A Formal Model for Parametrised Solids in a Visual Design Language

Philip T. Cox *Trevor J. Smedley*

Faculty of Computer Science, Dalhousie University
Halifax, Nova Scotia, Canada

Abstract

Software for designing structured objects such as machinery and buildings originated from drawing or drafting, and has evolved into suites of applications for different aspects of design. Modern CAD software usually consists of sophisticated tools for graphically building multidimensional models, together with textual programming languages for dealing with tasks such as specifying parametrised constructs or complex relationships between components. This dichotomy between design with graphics and programming with text divides the users of such systems into two groups with quite different skills.

Programming languages, on the other hand, have evolved in the opposite direction, from purely textual descriptions of algorithms and data, to formalisms that rely primarily on graphics. This has led to the conjecture that the usability of design software might be improved by replacing their textual programming facilities by visual programming.

Previously we presented a preliminary proposal for a design language LSD that uses a visual representation of logic programming to provide a homogeneous view of design objects and the operations that transform them. In the current work we present a model that captures the essential features of solids and operations on solids in a design space, then generalise LSD based on this model.

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, 9] 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, users of commercial CAD systems usually purchase separate packages, to fulfill their need for parameterised components. 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 [11]. This language was obtained by extending Prograph, a general purpose visual programming language [5, 10], by adding a new picture data type, rules for combining and transforming pictures, and a construct for iteratively aggregating pictures. Even though all aspects of this language are visual, however, 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 [8] 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 [7] 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 [7] and [8] 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.

In order to address this issue, in [6] we proposed a model for parametrised solid objects and operations on such objects. The aim was to concentrate on the design-space side of the interface between language and objects, so that LSD could be generalised to account for any kind of transformation and manipulation of solids.

In the following sections we flesh out the details of the model outlined in [6]. Section 2.2 provides an informal summary of the LSD language as presented in [7], together with a brief description of Lograph and the special form of first-order predicate calculus on which it depends. In Section 3 the notions "design space", "solid" and "operation" are defined and some illustrative examples provided. The task of integrating the model for solids which these definitions provide begins in Section 4, which presents a series of constructions aimed at identifying the minimal amount of information that needs to be communicated to the design language from the design space. In Section 5 we generalise the previous definition of LSD by replacing the notions "e-component" and "bond" with abstract equivalents to solid objects and operations in a design space, via the minimal interface defined in the previous section. Section 6 discusses 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. LSD, Lograph and flat Horn clauses

As mentioned above, LSD is derived from Lograph, which in turn is a visual representation of flat Horn clauses, a particular form of first-order predicate calculus. In this section we briefly describe these concepts and relationships, both to prepare the reader for the main points we wish to make, and so that we can later show how the generalisations of LSD that follow from our model for solids are expressed in Lograph, providing a framework for the generalised LSD to which visual representations of domain-specific operations can be added. These explanations are abbreviated versions of those found in [7]. We urge the reader to consult [3], [4] and [7] for details.

2.1 Lograph and flat Horn clauses

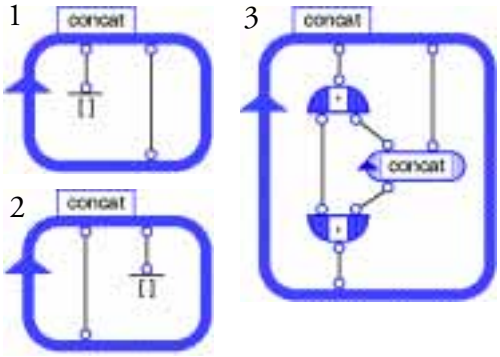


Figure 1: A Lograph definition consisting of three cases

A Lograph *program* is a set of literal definitions with no terminals in common. A *literal definition* is a set of cases with the same name, the same arity, and mutually disjoint sets of terminals. Figure 1 shows a program consisting of one literal definition made up of three cases of arity 3 named **concat**. The numbers beside the cases in this diagram are for reference, and are not part of the program. *Terminals* correspond to variable occurrences in the underlying logic and are represented by the small circles adjacent to the other icons.

A *case* consists of a name, a head and a body. The *head* of a case is a sequence of terminals of length n for some integer $n \geq 0$ called the *arity* of the case. As shown in Figure 1, the head of a case is represented by a rounded rectangle with the terminals arranged around the inside of its perimeter. A clockwise-pointing arrow on the perimeter, the *origin* of the case, indicates the starting point for the sequence of terminals. The rectangle encloses the body of the case, and has a tab at the top bearing the case name.

The *body* of a case is a set of cells, each of which is either a function cell or a literal cell. A *function cell* consists of a name, a special terminal called the *root* of the cell, and a sequence of terminals of length n for some $n \geq 0$ called the *arity* of the cell. A function cell is represented by an icon bearing the name of the cell, with one flat face and one curved face, with two possible orientations as illustrated by the function cells  and  named • in the case labelled 3 in Figure 1. The root of a function cell is located on the curved face and the other terminals on the flat face. The sequence of terminals is always read from left to right, regardless of the orientation of the icon. Cases 1 and 2 in Figure 1 illustrate an abbreviated representation for function cells of arity 0, also called *constants*; for example, the constant  .

A *literal cell* consists of a name and a sequence of terminals of length n for some $n \geq 0$, called the *arity* of the cell, for example, the cell  in case 3 of Figure 1. A literal cell is represented by an icon bearing the name of the cell, with one curved face along which are arranged the terminals of the cell. The starting point for the sequence of terminals is indicated by a clockwise-pointing arrowhead called the *origin* which may be placed anywhere on the perimeter of the cell.

A terminal may occur in several cells and the head of a case. This is indicated by lines called *wires* connecting the small circles representing occurrences of the terminal.

If we assume that a list is constructed in the usual recursive fashion from function cells named • of arity 2, and a constant named **[ ]** signifying the empty list, then the program in Figure 1 describes the concatenation of lists.

