A Model for Object Representation and Manipulation in a Visual Design Language

Philip T. Cox   Trevor J. Smedley

Dalhousie University, Halifax, Nova Scotia, Canada

Abstract

Languages used for design activities in CAD software are usually textual languages akin to standard procedural programming languages such as Basic or C++. This creates a discontinuity between the drafting and solid modelling aspects of design, and the programming aspects which are becoming increasingly important as designers attempt to economise on their activities by building parametrised specifications. The declarative language LSD addresses this issue by applying visual programming to design.

Here we present a formal model for objects in a design space and for operations on design objects. We also show how this model is integrated with LSD to provide a general mechanism for extending the language through the addition of new operations.

1 Introduction

Visual software tools for some design tasks, CAD/CAM systems for example, have been in widespread use for many years. Systems such as AutoCAD, ArchiCAD and MicroStation [1, 2, 8] provide sophisticated general-purpose and special-purpose tools for drawing and solid modelling. Support for parameterised designs is also provided, but it is either quite rudimentary, or requires the use of a textual language very much like a programming language. AutoCAD supplies AutoLISP for programming, but also allows connections to modules written in other textual languages, and ArchiCAD includes GDL, a low-level Basic-like language. As a result of this dichotomy between design and programming, to fulfill their need for parameterised components, users of commercial CAD systems usually purchase separate packages. For example a package for generating staircases of different styles and sizes is available for AutoCAD, and is implemented in C. Experience with visual programming languages indicates that visual languages for design might be able to provide the programming capabilities required for building parameterised designs, while at the same time integrating more closely with the drafting and solid modelling aspects of an industrial design system.

Based on this observation, a visual language for designing structured objects was proposed in [10]. This language was obtained by extending Prograph, a general purpose visual programming language [5, 9], by adding a new picture data type, rules for combining and transforming pictures, and a construct for iteratively aggregating pictures. However, even though all aspects of this language are visual, the visualisation is not homogeneous. When viewing the algorithms, the objects are not visible, and *vice versa*. The sharp division between algorithm and data in the language is a consequence of the dataflow nature of Prograph. A similar dichotomy would result if the basis were any other programming language that concentrated on process rather than specification. This leads to the conjecture that a declarative programming language may provide a more satisfactory foundation. In logic programming, for example, the primary focus is on functional expressions (*terms*), and a program consists of a set of logical sentences (*clauses*) that define the structure of terms we are interested in computing.

In [7] we noted that the visual logic programming language Lograph [3] provides a homogeneous visual representation for data and algorithms, and based on this observation, presented a preliminary proposal for a Language for Structured Design (LSD) based on Lograph. In [6] following a brief introduction to Lograph, we investigate this idea further, clearly delineating the interface between the language and the objects it manipulates, without considering the nature of the objects themselves. The descriptions of LSD presented in [6] and [7] focus on describing a particular design operation, “bonding”, which fuses two components to create a new one. However, other operations are obviously necessary, and may vary from one design domain to another.

Here we concentrate on the design-space side of the interface between language and manipulated objects, proposing a formal model for solid objects in a design space in just enough detail to integrate it into LSD. This model includes both the specification of solids in space, and operations on such solids. We then generalise the previous definition of LSD by replacing the notions of “e-component” and “bond” with abstract equivalents to solid objects and operations in a design space. We also discuss the visual representation of these entities in an LSD program, as well as some issues related to implementation of the proposed model in a CAD-like environment.

2 A short summary of LSD

To set the scene for discussing the main points we wish to make, in this section we give a short explanation of LSD using an example. In the interests of brevity, this explanation will be quite superficial so we urge the reader to consult [6] for details.

This work was partially supported by NSERC grants OGP0000124 and OGP0121312.

A program in LSD is a collection of *designs*, each of which defines a family of components. These designs can be executed in a process called *assembly*, to build explicit components. Figure 1 depicts an LSD program consisting of two designs **partial cog** and **cog**. The former defines a component, partial cog, with a given number of teeth, by recursively describing it as the component obtained by bonding a tooth on to a partial cog with one less tooth. In the recursive case of **partial cog** on the left of the figure, the icon named **tooth** is an *explicit component* (e-component), representing a two-dimensional object. The grey stripes are *bonds*, which connect edges along which components will be fused during assembly. These edges are either *open edges* of e-components, and will participate in the execution of a bond, or edge terminals, represented by ▪▪▪▪▪▪▪▪ , which propagate bonds during the assembly process. The icon named **partial cog** is an *implicit component* (i-component), representing an invocation of the design **partial cog**. The icon named **+1** is a *function cell*. Function cells are used for building abstract data structures as functional expressions, analogous to terms in Prolog. The cell **+1** occurring here plays the role of the successor function for defining integers, so during assembly, it will "decrement" the incoming integer specifying the number of teeth.

