr, NewExpr>

The effect of such a special function (notice that it is not a standard PLASM function) is that of generating a temporary instance of the definitions in ScriptName, without the object in OldExpr and with an instance of the evaluated NewExpr, mapped in the same containment box of OldExpr. REPLACE is hence an operator over PLASM scripts; its semantics can be outlined as follows:

(a) select the script corresponding to ScriptName, and make a copy called NewScriptName;
(b) replace in NewScriptName all instances of OldExpr with ScaledNewExpr;
(c) insert within a clause Where End in NewScriptName the definition Def ScaledNewExpr= STRUCT:<MAP:BOX:<NewExpr,OldExpr>, NewExpr>
(d) evaluate NewScript.

where MAP is a Predefined PLASM binary operator which takes as arguments two cuboids with the same intrinsic dimension and gives as result the transformation matrix which maps the first argument into the second. Remember that a unary function (like BOX), when applied to a sequence of arguments, evaluates to the sequence of applications.

Conclusions

In this paper the current status of development of the new design language PLASM has been presented. In particular the most significant decisions about syntax and semantics of the language has been discussed. Some important features of the language concern its dimension independent approach to the modeling, its general approach to variational geometry, the introduction of some new and powerful domain-oriented solid operators, the use of local coordinates for each object definition, and the reduction of most functions to the traversal of a structure.

The PLASM language is inspired to the very powerful applicative style introduced by the Backus's languages FP and FL for programming at function level. When the language is used at low-level, it may appear a bit cryptic, but it is intended as the kernel of specialized high-level macro-languages to be used in different areas of architectural design (e.g. housing, hospitals, architecture of interiors, etc.). A more verbose version of the syntax might be useful for non-technical user.

We believe that a key point for the successful acceptance of the language will be a well designed graphical user interface. Such interface should symbolically trace any graphical action of the designer on the currently selected shape (picking, grouping, cutting, pasting, dragging, etc), as well to provide graphical echo of expression evaluated in the listener window. Furthermore, the language should be considered a component of a very high level graphical environment, with solid modeling, rendering and real time visual simulation subsystems. Our aim is to provide standard interfaces toward such subsystems, providing at the same time a complete independence of the language, so that it will be possible to exploit the future hardware and software developments.

An huge computing power is required for using the language in practical cases, but next generation graphics workstations (the G-machine with more than 50 Mflops for $ 10,000 (Van Dam 1991)) are at the door, and parallel machines are already here.

Acknowledgements

This work has been supported in part by the "PF Edilizia" of the Italian Research Council, under the contract CT CNR 90.01702PF64. We would like to thank Claudio Gori Giorgi and Ettore Zambelli for encouragement and help in starting the project; Chanderjit Bajaj, Luigia Carlucci Aiello and Michael O'Connor for useful discussions and advice; Lorenzo Saladini for his help in revising the language syntax and making it more close to FL; the referees for their comments and suggestions. Our is the responsibility of any error and imprecision (e.g. the applicative style used in a naive way) in the paper, which must be considered a work in progress report. Some figures have been produced with the solid modeler MINERVA, developed by the CAD Group at DIS.

References

Backus, J. "Can Programming Be Liberated from the Von Neuman's Style? A Functional Style and its Algebra of Programs". 1978. Communications of the ACM, ACM Turing Award Lecture.

Backus, J., Williams, J.H., Wimmers, E.L., Lucas, P., and Aiken, A. 1989. *FL Language Manual, Parts 1 and 2.* Research RJ 7100 (67163), Computer Science, IBM Almaden Res. Center.

Barford, L.A. 1986. "Representing Generic Solid Models by Constraints". Dept. of Computer Science, Cornell University, Tech. rep. 86-801, Ithaca,NY.

Borning, A.H. 1981. "The Programming Language Aspects of ThingLab, a Constraint-Oriented Simulation Laboratory". *ACM Trans. on Programming Languages and Systems* 3 n° 4, 353-387.

Borning, A.H. 1986. "Defining Constraints Graphically". Proc. of *CHI'86 Conference*, 137-143.

Chitham, R. 1985. *The classical orders of architecture.* London: Architectural Press.

Bernardini, F., V. Ferrucci and A. Paoluzzi. 1991 "Working with Dimension Independent Polyhedra". Tech. Report TR-07-91, Dip. di Informatica e Sistemistica, Univ. "La Sapienza", Rome.

Flemming, U. "Syntactic Structures in Architecture: Teaching composition with Computer Assistance." In McCullough, M., W. J. Mitchell, and P. Purcell (eds.). 1990. *The Electronic Design Studio.* Cambridge, Mass.: MIT Press.