The semantics of Lograph are embodied in three execution rules, one of which involves definitions from the program. Executing a program consists of applying these rules to a *query*, which is a set of

cells, none of the terminals of which occur in the program. For example, the goal of the query in Figure 2(a) is to find the list which, when concatenated with the empty list, results in the list containing the single element **a**.

To simplify the explanation of the transformation rules, the cells to which a rule is applied are placed against a grey background, and the part of the resulting graph which is changed is indicated by an outline. The query in Figure 2(b) is obtained from that in Figure 2(a) by applying the *Replacement* rule. This rule replaces a literal cell by a copy of the body of a case of a definition having the same name and arity as the cell, and connects each terminal of the head of the case with the corresponding terminal of the replaced cell, starting at the origins of the case and the cell. By *connecting* two terminals, we mean that every occurrence of one of the terminals is replaced by a new occurrence of the other. The replacement illustrated in the figure uses case 3 of **concat**.

Figure 2(c) is obtained from Figure 2(b) by the *Merge* rule, which applies to two function cells with the same name, the same arity and the same root terminal. First corresponding terminals of the two cells are connected, then one of the cells is deleted. The third rule, *Deletion*, used to transform Figure 2(c) to Figure 2(d), removes a function cell, the root terminal of which has no other occurrences. Further application of these rules eventually transforms Figure 2(d) to the graph in Figure 2(e) which cannot be further transformed, and represents the answer to the original query.

Lograph is a visual representation of first-order predicate calculus formulae, each represented as a *flat Horn clause*. Except to say that flat Horn clauses are Horn clauses in which all nested terms have been replaced by equality literals, we will not go into further details. Instead we offer the examples in Figure 3, which illustrate the equivalence between Lograph and flat Horn clauses. The semantics of flat Horn clauses are defined by a set of deduction rules called *surface deduction* [4], the pictorial manifestations of which are the replacement, merge and deletion rules of Lograph.

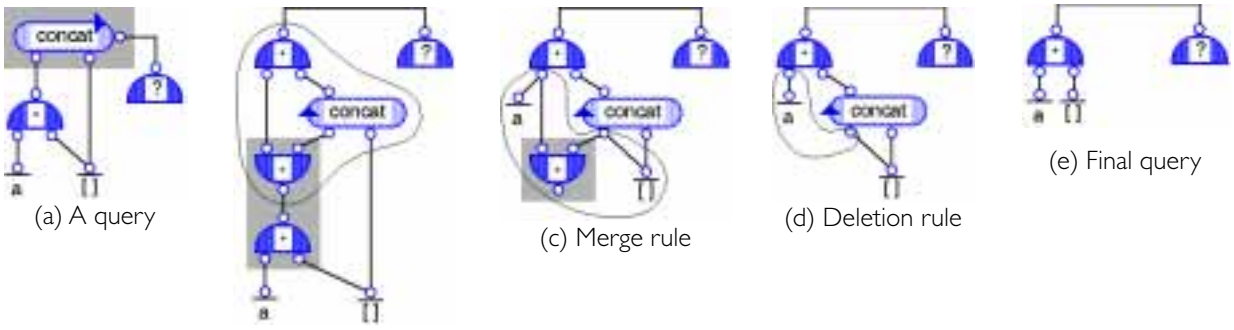
concat(*x, y, y*) :- *x* = **[ ]**.
concat(*y, x, y*) :- *x* = **[ ]**.
concat(*x, y, z*) :-
    x = •(*u, v*),
    z = •(*u, w*),
    concat(*v, y, w*).

(a)

:- **concat**(*x, y, z*),
  x = **?**.
  y = **[ ]**,
  z = •(*u, v*),
  u = **a**,
  v = **[ ]**.

(b)

Figure 3: Flat Horn clauses equivalent to (a) the definition in Figure 1; and (b) the query in Figure 2(a).



(a) A query

(b) Replacement rule

(c) Merge rule

(d) Deletion rule

(e) Final query

Figure 2: Transforming a query

2.2 LSD

First we informally describe LSD by relating it to Lograph, then introduce some formal definitions for later reference. Although LSD is derived from Lograph, it is intended for designing structures rather than general programming; hence the names of some Lograph entities have been changed to be more suggestive of their roles in the design world. When introducing the name for an LSD language construct, we will follow it by the name for the corresponding (but not necessarily identical) Lograph construct in square brackets, when it is different.

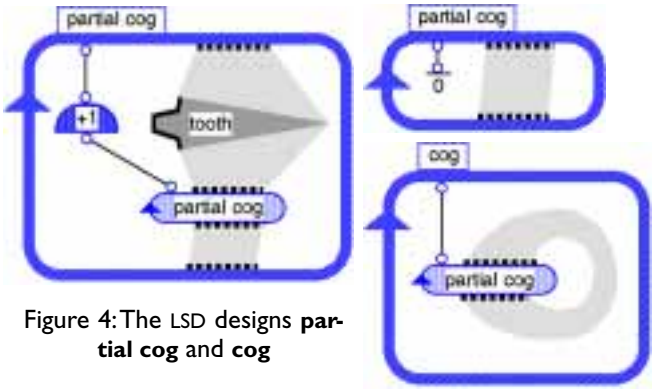


Figure 4: The LSD designs **partial cog** and **cog**

An LSD *program* is a set of *designs* [literal definitions]. Although the anatomy of a design is similar to that of a literal definition, the former defines the structure of a component while the latter defines a relation. The process of building a component according to the designs in a program is called *assembly* [execution], and is illustrated below. Figure 4 depicts a program consisting of two designs, **partial cog** and **cog**. The former defines a component, partial cog, with a given number of teeth, say *n*, by recursively describing it as the component obtained by bonding a tooth on to a partial cog with *n*-1 teeth. 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* [literal cell], or i-component for short, representing an invocation of the design **partial cog**. The function 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. We are, therefore, using integer constants as shorthand for terms built with a constant **0** and function cells **+1**. This is an extension to Lograph as defined above, but simplifies our example.