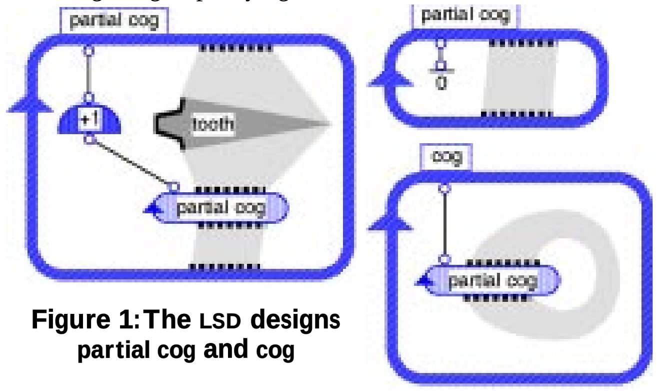

Figure 1: The LSD designs partial cog and cog

Assembly transforms a *specification*, a network of function cells and components, using four rules, *replacement*, *merge*, *deletion* and *bonding*, the first three of which are inherited from Lograph. Replacement expands an i-component by replacing it with the body of one of the cases of the corresponding design. Merge and deletion eliminate function cells, and bonding fuses e-components along open edges. Figure 2 illustrates these rules. The specification in 2(b) is obtained from the one in 2(a) by first replacing the i-component **partial cog** with the first case of the design **partial cog**, which introduces a **tooth** e-component, a **partial cog** i-component and a **+1** function cell. Merge and deletion then result in reducing the function cells **8** and **+1** to the cell **7**. Note that one of the consequences of this transformation is that the specification now contains a bond connecting two open edges. Executing this bond effects the transformation from 2(b) to 2(c), producing a new e-component. Assembly stops when a specification is produced that cannot be further transformed, and hopefully consists of an e-component. An animation of an assembly can be found at (http://www.cs.dal.ca/~smedley/papers/cog.mov).

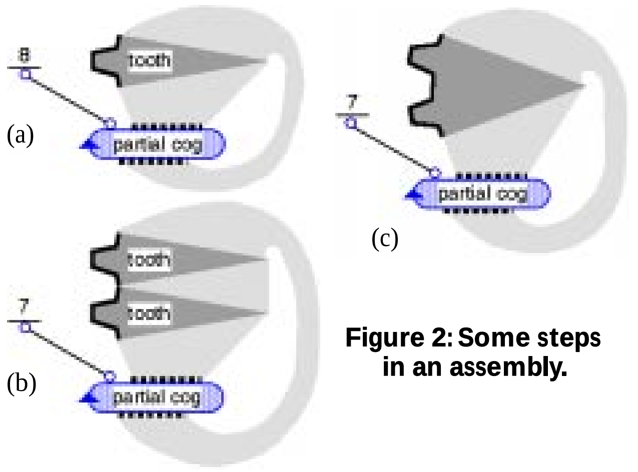

Figure 2: Some steps in an assembly.

Lograph, on which LSD is based, is a logic programming language with an underlying textual representation in which each case corresponds to a Horn clause of a special *flat* form in which there are no nested terms. Its semantics are defined by a set of deduction rules on flat clauses called *surface deduction* [4], the pictorial manifestations of which are the replacement, merge and deletion rules that LSD inherits from Lograph. The definitions below are reproduced from [6] in order that we may refer to them in later sections.

In the following we assume the existence of an alphabet consisting of disjoint sets of function symbols, predicate symbols, and variables. The set of predicate symbols is partitioned into *implicit symbols* and *explicit symbols*. The latter cannot have defining clauses. For convenience, we assume the existence of special function symbols f^0, f^1, f^2, … of arity 0, 1, 2 … respectively, which we will use for grouping terms. For each $i \geq 0$ and terms $t_1, \ldots, t_i$ we denote the term $f^i(t_1, \ldots, t_i)$ by $[t_1, \ldots, t_i]$.

Definition 2.1: A *function cell* is a literal of the form $x = f(y_1, \ldots, y_k)$ where f is some k-ary function symbol, $k \geq 0$, x is a variable, and y_i is a variable for each i ($1 \leq i \leq k$). Each y_i is a called a *terminal* of the cell, and x is called the *root*.

Definition 2.2: An *implicit component* (*i-component*) is a literal of the form $p(v_1, \ldots, v_k)$ where p is some k-ary implicit symbol, $k \geq 0$, and for each i ($1 \leq i \leq k$), v_i is a variable called a *terminal* of the component. The terminals are partitioned into two sets: *simple terminals* and *edge terminal*. The *signature* of an i-component $p(v_1, \ldots, v_k)$ is a list $(F_1, \ldots, F_k)$ where $F_i = s$ if v_i is a simple terminal, and otherwise $F_i = e$ for each i ($1 \leq i \leq k$).

Definition 2.3: An *explicit component* (*e-component*) is a set of literals consisting of

- some literals, called *open edges*, of the form $w = [[x_1, y_1, x_2, y_2], [u_1, \ldots, u_m]]$, , for some $m \geq 0$, where $w, x_1, y_1, x_2, y_2, u_1, \ldots, u_m$ are variables distinct from each other, and w, called an *implicit edge terminal*, has no other occurrences in the component; and

- a literal of the form $q(v_1, \ldots, v_k)$,called the *anchor* of the component, where q is a k-ary explicit symbol for some $k \geq 0$, and $\{v_1, \ldots, v_k\}$ is the set of all variables occurring in the open edges of the component, excluding the edge terminals.