Ferrucci, V., and A. Paoluzzi. 1991. "Extrusion and boundary evaluation for multidimensional polyhedra". *Computer Aided Design*, 23 n° 1, 40-50.

Ferrucci, V. and F. Bernardini. 1990. "Simple_X^n: User Manual and Implementation Notes. Parts 1 and 2". Tech. Report, Dip. di Informatica e Sistemistica, Univ. "La Sapienza", Rome.

Gossard, D.C., R.P. Zuffante and H. Sakurai. 1988. "Representing Dimensions Tolerances, and Features in MCAE Systems". *IEEE Computer Graphics and Applications*, 8 n° 2, 51--59.

Goguen, J.a. and T. Winkler. 1988. "Introducing OBJ3", Res. report, RSI-CSL-88-9, Menlo Park, California.

PHIGS (Programmer's Hierarchical Interactive Graphics System). 1987. Draft Standard ISO dp9592-1. International Standards Organization, Geneva.

Kalay, Y. E. 1989. *Modeling Objects and Environments*. New York. NY: Wiley Interscience.

Kalay, Y. E. 1989. "The hybrid edge: a topological data structure for vertically integrated geometric modeling". *Computer Aided Design*, 21 n° 3, 130--140.

Knuth, D.,E. 1984. *The TeXbook*. Reading, Mass: Addison Wesley.

Lamport, L. 1986. *LaTeX*. Reading, Mass: Addison Wesley.

Le Corbusier. 1965. *Le Modulor*. Boulogne: Editions de l'Architecture d'Aujourd'hui.

Leler, W.M. 1985. "Constraint Languages for Computer Aided Design". *SIGDA Newsletter* 15 n° 2, 11-15.

Leler, W.M. 1988. Constraint Programming Languages, Reading, Mass.: Addison-Wesley.

March, L. and P. Steadman. 1971. *The geometry of environment*. London: RIBA Publ.

Mitchell, W.J.. 1990. The Logic of *Architecture*. Cambridge, Mass:□The MIT Press.

Mitchell,W.J., R.S Liggett and T. Milton. "Top-Down Knowledge-Based Design". In McCullough, M., W.J.□Mitchell, and P. Purcell (eds.). 1990. *The Electronic Design Studio*. Cambridge, Mass.: MIT Press.

Paoluzzi, A. 1990. "A set of new solid operators, with CAAD examples", manuscript.

Paoluzzi, A. and C. Cattani. 1990. "Simplicial based representation and algorithms for multidimensional polyhedra". Submitted paper. A preliminary version appeared as: C. Cattani and A. Paoluzzi. "Solid Modeling in Any Dimension", Dip. di Informatica e Sistemistica, Univ. "La Sapienza", Tech. rep. 02-89, Rome, 1989.

Paoluzzi, A., M. Ramella, and A. Santarelli . 1989. "Boolean Algebra over Linear Polyhedra". *Computer Aided Design*, 21 n° 8, 474--484.

Pixar. 1989. *The RenderMan Interface Specification*. San Rafael, California.

Requicha,A.A.G. 1980. "Representations for Rigid Solids: Theory, Methods and Systems". *ACM Computing Surveys* 12 n° 4, 437--464.

Rossignac J.R. and M.A. O'Connor. 1990. "SGC: A Dimension-independent model for pointsets with internal structures and incomplete boundaries". In M. J. Wozny, J.U. Turner and K.Preiss (Eds.), 1990. *Geometric Modeling for Product Engineering*, Proc. of the 1988 IFIP/NSF Workshop on Geometric Modelling, Rensselaerville, NY, September 18--22, 1988. North-Holland, 145--180.

Schmitt G. "Classes of Design - Classes of Tools". In McCullough, Malcolm, William J. Mitchell, and Patrick Purcell (eds.). 1990. *The Electronic Design Studio*. Cambridge, Mass.: MIT Press.

Steele, G.L. 1975. "The Definition and Implementation of a Computer Programming Language Based on Constraints". PhD Thesis, Stanford (Published as Comp. Sc. Dpt. Report STAN-CS-75-499).

Williams, J.H. 1982, "Notes on the FP Style of Functional Programming", In Darlington, J., P. Henderson and D.A. Turner (Eds.), *Functional Programming and its Applications,* Cambridge: Cambridge Univ. Press.

Wittkower, R. 1949. *Architectural Principles in the age of humanism*. London: Academy Editions.

Wolfram, S. 1988. *Mathematica*. Redwood City: Addison Wesley