Assembly transforms a *specification* [query], a network of function cells and components, using the three Lograph rules replacement, merge, deletion, and *bonding*, which fuses e-components along open edges. Figure 5 illustrates these rules. The specification in Figure 5(b) is obtained from the one in Figure 5(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 reduce 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 con-

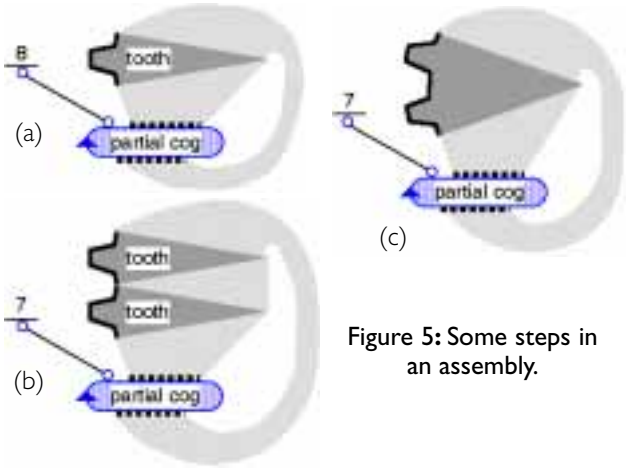


Figure 5: Some steps in an assembly.

necting two open edges. Executing this bond effects the transformation from 5(b) to 5(c), producing a new e-component, a two-toothed partial cog. Assembly stops when a specification is produced that cannot be further transformed, and hopefully consists of an e-component.

The definitions below are reproduced from [7] 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 $\mathbf{f}^0$, $\mathbf{f}^1$, $\mathbf{f}^2$, … 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 $\mathbf{f}^i(t_1, \ldots, t_i)$ by $[t_1, \ldots, t_i]$.

Definition 2.3: A *function cell* is a literal of the form $x = \mathbf{f}(y_1, \ldots, y_k)$ 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 2.4: An *implicit component* (*i-component*) is a literal of the form $\mathbf{p}(v_1, \ldots, v_k)$ 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 $\mathbf{p}(v_1, \ldots, v_k)$ is a list $(F_1, \ldots, F_k)$ where $F_i = \mathbf{s}$ if v_i is a simple terminal, and otherwise $F_i = \mathbf{e}$ for each i $(1 \leq i \leq k)$.

Definition 2.5: 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 $\mathbf{q}(v_1, \ldots, v_k)$,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 2.6: For each explicit component **e**, there exists a formula $\mathbf{K_e}$ called the *specification* of **e** such that every variable occurring in **e**, except the edge terminals, also occurs in $\mathbf{K_e}$. An e-component is *valid* iff its specification is satisfiable; otherwise it is *invalid*.

Definition 2.7: 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 2.8: 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 either in the head or in a bond, otherwise both occurrences must be in the head.

Except for the specification of an e-component, all of the above LSD concepts are defined in terms of flat Horn clauses, and are therefore directly expressible in Lograph. This is illustrated by the Lograph case in Figure 6, which corresponds to the recursive case of the design **partial cog** in Figure 4. The anonymous function cells in Figure 6 correspond to the function symbols used for "grouping" variables, introduced before Definition 2.3. In this figure, the literal cell **tooth** corresponds to the e-component **tooth**; the terminals labelled **e** correspond to the edge terminals (not implicit); the group of cells labelled **B** correspond to the internal bond between the i-component **partial cog** and the e-component **tooth**; and the groups of cells labelled **E** correspond to the open edges of the e-component **tooth**. Hence, what LSD provides from a syntactic point of view is a more concrete, domain-dependent visualisation of some abstract Lograph structures.

The above definition of internal bond is such that the merge and deletion rules accomplish most of the bonding process illustrated in the preceding example. For example, the four function cells with the line drawn around them in Figure 6 can be immediately removed by two applications of merge and two applications of deletion. However, one final step is required, necessitating a minor addition to the semantics: that is, a new e-component must be created out of the literal cells 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.

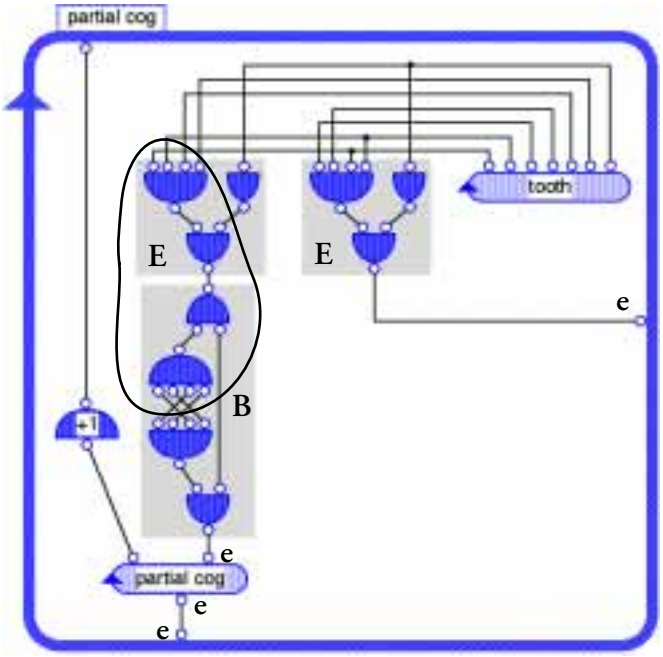


Figure 6: Lograph case corresponding to the recursive case of the design **partial cog** in Figure 4.

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 5 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.

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 $\mathbf{R}^m \times \mathbf{R}^r$. 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 $\mathcal{D}$ is a design space and n is an integer ≥ 0, a *solid in $\mathcal{D}$ in n variables* is a function Φ: $\mathbf{R}^n \rightarrow \mathcal{D}$ such that, if (v, p) and $(v, q) \in \Phi(y)$ for some $y \in \mathbf{R}^n$, 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 $\{S \mid S = \Phi(y),\ S \neq \varnothing,\ y \in \mathbf{R}^n\} = \{S \mid S = \Psi(y),\ S \neq \varnothing,\ y \in \mathbf{R}^k\}$.