Definition 2.4: For each explicit component e, there exists a formula K_e called the *specification* of e such that every variable occurring in e, except the edge terminals, also occurs in K_e. An e-component is *valid* iff its specification is satisfiable; otherwise it is *invalid*.

Definition 2.5: An *internal bond* is a pair of equalities of the form $u = [[x_1, y_1, x_2, y_2], w]$, $v = [[x_2, y_2, x_1, y_1], w]$, where w, x_1, y_1, x_2, y_2, u, v and w are variables distinct from each other. u and v are called *implicit edge terminals* of the bond.

Definition 2.6: A *component design* (or simply *design*) consists of a set of cases with no variables in common, such that the heads have the same implicit symbol and signature. A *case* is a flat clause the head of which is a literal of the same form as an implicit component, with simple terminals, edge terminals and signature defined analogously. The *body* of a case is a set of function cells, components or bonds, satisfying the following conditions:

- No variable occurring in an e-component or bond occurs anywhere else in the case, with the exception of the implicit edge terminals of the component or bond.
- Any variable in the case which occurs as an edge terminal or implicit edge terminal has exactly two occurrences. If one of these occurrence is in a component, the other must be in the head or a bond, otherwise both occurrences must be in the head.

The above definition of internal bond is such that the merge and deletion rules accomplish most of the bonding process as illustrated in the preceding example. However, one final step is required, requiring a minor addition to the semantics: that is we must create a new e-component out of the literals that remain from the two components involved in the bonding. This is accomplished by replacing the two anchors with a single anchor constructed with a new explicit symbol. We define the specification for the new component as the conjunction of the specifications of the two combined components, and check that this specification is satisfiable. If it is not, execution halts.

3 Solids as primitive data

The above definitions of e-components and bonds deal only with those aspects which are necessary for incorporating them syntactically into the underlying flat Horn clause representation of Lograph, and to account for their interaction with the surface deduction rules. They do not characterise the properties of e-components as objects in a design space, nor do they deal with incorporating visual representations for e-components and bonds into the language. Neither does LSD as described permit any other operations on components, which would clearly be required in a practical design system. We address these issues here by defining solids in a design space, and in Section 4 relate them to e-components.

By *design space* we mean an augmented 3-space defined by the usual three real dimensions, together with an arbitrary but fixed finite set of extra real-valued dimensions called *properties*. We will define a solid as a function which maps a vector of parameter values to a set of points in space constituting the volume of the solid, and associates with each of these points a unique value for each of the properties. Therefore, a solid in the design space actually represents a family of solids, each member of which corresponds to a particular choice of parameter values.

Although we restrict the values of properties to be real numbers, this clearly does not reduce the generality of our definitions. Also, for simplicity we require every solid to have a value for every property at every point in its volume. This might seem to be unrealistic since, for example, one may not be interested in the electrical potential at some point inside a wooden chair leg. However, since solids are parameterised, we could define the wooden chair leg as a solid with a parameter that determines electrical potential. If we provide values for all parameters except this one, we get a wooden chair leg which is fully specified in all respects except electrical potential, which we are not interested in anyway.

Here and in following sections, we denote by $|x|$ the number of elements in a set or sequence x. We will use overscored variables, as in $\bar{x}$, to abbreviate a vector of variables that should be expanded in place. For example $A(\bar{x}, \bar{y})$ is short for an expression of the form $A(x_1, \ldots, x_m, y_1, \ldots, y_n)$.

Definition 3.1: A *design space in m dimensions over r properties* for some integers $n \geq 0$ and $r \geq 0$ is the set of all subsets of $R^m \times R^r$. Note that although we are primarily interested in spaces up to three dimensions, there is no reason to limit the generality of our definitions.

Definition 3.2: If D is a design space and n is an integer ≥ 0, a *solid in D in n variables* is a function $\Phi: R^n \rightarrow D$ such that, if (v, p) and $(v, q) \in \Phi(y)$ for some $y \in R^n$, then $p = q$. By the *variables* of Φ we mean the set of integers $\{1, \ldots, n\}$. We may also use symbolic names to refer to the variables of a solid.

Definition 3.3: If Φ and Ψ are solids in n and k variables respectively, Φ and Ψ are said to be *equivalent*, denoted $\Phi \equiv \Psi$, iff $\{S \mid S = \Phi(y),\ S \neq \emptyset,\ y \in R^n\} = \{S \mid S = \Psi(y),\ S \neq \emptyset,\ y \in R^k\}$.

Definition 3.4: Let Φ be a solid in n variables, and P be a subset of its variables. Then P is said to be *sufficient for* Φ iff for all $y, z \in R^n$, if $y_i = z_i$ for all $i \in P$, then $\Phi(y) = \Phi(z)$. Clearly, if P is sufficient for Φ then the projection of Φ on to P, denoted Φ_P is a solid in $|P|$ variables and $\Phi \equiv \Phi_P$. P is called a *parameter set for* Φ iff P is sufficient for Φ and no proper subset of P is sufficient for Φ.