Example 3.4: To illustrate these definitions, suppose we have a 2D design space with properties *colour*, *material* and *temperature*, containing an ellipse as shown in Figure 7. This ellipse, or more precisely, family of ellipses, can be characterised as a solid Φ in twelve variables consisting of the variables in the diagram together with *colour*, *material* and *temperature*. Let us suppose that *colour* is determined by *material* and *temperature*, and *temperature* is determined by *material* and *colour*. There are various other solids in fewer than 12 variables which also characterise this ellipse; for example, a solid Ψ in variables a_1, a_2, c_1, c_2, α, *material, colour*, and a solid Θ in variables a_1, a_2, x_1, y_1, α, *material, temperature*. The solids Φ, Ψ and Θ are all equivalent.

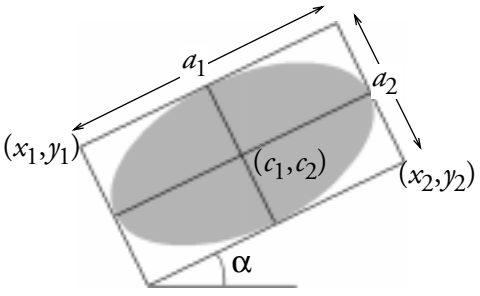


Figure 7: An overspecified solid.

Obviously there is an infinite number of solids equivalent to a given solid, since a new function can be created by introducing irrelevant variables or repeating relevant ones. However, even if we restrict the variables of a solid to be "relevant", the number of equivalent solids is still likely to be very large. We therefore need, for a particular equivalence class of solids, to determine what the relevant variables are, then to ensure that the solid we choose as the representative of the class is defined in terms of these variables. Since our aim is to manipulate solids in a design environment, we must consider what information operations will need from solids in order to do their transformations. Hence we turn our attention to defining operations that compute new solids.

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 position-

ing, 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, but not merging the objects into one. 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".

In the following, if x is a sequence of length n, and $m \leq n$, we denote by x_{m*} and x_{*m} the length m prefix and length m suffix of x respectively. If x and y are sequences, we denote the sequence that results from concatenating x and y by $x{\cdot}y$. For example, if $|x| = n$ and $|y| = m$, $\mathbf{A}(x{\cdot}y)$ is short for an expression of the form $\mathbf{A}(x_1,\ldots,x_m,y_1,\ldots,y_n)$. For consistency we extend this notation to items which are not sequences, so if x is a sequence of length n and y is not a sequence, $x{\cdot}y$ denotes $x_1,\ldots,x_n,y$, $y{\cdot}x$ denotes $y,x_1,\ldots,x_n$, and $y{\cdot}y$ denotes y,y.

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 triple ($\mathcal{F}$, $\mathcal{L}$, **C**) where

- $\mathcal{F}$ is a function from $\mathcal{D}^n$ to $\mathcal{D}$.
- $\mathcal{L} = (\mathbf{L}_1(y_1{\cdot}z_1),\ldots,\mathbf{L}_n(y_n{\cdot}z_n))$ is a sequence of formulae, called *selectors,* such that for each i $(1 \leq i \leq n)$ y_i is a variable, z_i is a sequence of variables and $\{y_i{\cdot}z_i\}$ is the set of free variables of $\mathbf{L}_i(y_i{\cdot}z_i)$. The integer $|z_i|$ is called the *size* of $\mathbf{L}_i$.
- $\mathbf{C}(x_1{\cdot}\ldots{\cdot}x_n)$ is an open formula, called the *constraint* of the operation, such that $\{x_1{\cdot}\ldots{\cdot}x_n\}$ is the set of free variables of $\mathbf{C}(x_1{\cdot}\ldots{\cdot}x_n)$, and for each i $(1 \leq i \leq n)$ x_i is a sequence of variables and $|x_i|$ is size of $\mathbf{L}_i$.

Definition 3.6: If Φ is a solid in n variables and $\mathbf{L}$ is a selector of size k of some operation, then *an* $\mathbf{L}$*-interface to* Φ is a partial function $\phi{:}\mathbf{R}^n \rightarrow \mathbf{R}^k$ such that $\forall y \in \mathbf{R}^n$, if $\Phi(y) = \varnothing$ then ϕ is undefined, otherwise $\mathbf{L}(\Phi(y),\phi(y))$ is valid, and $\forall y, z \in \mathbf{R}^n$, if $\Phi(y) = \Phi(z) \neq \varnothing$ then $\phi(y) = \phi(z)$.

Definition 3.7: If $D \in \mathcal{D}$ then $\downarrow D = \varnothing$ if $\exists$ (u, p), $(u, q) \in D$ such that $p \neq q$, otherwise $\downarrow D = D$.

Definition 3.8: Let $\otimes = (\mathcal{F}, \mathcal{L}, \mathbf{C})$ be an n-ary operation; where $\mathcal{L} = (\mathbf{L}_1,\ldots,\mathbf{L}_n)$, and for each i $(1 \leq i \leq n)$ let Φ_i be a solid in n_i variables, and ϕ_i be an $\mathbf{L}_i$-interface to Φ_i for $\otimes$. We define a solid Ψ in $t = \sum_{i=1}^{n} n_i$ variables called *the application of* $\otimes$ *to* $\Phi_1,\ldots,\Phi_n$ *via* $\phi_1,\ldots,\phi_n$ as follows. If $y \in \mathbf{R}^t$ 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

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

Note that the set of points that results from applying $\mathcal{F}$ 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.