To illustrate these definitions, suppose we have a 2D design space with properties *colour*, *material* and *temperature*, containing an ellipse defined by a function Φ in 12 variables corresponding to the quantities marked on the diagram in Figure 3, together with *colour*, *material* and *temperature*. Let us also suppose that *colour* is determined by *material* and *tem-*

perature, and *temperature* is determined by *material* and *colour.* There are 24 parameter sets for Φ, for example (a_1, a_2, c_1, c_2, α, *material, colour*) and (a_1, a_2, x_1, y_1, α, *material, temperature*). Hence there are at least 24 solids with non-redundant variables equivalent to Φ.

Having characterised solid objects, we now turn our attention to defining operations that compute new solids from existing ones. Operations performed in a CAD system or other design environment are usually of three kinds: applying some transformation to a single object, combining two objects to create a new one, or grouping objects [1,2]. The first can be characterised as constraining the given object in some way; for example creating a cube from a rectangular solid. Operations in the second category involve positioning, orienting or scaling two objects relative to each other, while at the same time blending them into one. Such an operation can be viewed as constraining the two objects while combining them with some set operation. Grouping operations can be regarded as applying some constraint to a set of objects. Therefore, to define operations on solids, we need to be able to capture two notions: combining objects viewed as sets of points, and constraining objects. To this end, we define a generic concept "operation".

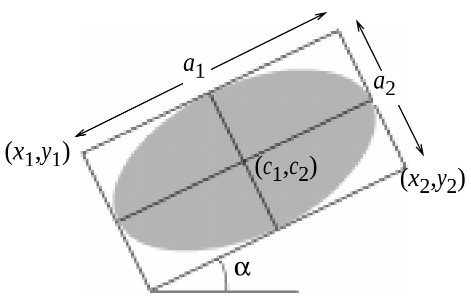

Figure 3: An overspecified solid.

Definition 3.5: If $\mathcal{D}$ is a design space and n is a positive integer, an *n-ary operation in* $\mathcal{D}$ is a 4-tuple ($\mathcal{P}$, $\mathcal{E}$, $\mathcal{L}$, C) where

- $\mathcal{P}$ = ($p_1,\ldots,p_n$) is a sequence of n distinct variables called *operands*;
- $\mathcal{E}$ is an expression constructed from the operands and set operations in the usual way, such that all the operands occur in $\mathcal{E}$;
- $\mathcal{L}$ = ($L_1(y_1,\bar{z}_1),\ldots,L_n(y_n,\bar{z}_n)$) is a sequence of formulae, called *selectors*, such that for each i ($1 \le i \le n$) if w is a free variable of $L_i(y_i, \bar{z}_i)$ then $w = y_i$ or w is one of the variables in $\bar{z}_i$.
- C($\bar{x}_1,\ldots,\bar{x}_n$) is an open formula, called the *constraint* of the operation, in which the only free variables are those in $\bar{x}_1,\ldots,\bar{x}_n$
- for each i ($1 \le i \le n$), $|\bar{z}_i| = |\bar{x}_i|$

Note that to each selector we can associate an integer, identified in the fifth bullet of this definition. We will call this the *size* of the selector.

Definition 3.6: If D $\in \mathcal{D}$ then $\downarrow$D = $\emptyset$ if $\exists$ (v, p), (v, q) $\in$ D such that $p \neq q$, otherwise $\downarrow$D = D.

Definition 3.7: If Φ is a solid in m variables and L is a selector of some operation, then *an* L-*interface to* Φ is a function ϕ:$R^m \rightarrow R^k$, where k is the size of L, such that $\forall y \in R^m$, if $\Phi(y) \neq \emptyset$ then L($\Phi(y)$,$\phi(y)$) is valid. Φ is said to *expose* ϕ iff there exists a subset I of the variables of Φ, which we suppose without loss of generality to be {1,…,k}, such that for every $y \in R^m$, if $\Phi(y) \neq \emptyset$ then $y_k = \phi(y)$, where y_k denotes the first k elements of y. Each of the variables in I is said to be *required by* ϕ. Two interfaces are said to be *equivalent* if they correspond to the same selector.

Lemma 3.8: If Φ is a solid with an interface ϕ, then there exists a solid Φ' such that $\Phi' \equiv \Phi$ and Φ' exposes an interface equivalent to ϕ.

Proof: In the following, if $y \in R^{k+m}$ we denote by y_k and y_m the first k and remaining m elements of y respectively.

Suppose Φ is a solid in m variables and the selector L of ϕ is of size k. Let Φ' be the function from R^{k+m} to Σ such that for $y \in R^{k+m}$, $\Phi'(y) = \emptyset$ if $\phi(y_m) \neq y_k$ and otherwise $\Phi'(y) = \Phi(y_m)$. Then Φ' is clearly a solid in k+m variables

Suppose X = $\Phi(y)$ for some $y \in R^m$ and X $\neq \emptyset$. Let y' be the element of R^{k+m} obtained by appending $\phi(y)$ to the beginning of y. Then $\Phi'(y') = \Phi(y)$ = X. Now suppose X = $\Phi'(y)$ for some $y \in R^{k+m}$ and X $\neq \emptyset$. Since $\Phi'(y) \neq \emptyset$, $\Phi(y_m) = \Phi'(y)$ = X. Hence $\Phi' \equiv \Phi$.

Suppose ϕ is an L-interface to Φ. Let ϕ' be the function from R^{k+m} to R^k such that for $y \in R^{k+m}$, $\phi'(y) = \phi(y_m)$. If $y \in R^{k+m}$ and $\Phi'(y) \neq \emptyset$, then $\Phi'(y) = \Phi(y_m)$, so L($\Phi'(y)$,$\phi'(y)$) = L($\Phi(y_m)$,$\phi(y_m)$) is valid. Therefore ϕ' is an L-interface for Φ'. Finally, if $y \in R^{k+m}$, then according to the definition of ϕ', $\phi'(y) = \phi(y_m)$, and if $\Phi'(y) \neq \emptyset$, by the definition of Φ', $\phi(y_m) = y_k$. Hence $\phi'(y) = y_k$ proving that Φ' exposes ϕ'.□

Note that if Φ is a solid with several interfaces, by repeated applications of this lemma, we can construct a solid Ψ that exposes equivalent interfaces.

Although not strictly necessary, it would be useful for practical reasons to purge the redundant variables from a solid, while retaining those that expose interfaces we may be interested in. Hence the following definition.

Definition 3.9: If Φ is a solid which exposes each interface in some set $\mathcal{N}$ of interfaces to Φ, let Q be the set of all variables of Φ required by the interfaces, and let P be a minimal superset of Q such that P is a set of parameters for Φ, then Φ_P as defined in 3.4 is said to be *reduced with respect to* $\mathcal{N}$.

Lemma 3.10: If Φ, $\mathcal{N}$, P, Q and Φ_P are as in 3.9 then Φ_P exposes interfaces equivalent to those in $\mathcal{N}$.

The proof is straightforward and left to the reader. Note also that, as observed in Definition 3.4, Φ_P is equivalent to Φ.

Returning to our earlier example, suppose we have an ellipse defined by the function Φ in 12 variables corresponding to the quantities marked on the diagram in Figure 3 together with colour, material and temperature. If the variables a_1, c_1, x_1, *colour,* and *temperature* are required by some set of interfaces, then there is a solid Ψ in the 9 variables {a_1,

a_2, c_1, c_2, x_1, α, *material*, *colour*, *temperature*} such that Ψ is equivalent to Φ and is reduced with respect to the set of interfaces.

Suppose now that ϕ_1 and ϕ_2 are interfaces to Φ corresponding respectively to selectors L_1 and L_2, both of size 3, such that $L_1(y,\bar{z}) =$ true iff y is an ellipse and $\bar{z}$ is (u_1,v_1,d) where $d=d_1+d_2$ (see Figure 4), and $L_2(y,\bar{z}) =$ true iff y is an ellipse and $\bar{z}$ is $(x_1,y_1,colour)$. Then there is a solid in the 9 variables $\{u_1, v_1, d, x_1, y_1, colour, material, u_2, v_2\}$ which is equivalent to Φ and reduced with respect to $\{\phi_1, \phi_2\}$. Another choice for an equivalent reduced solid is one in the 10 variables $\{u_1, v_1, d, x_1, y_1, x_2, y_2, \alpha, colour, material\}$.

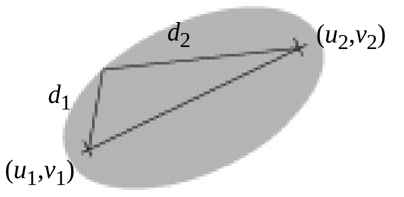

Figure 4: Another interface to an ellipse.

Definition 3.11: Let $\otimes = (P, E, L, C)$ be an n-ary operation; where $L = (L_1,\ldots,L_n)$, and for each i $(1 \le i \le n)$ let Φ_i be a solid in n_i variables, and ϕ_i be an L_i-interface to Φ_i for $\otimes$. We define a solid Ψ in $\sum_{i=1}^{n} n_i$ variables called *the application of* $\otimes$ *to* $\Phi_1,\ldots,\Phi_n$ *via* $\phi_1,\ldots,\phi_n$ as follows. If $y \in R^t$ denote by y_1 the first n_1 elements of y, denote by y_2 the next n_2 elements of y and so forth, then we define

$$\Psi(y) = \{ z \mid z \in \downarrow E(\Phi_1(y_1),\ldots,\Phi_n(y_n)) \text{ and } C(\phi_1(y_1),\ldots, \phi_n(y_n)) \text{ is valid}\}$$

Note that the set of points that results from applying the set expression to the operands may contain several copies of the same point with different property values. Such a set is not a solid. The role of the $\downarrow$ operator is to reduce the set to $\varnothing$ in such cases.

Some examples of operations for creating new solids from existing ones are as follows.

Constraining: An operation for ensuring that a rectangular solid is a cube is ({p}, p, {L}, C), where $L(x,h,w,l) =$ true iff x is a rectangular solid and h,w and l are respectively the height, width and length of the rectangular solid, and $C(x,y,z)$ is the formula $x=y \wedge x=z$.

Union: Unioning two solids is accomplished by the operation ({p,q}, p $\cup$ q, {true, true}, true).