Example 3.9: To illustrate these definitions, let $\mathcal{D}$ be a design space in 2 dimensions over 3 properties, and let **punch** = ($\mathcal{F}$, $\mathcal{L}$, **C**) be the binary operation in $\mathcal{D}$ defined as follows:

$$\mathcal{F}(E_1,E_2) = \{\, (x,y) \mid (x,y) \in E_1 \wedge \neg\exists\, x \in \mathbf{R}^3\, ((x,y) \in E_2)\,\}$$

$$\mathcal{L} = (\mathbf{L}_1, \mathbf{L}_2)$$

where $\mathbf{L}_1(a,b,c,d)$ is valid iff a is an ellipse with a focus at (c,d), and b is the shortest distance from a focus to the perimeter of a; and $\mathbf{L}_2(a,b,c,d,e,f)$ is valid iff a is an ellipse with foci at (c,d) and (e,f), and b is the shortest distance from a focus to the perimeter of a.

$$\text{and } \mathbf{C}(b_1,c_1,d_1,b_2,c_2,d_2,e_2,f_2) = [\ (c_1 = c_2 = e_2) \wedge (d_1 = d_2 = f_2) \wedge (b_1 = 2b_2)\ ]$$

This operation creates an elliptical solid with a circular hole centred on one of its foci and extending halfway to the perimeter, as shown in Figure 8. The selectors $\mathbf{L}_1$ and $\mathbf{L}_2$ ensure that the operands are ellipses, and specifies the required interfaces to these ellipses. For example, $\mathbf{L}_1$ dictates the need for an interface ϕ_1 that finds a focus and determines the shortest distance from a focus to the perimeter. The constraint $\mathbf{C}$ transforms the second ellipse into a circle, moves it into the correct position and resizes it. The function $\mathcal{F}$ defines a new solid by subtracting from the first ellipse all points which are also in the second one.

Figure 8: An elliptical solid with a hole at a focus.

Example 3.10: Some more 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 $(I, \{\mathbf{L}\}, \mathbf{C})$, where I is the identity function on $\mathcal{D}$, $\mathbf{L}(x,h,w,l)$ = **true** iff x is a rectangular solid and h,w and l are respectively the height, width and length of the rectangular solid, and $\mathbf{C}(x,y,z)$ is the formula $x{=}y \wedge x{=}z$.

Union: Creating the union of two solids is accomplished by the operation $(\mathrm{p} \cup \mathrm{q}, \{\mathbf{true}, \mathbf{true}\}, \mathbf{true})$.

Bonding: Bonding solids in a 2D design space as illustrated in Section 2.2, is defined by the operation $(\mathrm{p} \cup \mathrm{q}, \{\mathbf{edge},\mathbf{edge}\}, \mathbf{bond})$ where $\mathbf{bond}(u_1,v_1,u_2,v_2,u_3,v_3,u_4,v_4)$ is the formula $(u_1,v_1) = (u_4,v_4) \wedge (u_2,v_2) = (u_3,v_3)$ and $\mathbf{edge}(x,u_1,v_1,u_2,v_2)$ = **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 the definitions in this section 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.

In the above, we have regarded solids as independent, interacting only via operations. This ignores global properties: for example, in some design spaces it is not possible for objects to intersect. This shortcoming can be addressed, however, if we extend the notion of design space by adding to a design space $\mathcal{D}$ a predicate $\mathcal{P}$ on $2^{\mathcal{D}}$ which provides a global constraint on a set of solids in $\mathcal{D}$. For example, $\mathcal{P}$ might reject any set of solids containing two solids which share a point, and do not both have the value **yes** for the property **intersect** at that point.

4. Reducing solids

An object may be specified by many different equivalent solids, which are functions in different sets of variables. Although the information that an operation requires from a solid in order for the solid to be an operand can obviously be computed from the variables of the solid, this computation will depend on the nature of the solid. For example, bonding relies on being provided with open edges, but the exact relationship of an open edge to the solid to which it belongs will depend on whether the solid is a tooth, a stair tread or something else. Clearly, we should try to make these computations intrinsic to the solid by choosing a function the variables of which include exactly those that are required. That is, we would like all the interfaces we are interested in to be trivial functions that simply project on to some selection of their parameters. For example, none of the twelve variables of the solid Φ in Example 3.4 provide precisely the information required for the ellipse to serve as either operand of the operation defined in Example 3.9. In this section, we show how to modify a solid so that the information required by operations is directly accessible.

Definition 4.1: If Φ is a solid in n variables, $\mathbf{L}$ is a selector of size k of some operation, and ϕ is an $\mathbf{L}$-interface to Φ, Φ is said to *expose* ϕ iff there exists a sequence $p_1,\ldots,p_k$ of variables of Φ, such that for every $y \in \mathbf{R}^n$, if $\Phi(y) \neq \varnothing$ then for all i $(1 \leq i \leq k)$ $\phi(y)_i = y_{p_i}$. Each of the variables p_i is said to be *required by* ϕ. The variables $p_1,\ldots,p_k$ are not necessarily distinct, and the sequence $p_1,\ldots,p_k$ is not necessarily unique.

Lemma 4.2: Suppose Φ and Ψ are equivalent solids in n and r variables respectively, and ϕ is an $\mathbf{L}$-interface to Φ, where $\mathbf{L}$ is a selector of size k of some operation. Let ψ be a function from $\mathbf{R}^r$ to $\mathbf{R}^k$ such that for $z \in \mathbf{R}^r$, $\psi(z) = \phi(y)$, for some $y \in \mathbf{R}^n$ such that $\Psi(z) = \Phi(y)$. Then ψ is an $\mathbf{L}$-interface to Ψ. We say that ϕ and ψ are *equivalent*.

Proof: First we note that ψ is well defined since if there exists some $z \in \mathbf{R}^k$ such that $\Psi(z) = \Phi(x) = \Phi(y) \neq \varnothing$, where $x, y \in \mathbf{R}^n$ and $x \neq y$, then $\psi(z) = \phi(x) = \phi(y)$.