Bonding: Bonding solids in a 2D design space as illustrated in Section 2 is defined by the operation ({p,q}, p $\cup$ q, {edge,edge}, bond) where bond$(u_1,v_1,u_2,v_2,u_3,v_3,u_4,v_4)$ is the formula $(u_1,v_1) = (u_4,v_4) \wedge (u_2,v_2) = (u_3,v_3)$ and edge(x,u_1,v_1,u_2,v_2) = true iff (u_1,v_1) and (u_2,v_2) are points in the set x of points, every point on the line between them is in x, every point to the right of this line is in x, and every point to the left of this line is not in x.

The intuition behind these definitions is that an interface to a solid delivers the information necessary to apply an operation to the solid. In the case of bonding, for example, an open edge interface defines the end points of the edge together with other characteristics of the component that must be accounted for in bonding. Note that a selector may occur more than once in an operation. For example, since bonding is a symmetric binary operation it has two identical selectors. A selector may also occur in several operations, in which case a solid with a corresponding interface may serve as an operand via that interface to any such operation. A solid may, via different interfaces, serve as more than one operand to an operation. The **cog** design in Figure 1, for example, assumes that a partial cog has two open edge interfaces and applies bonding to them.

4 Generalising e-components

In LSD as defined in [6], bonding and the associated concepts, "open edge" and "edge terminal", are built in. However, other operations are necessary, and the set of operations required may differ from one domain to another. Rather than try to come up with a comprehensive toolbox of operations, we will take the opposite approach. That is, we generalise the definitions in Section 2 to accommodate the concept of operations on solids defined in Section 3, thereby providing a basis for incorporating any required operations into the language.

The following definitions assume the existence of a design space D, a set S of solids in D, and a set O of operations in D. In view of Lemmas 3.8 and 3.10 we can assume that each solid exposes a set of interfaces and is reduced with respect to that set. The alphabet from which the various entities are constructed consists of disjoint sets of function symbols, predicate symbols, and variables. Corresponding to each selector occurring in an operation in O there is a unique function symbol called an *interface symbol*. The other function symbols are called *abstract*. Similarly, to each solid in k variables corresponds a unique k-ary predicate symbol called an *explicit symbol*, and to each operation a unique predicate symbol called a *link symbol*. The sets of explicit and link symbols are disjoint. Remaining predicate symbols are called *implicit symbols*. If X is a selector, solid or operation, we denote the corresponding symbol by $\lbrace X \rbrace$.

The definition of function cell remains as in Definition 2.1 except that only abstract symbols are used to construct a function cell. Implicit components are defined as in Definition 2.2 except that each terminal of an i-component is classified as a *simple terminal* or as an *explicit group terminal of type* L where L is a selector of some operation, and the signature of an i-component $p(v_1, \ldots, v_k)$ is a list $(F_1, \ldots, F_k)$ where for each i $(1 \le i \le k)$, $F_i = s$ if v_i is a simple terminal, and $F_i = L$ if v_i is a group terminal of type L.

Definition 4.1: An *explicit component* (*e-component*) consists of

- a literal of the form $\lbrace \Phi \rbrace(v_1, \ldots, v_k)$, called the *anchor* of the component, where Φ is a solid in k variables for some $k \ge 0$, and $v_1, \ldots, v_k$ are distinct variables; and

- for each exposed interface ϕ of Φ, one literal of the form $w = \{L\}(y_1,\ldots,y_m)$, called a *group*, where L is the selector corresponding to ϕ, $y_1,\ldots,y_m$ are distinct variables from $\{v_1,\ldots, v_k\}$, and $\{j \mid y_i = v_j$ for some $1 \le i \le m\}$ is the set of variables of Φ required for ϕ. The variable *w*, *is* called an *implicit group terminal of type* L.

Definition 4.2: An e-component is *valid* iff for some $y \in \mathrm{R}^m$, $\Phi(y)\neq\varnothing$ where Φ is the solid corresponding to the e-component and has *m* variables: otherwise the e-component is *invalid*.

Definition 4.3: A *link* consists of

- a literal, called a *knot*, of the form $\{\otimes\}(\bar{u}_1, \ldots,\bar{u}_n)$ where $\otimes$ is an *n*-ary operation and the variables in $\bar{u}_1, \ldots,\bar{u}_n$ are distinct; and
- for each i $(1 \le i \le n)$ a literal $w_i = \{L_i\}(\bar{u}_i)$ also called a *group*. For each i $(1 \le i \le n)$ w_i is called an *implicit group terminal of type* L_i.

Definition 4.4: A *component design* consists of a set of cases with no variables in common, such that the heads have the same implicit symbol and signature. A *case* is a flat clause the head of which is a literal of the same form as an implicit component, with simple terminals, link terminals and signature defined analogously. The *body* of a case is a set of function cells, components and links, satisfying the following conditions:

- No variable occurring in an e-component or link occurs anywhere else in the case, with the exception of the implicit group terminals of the component or link.
- Any variable in the case which occurs as a group terminal or implicit group terminal has exactly two occurrences which must both be of the same type. If one of these occurrences is in a component, the other must be in the head or a link, otherwise both occurrences must be in the head.

The semantics of e-components and links is analogous to that informally described at the end of Section 2. In the interests of brevity, we will give a similarly informal description of the revised semantics. The merge and deletion rules will collapse and remove groups from e-components and links that are joined by their implicit group terminals. A knot can be executed once all its groups have disappeared. This involves the following steps:

- computing the application of the operation represented by the knot to the solids represented by the associated anchors (Definition 3.11);
- ensuring this new solid exposes and is reduced with respect to the relevant interfaces, that is, those exposed by the replaced solids but not involved in the operation (Lemmas 3.8 and 3.10);
- replacing the knot and associated anchors with a new anchor corresponding to the new solid, the variables of which are those from the replaced anchors which also occur in groups or other knots.

If the new solid is invalid, assembly halts.

The definitions in this section provide the generalisation of the previous definition of LSD we seek. Whereas the original definition of e-component provided specialised open edges along which bonding can occur, we now define groups, a general purpose mechanism for providing information about the component to be used in the execution of an operation. A tooth, as in our cog example, is associated with a solid through its anchor, and defines two groups specifying the same information as the two open edges in our earlier discussion. The special purpose bond operation is now replaced with the generalised link. The link defining bonding would include a knot, which would be associated with the operation to perform the bonding of the solids as outlined in the example towards the end of Section 3, and two groups, indicating that the information provided by the e-components participating in the operation must be in the form of open edges.

5 Visual representations and environment

In the preceding sections we have described an underlying model for representing solid objects, and shown how to incorporate such solids and operations on them into LSD as e-components and links. In this section we investigate how these language constructs might be visually represented. Although our definitions of solids and operations are not limited to two or three dimensions, a practical design system is likely to be concerned with at most three dimensions, so our discussion will be similarly limited. Since solids and operations belong to the “real world” rather than the abstract world of pure Lograph entities, their representations are more complex and varied, and will need to be specified to a great extent by the user of the system. Consequently we will suggest ways in which an LSD-based environment might assist the user in this regard. Since tools for creating visualisations are very dependent on how solids and operations are implemented, we will also discuss some implementation questions.

5.1 E-Components

Since e-components are syntactic manifestations of solids, they should bear some resemblance to the solids they represent. Therefore, the appearance of an e-component is a drawing representing the set of points in space defined by the corresponding solid, Since a solid is a function, this picture depends on the values of parameters which may correspond to characteristics such as size, position, orientation in the plane or colour. However, an e-component may correspond to a solid for which some parameter values are not specified. Such an e-component is said to be *free*. In this case the appearance of the e-component is an “average” one, chosen to be representative of that family, and bears the symbol Ⓕ, as shown in the example in Figure 5. We do not have a feel-

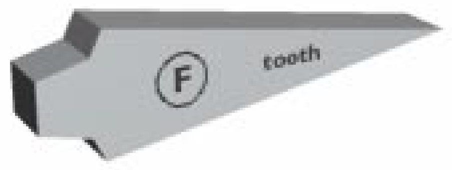

Figure 5: A free e-component tooth

ing yet for the degree to which the process of selecting parameter values for creating an average representation might be automated. An obvious possibility is to choose a size for the generated icon in relation to the other icons in the program in which the component is embedded. Clearly the more parameters a solid has and the less constrained they are, the less likely it is that an average representation could be automatically generated.

We expect that an LSD-based design environment would provide modelling tools for building solids, similar to those in CAD systems, or be capable of importing objects from and exporting objects to other systems. In addition to defining solids, such tools would also provide facilities for specifying how to compute average representations of solids. These specifications might be generated by programming, by filling in blanks in a dialogue, or by directly manipulating or annotating a picture of the solid in an appropriate editor.

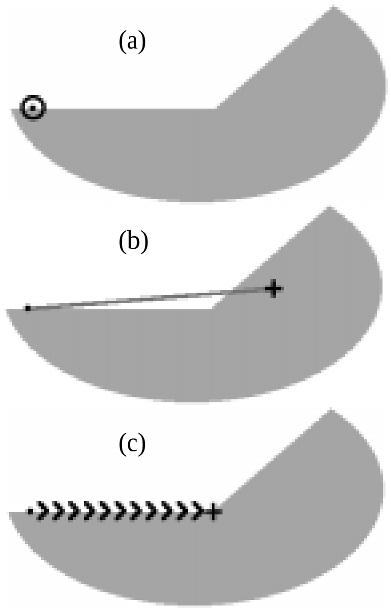

Figure 6: Editor for defining an open edge on a 2D solid.

Our model for solids as functions give just enough detail to allow us to tie it to the design language. A sound software engineering approach to implementation would be to specify an “application programming interface” for solids (SAPI), a library of routines that capture the functionality of solids as defined in previous sections. This SAPI could then be implemented in a variety of ways.

5.2 Operations

The generic definition of operations in Section 3 provides a foundation on which to base a design environment for any domain. An operation for a specific design space would be defined by a “meta-user” for delivery to the designers using such a system. This would involve specifying the functionality of the operation, its selectors, its visual representation and the visual representation of its group terminals and implicit group terminals. In addition, it would be necessary to create tools for extracting the necessary interfaces from solids to allow them to interact with the operation.