Now suppose that $z \in \mathbf{R}^k$ and $\Psi(z) \neq \varnothing$; then $\Psi(z) = \Phi(x)$ and $\psi(z) = \phi(x)$ for some $x \in \mathbf{R}^n$. Hence $\mathbf{L}(\Psi(z),\psi(z)) = \mathbf{L}(\Phi(y),\phi(y))$ is valid. Finally suppose $y, z \in \mathbf{R}^r$ and $\Psi(y) = \Psi(z) \neq \varnothing$; then there exist $u, v \in \mathbf{R}^n$ such that $\Psi(y) = \Phi(u)$, $\psi(y) = \phi(u)$, $\Psi(z) = \Phi(v)$ and $\psi(z) = \phi(v)$. Since $\Phi(u) = \Phi(v) \neq \varnothing$, $\phi(u) = \phi(v)$, and therefore $\psi(y) = \psi(z)$. □

Example 4.3: Let Ψ be the solid in 7 variables in Example 3.4, let $\mathbf{L}_1$ be the selector in Example 3.9, and let ψ and ρ be the functions from $\mathbf{R}^7$ to $\mathbf{R}^3$ defined as follows:

$$\psi(a_1,a_2,c_1,c_2,\alpha,colour,material) = (r,\ u_1,\ v_1)$$

$$\rho(a_1,a_2,c_1,c_2,\alpha,colour,material) = (r,\ u_2,\ v_2)$$

$$\text{where } r = \frac{a_1 - \sqrt{a_1^2 - a_2^2}}{2}$$

$$u_1 = c_1 - \cos\alpha\frac{\sqrt{a_1^2 - a_2^2}}{2}$$

$$v_1 = c_2 - \sin\alpha\frac{\sqrt{a_1^2 - a_2^2}}{2}$$

$$u_2 = c_1 + \cos\alpha\frac{\sqrt{a_1^2 - a_2^2}}{2}$$

$$\text{and } v_2 = c_2 + \sin\alpha\frac{\sqrt{a_1^2 - a_2^2}}{2}$$

then ϕ and ψ are both $\mathbf{L}_1$-interfaces to Ψ, neither of which are exposed by Ψ.

Let θ and τ be the functions from $\mathbf{R}^7$ to $\mathbf{R}^3$ defined as follows:

$\theta(a_1,a_2,x_1,y_1,\alpha,material,temperature) = (r,\ u_1,\ v_1)$

$\tau(a_1,a_2,x_1,y_1,\alpha,material,temperature) = (r,\ u_2,\ v_2)$

where r, u_1, v_1, u_2 and v_2 are as defined as above, $c_1 = x_1+(a_1\cos\alpha + a_2\sin\alpha)/2$ and $c_2 = y_1+(a_1\sin\alpha - a_2\cos\alpha)/2$: then θ and τ are the $\mathbf{L}_1$-interfaces to the solid Θ in Example 3.4 equivalent to ψ and ρ respectively.

Lemma 4.4: If Φ is a solid with an interface ϕ, then there exists a solid Φ' such that $\Phi' \equiv \Phi$ and Φ' exposes the interface equivalent to ϕ.

Proof: Suppose Φ is a solid in n variables and ϕ is an $\mathbf{L}$-interface of size k. Let Φ' be the function from $\mathbf{R}^{k+n}$ to $\mathcal{D}$ such that for $y \in \mathbf{R}^{k+n}$, $\Phi'(y) = \varnothing$ if $\phi(y_{*_n}) \neq y_{k^*}$ and otherwise $\Phi'(y) = \Phi(y_{*_n})$. Then Φ' is clearly a solid in $k+n$ variables

Suppose $y \in \mathbf{R}^n$ and $\Phi(y) \neq \varnothing$, then $\Phi'(\phi(y)\bullet y) = \Phi(y)$. Now suppose $y \in \mathbf{R}^{k+n}$ and $\Phi'(y) \neq \varnothing$, then $\Phi(y_{*_n}) = \Phi'(y)$. Hence $\Phi' \equiv \Phi$.

Let ϕ' be the interface to Φ' equivalent to ϕ. If $y \in \mathbf{R}^{k+n}$, then $\phi'(y) = \phi(y_{*_n})$, and if $\Phi'(y) \neq \varnothing$, by the definition of Φ', $\phi(y_{*_n}) = y_{k^*}$. Hence $\phi'(y) = y_{k^*}$ showing 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.

Example 4.5: Applying this construction to the solid Θ in 7 variables from Example 3.4 and the $\mathbf{L}_1$-interface θ from Example 4.3, we obtain a solid Θ_1 in the 10 variables r, u_1, v_1, a_1, a_2, x_1, y_1, α, *material* and *temperature* with the property that $\Theta_1(r,u_1,v_1,a_1,a_2,x_1,y_1,\alpha,material,temperature) = \varnothing$ unless r, u_1 and v_1 are related to a_1, a_2, x_1, y_1 and α as in Example 4.3. The $\mathbf{L}_1$-interfaces θ_1 and τ_1 to Θ_1 equivalent to θ and τ are

$\theta_1(r,u_1,v_1,a_1,a_2,x_1,y_1,\alpha,material,temperature) = (r,u_1,v_1)$

and $\tau_1(r,u_1,v_1,a_1,a_2,x_1,y_1,\alpha,material,temperature) = (r,u_2,v_2)$

the first of which is exposed by Θ_1. Applying the construction again to Θ_1 and τ_1, produces a solid Θ_2 in 12 variables such that $\Theta_2(r,u_2,v_2,r,u_1,v_1,a_1,a_2,x_1,y_1,\alpha,material,temperature) = \varnothing$ unless r, u_1, u_2, v_1

and v_2 are related to a_1, a_2, x_1, y_1 and α as in Example 4.3. The $\mathbf{L}_1$-interfaces θ_2 and τ_2 to Θ_2 equivalent to θ_1 and τ_1 are

$$\theta_2(r,u_2,v_2,r,u_1,v_1,a_1,a_2,x_1,y_1,\alpha,\mathit{material},\mathit{temperature}) = (r,u_1,v_1)$$

$$\text{and } \tau_2(r,u_2,v_2,r,u_1,v_1,a_1,a_2,x_1,y_1,\alpha,\mathit{material},\mathit{temperature}) = (r,u_2,v_2)$$

both of which are exposed by Θ_2.

The construction of Lemma 4.4 introduces new variables to expose each interface rather that relying on variables already present, and therefore is likely to produce a solid with repeated variables. We now show how to reduce a solid to an equivalent one that exposes the interfaces we are interested in, but has a minimal set of variables.

It is useful to make two observations about solids which are equivalent in a trivial sense, and interfaces to them. First, if Φ is a solid in n variables and π is a permutation of $\{1,\ldots,n\}$ then the function Ψ defined by $\Psi(y) = \Phi(y_{\pi^{-1}(1)},\ldots,y_{\pi^{-1}(n)})$ for all $y \in \mathbf{R}^n$ is a solid equivalent to Φ. Second, if ϕ is an $\mathbf{L}$-interface to Φ, the function ψ defined by $\psi(y) = \phi(y_{\pi^{-1}(1)},\ldots,y_{\pi^{-1}(n)})$ for all $y \in \mathbf{R}^n$ is the $\mathbf{L}$-interface to Ψ equivalent to ϕ, and is exposed by Ψ iff ϕ is exposed by Φ. Hence the variables of a solid can be permuted arbitrarily without disturbing its properties or those of its interfaces.

Definition 4.6: If Φ be a solid and i and j are variables of Φ, then i and j are said to be *equivalent* iff for all $y \in \mathbf{R}^n$, $y_i \neq y_j$ implies $\Phi(y) = \varnothing$.

Lemma 4.7: If Φ be a solid which exposes an interface ψ, then there exists a solid Ψ which is equivalent to Φ, exposes the interface ψ equivalent to ϕ, and has no distinct variables that are equivalent.

Proof: Either Φ has no distinct equivalent variables, and there is nothing to prove, or two variables of Φ are equivalent. Suppose Φ that has n variables. Because of the observation preceding Definition 4.6, we can assume without loss of generality that n-1 and n are equivalent. Define Φ' to be the function $\mathbf{R}^{n-1} \to \mathcal{D}$ such that $\Phi'(y) = \Phi(y_1,\ldots,y_{n-1},y_{n-1})$ for all $y \in \mathbf{R}^{n-1}$. Then Φ' is clearly a solid in n-1 variables.

To show that Φ' is equivalent to Φ we first note that $\{S \mid S = \Phi'(y),\ S \neq \varnothing,\ y \in \mathbf{R}^{n-1}\} = \{S \mid S = \Phi(y_1,\ldots,y_{n-1},y_{n-1}),\ S \neq \varnothing,\ y \in \mathbf{R}^{n-1}\} \subseteq \{S \mid S = \Phi(y),\ S \neq \varnothing,\ y \in \mathbf{R}^n\}$. Now suppose $\Phi(y) \neq \varnothing$ for some $y \in \mathbf{R}^n$, then $y_{n-1} = y_n$ so $\Phi(y) = \Phi(y_1,\ldots,y_{n-1},y_n) = \Phi(y_1,\ldots,y_{n-1},y_{n-1}) = \Phi'(y_1,\ldots,y_{n-1},y_{n-1})$. Hence $\Phi \equiv \Phi'$.

Suppose ϕ is an $\mathbf{L}$-interface to Φ of size k, and denote by ϕ' the interface to Φ' equivalent to Φ; then $\forall y \in \mathbf{R}^{n-1}$, $\phi'(y) = \phi(y_1,\ldots,y_{n-1},y_{n-1})$. Let $p_1,\ldots,p_k$ be a sequence of variables of Φ required by ϕ, and let $q_1,\ldots,q_k$ be the sequence of integers obtained by replacing any occurrence of n in $p_1,\ldots,p_k$ by n-1. If $y \in \mathbf{R}^{n-1}$, let $y' = y_1,\ldots,y_{n-1},y_{n-1}$, then $\phi'(y) = \phi(y')$ so for each i $(1 \le i \le k)$, $\phi'(y)_i = \phi(y')_i = y'_{p_i} = y_{q_i}$. Therefore Φ' exposes ϕ'.

Each application of this construction produces a solid in fewer variables, so repeating it must therefore eventually result in a solid Ψ and interface ψ with the required properties. □

Example 4.8: Applying this construction to the solid Θ_2 in 12 variables in Example 4.5 results in the solid Θ_3 in 11 variables such that $\Theta_3(r,u_2,v_2,u_1,v_1,a_1,a_2,x_1,y_1,\alpha,material,temperature) = \varnothing$ unless r, u_1, u_2, v_1 and v_2 are related to a_1, a_2, x_1, y_1 and α as in Example 4.3. The $\mathbf{L}_1$-interfaces θ_3 and τ_3 to Θ_3 equivalent to θ_2 and τ_2 are

$$\theta_3(r,u_2,v_2,u_1,v_1,a_1,a_2,x_1,y_1,\alpha,material,temperature) = (r,u_1,v_1)$$

$$\text{and } \tau_3(r,u_2,v_2,u_1,v_1,a_1,a_2,x_1,y_1,\alpha,material,temperature) = (r,u_2,v_2)$$

By applying the constructions described above we obtain a solid which exposes all the interfaces we are interested in, and has no pairs of equivalent variables. However, there may still be variables which are dependent on others but not required by any interface that we are interested in. This is the case in Example 4.8.

Definition 4.9: If Φ be a solid in n variables and $\mathcal{N}$ is a set of interfaces to Φ, a subset P of its variables is said to be *sufficient for* Φ *with* $\mathcal{N}$ iff P includes all the variables required by the interfaces in $\mathcal{N}$, and for all $y, z \in \mathbf{R}^n$, if $y_i = z_i$ for all $i \in$ P, then $\Phi(y) = \Phi(z)$. P is called a *parameter set for* Φ *with* $\mathcal{N}$ iff P is sufficient for Φ with $\mathcal{N}$, P contains no distinct equivalent variables, and no proper subset of P is sufficient for Φ.