The functionality of an operation lies in its constraint. If the constraint is expressible in Horn clauses, Lograph could be used as the programming language as illustrated in [6] where a Lograph implementation of bonding is presented. This is a reasonable approach to simple operations like bonding, which merely unify some variables from groups, but more complex computations will require access to the structures that implement the solids. So a more appropriate choice might be a language oriented towards the details of that implementation, or one built on facilities provided by the solid API suggested above. Since many of the relationships specified by a constraint would be spatial, such a language might be partly visual. It is important to note, however, that LSD users would not be confronted by this language since it is for the meta-user.

Specifying a selector of an operation involves building a formula which refers to solids in a generic way, in the sense that a selector might apply to a very broad range of different solids. Again, a language relying on the proposed SAPI could be designed for this purpose, and being for the meta-user could be more technical. However, since a selector is used to extract interfaces from solids, the system used to define it must also either automatically generate a visual editor which enables the LSD user to extract an interface, or give the meta-user facilities for building such an editor.

As an example, consider the open edge selector in the example of bonding following Definition 3.11. edge(x,u_1,v_1,u_2,v_2) defined to be true iff (u_1,v_1) and (u_2,v_2) are points in the set x of points, every point on the line between them is in x, every point to the right of this line is in x, and every point to the left of this line is not in x. This formula defines the syntax of the open edge interface, and could therefore provide the specifications for a syntax directed editor for constructing open edges on a solid. Figure 6 shows how an open edge editor might work. The cursor, as it moves over the solid, changes to ⊙ whenever it is over a point satisfying the criteria for the tail of an open edge as in (a). A click on such a point fixes it and creates a “rubber band” from the point to the cursor as shown in (b). Whenever the cursor passes over a point which qualifies as the head of the open edge, the rubber band changes to a line of arrowheads as in (c), at which time a click defines the open edge, depositing the arrowheads as its visual representation.

Taking the view of selectors as syntax specifications for interface editors could be a promising approach to designing a language for them.

An LSD environment would need to provide editors for the meta-user to build visual representations for links, explicit group terminals and interfaces, which are all geometrically related. We envisage that this process would start with an interface, the visual representation of which would be required for constructing the interface editor discussed above. This would be accompanied by the design of the representation for the explicit group terminal corresponding to the interface. Once the appearance and geometry of all the interfaces related to an operation had been determined, a representation for the associated link would be designed, and for the connectors that join explicit group terminals of a particular type.

6 Concluding remarks

As a continuation of our work on applying visual programming language technology to the design of structured objects, we have generalised our earlier proposal for the design language LSD. The visual logic programming language

Lograph was chosen as the basis for LSD because logic programming represents data and operations on data homogeneously — an important consideration for a design language where the visual aspects of data are paramount.

In our original proposal, LSD was obtained by adding design objects to Lograph as "explicit components", together with one operation, bonding, for fusing these components. Although, the underlying details of components were ignored in favour of the language issues, a clean interface between the language and the design space was established.

Here we have addressed the design space side of this interface in order to accomplish several goals: to get a clearer idea of how to visually represent an e-component in an LSD program; to provide a well-defined data model for the language; and to generalise the language so that it can deal with any operations on components. To this end we have proposed a model for parameterised solids as functions from parameter values into a design space, and for operations on such solids. This model has some interesting implications for implementation. For example, although it certainly provides a basis for customisable CAD system into which any operation on solids can be incorporated, implementing such a system would require the development of some sophisticated visual editors and a visual editor generator.

7 References

[1] Autodesk Inc., *AutoLISP Release 12 Programmers Reference Manual* (1992).

[2] Bentley Systems Inc., *MicroStation 95 User's Guide*, (1995).

[3] P.T. Cox, T. Pietrzykowski, LOGRAPH: a graphical logic programming language, *Proceedings IEEE COMPINT 85*, Montreal (1985), 145-151.

[4] P.T. Cox, T. Pietrzykowski, Incorporating equality into logic programming via Surface Deduction, *Annals of Pure and Applied Logic* 31, North Holland (1986), 177-189.

[5] P.T. Cox, F.R. Giles, T. Pietrzykowski, Prograph: A step towards liberating programming from textual conditioning, *Proceedings, IEEE Workshop on Visual Languages*, Rome (1989), 150-156.

[6] P.T. Cox, T.J. Smedley, LSD: A Logic-Based Visual Language for Designing Structured Objects, *Journal of Visual Languages and Computing*, Academic Press (1998), to appear.

[7] P.T. Cox, T.J. Smedley, A Declarative Language for the Design of Structures, *Proceedings, IEEE Symposium on Visual Languages, Capri* (1997), 442-449.

[8] Graphisoft R&D Rt., *ArchiCAD 5.0: GDL Reference Manual* (1996).

[9] Pictorius Incorporated. *Prograph CPX User's Guide*. (1993).

[10] T.J. Smedley, P.T. Cox. Visual languages for the design and development of structured objects, *Journal of Visual Languages and Computing*, v8, Academic Press (1997), 57-84.