Lemma 4.10: If Φ is a solid and $\mathcal{N}$ is a set of interfaces to Φ, then there exists a solid Ψ equivalent to Φ such that the set of variables of Ψ is a parameter set of Ψ with $\mathcal{M}$, where $\mathcal{M}$ is the set of interfaces to Ψ equivalent to those in $\mathcal{N}$. Ψ is said to be *reduced* with respect to $\mathcal{M}$.

Proof: We can assume that the set of variables of Φ includes all the variables required by the interfaces in $\mathcal{N}$ (by lemma 4.4) and contains no distinct equivalent variables (by lemma 4.7). Either the set of variables of Φ is a parameter set for Φ with $\mathcal{N}$, in which case $\Psi = \Phi$; or there exists a proper subset of the variables of Φ that is sufficient for Φ with $\mathcal{N}$. Let P be such a subset with the added property that it has no proper subset sufficient for Φ.

Suppose Φ has n variables. According to the observation preceding Definition 4.6 we can assume without loss of generality, that P = $\{1,\ldots,p\}$ where $1 \le p < n$. Let Ψ be the function $\mathbf{R}^p \to \mathcal{D}$ such that for all $x \in \mathbf{R}^p$, $\Psi(x) = \Phi(y)$ for some $y \in \mathbf{R}^n$ such that $y_{p^*} = x$; then Ψ is well defined since P is sufficient for Φ with $\mathcal{N}$, and Ψ is clearly a solid in p variables equivalent to Φ.

Let $\phi \in \mathcal{N}$ be an $\mathbf{L}$-interface to Φ of size k, which is clearly less than or equal to p. According to the observation preceding Definition 4.6 we can assume, again without loss of generality, that the variables required by ϕ are $\{1,\ldots,k\}$. Let ϕ_P be the interface to Ψ equivalent to Φ, then for all $y \in \mathbf{R}^p$, $\phi_P(y) = \Psi(y) = y_{k^*}$, so Ψ exposes ϕ_P.

Let $\mathcal{M}$ be the set of interfaces to Ψ equivalent to those in $\mathcal{N}$, then Ψ is a solid equivalent to Φ, and the set of variables of Ψ is sufficient for Ψ with $\mathcal{M}$. □

Example 4.11: The variables r, u_2, v_2, u_1, v_1, *material* and *temperature* constitute a parameter set for Θ_3 with $\{\theta_3, \tau_3\}$. Therefore the solid Θ_4 is reduced with respect to $\{\theta_4, \tau_4\}$, Θ_4 is equivalent to Θ_3 and θ_4 and τ_4 are equivalent to θ_3 and τ_3 respectively, where:

$$\Theta_4(r,u_2,v_2,u_1,v_1,material,temperature) = \Theta_3(r,u_2,v_2,u_1,v_1,z_1,\ldots,z_5,material,temperature)$$
$$\text{for some } z \in \mathbf{R}^5$$

$$\theta_4(r,u_2,v_2,u_1,v_1,material,temperature) = (r,u_1,v_1)$$

$$\text{and } \tau_4(r,u_2,v_2,u_1,v_1,material,temperature) = (r,u_2,v_2)$$

5. Generalising LSD

In LSD as defined in [7], bonding and the associated concepts, "open edge" and "edge terminal", are built in. Other operations are necessary, however, and the set of required operations 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 2.2 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 $\mathcal{D}$, a set $\mathcal{S}$ of solids in $\mathcal{D}$, and a set $\mathcal{O}$ of operations in $\mathcal{D}$. In view of Lemma 4.10 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 $\mathcal{O}$ there is a unique function symbol with arity equal to the size of the selector, 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 n-ary operation a unique n-ary 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 $\{\mathrm{X}\}$.

The definition of function cell remains as in Definition 2.3 except that only abstract symbols are used to construct a function cell. Implicit components are defined as in Definition 2.4 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 $\mathbf{p}(v_1, \ldots, v_k)$ is a list $(\mathrm{F}_1, \ldots, \mathrm{F}_k)$ where for each i $(1 \le i \le k)$, $\mathrm{F}_i = \mathbf{s}$ if v_i is a simple terminal, and $\mathrm{F}_i = \mathbf{L}$ if v_i is a group terminal of type **L**.

Definition 5.1: An *explicit component* (*e-component*) consists of

- a literal of the form $\{\Phi\}(v_1, \ldots, v_n)$, called the *anchor* of the component, where Φ is a solid in n variables for some $n \ge 0$, and $v_1, \ldots, v_n$ are distinct variables; and
- for each exposed interface ϕ of Φ, one literal of the form $w = \{\mathbf{L}\}(v_{i_1},\ldots,v_{i_k})$, called a *group*, where **L** is the selector corresponding to ϕ, k is the size of **L**, and $i_1,\ldots,i_k$ is the sequence of variables of Φ required by ϕ. The variable w, *is* called an *implicit group terminal of type* **L**.

Definition 5.2: An e-component is *valid* iff for some $y \in \mathbf{R}^n$, $\Phi(y) \neq \varnothing$ where Φ is the solid corresponding to the e-component and has n variables: otherwise the e-component is *invalid*.

Definition 5.3: A *link* consists of

- a literal, called a *knot*, of the form $\{\otimes\}(u_1 \cdot \ldots \cdot u_m)$ where $\otimes$ is an m-ary operation, and for each i ($1 \le i \le m$) u_i is a sequence of variables of length equal to the size of the i^{th} selector of $\otimes$, and the variables in $u_1, \ldots, u_m$ are distinct; and
- for each i ($1 \le i \le m$) a literal $w_i = \{\mathbf{L}_i\}(u_i)$ also called a *group*. For each i ($1 \le i \le m$) w_i is called an *implicit group terminal of type* $\mathbf{L}_i$.

Definition 5.4: 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.

Example 5.5: Figure 9 depicts a portion of an LSD program in which the operation **punch** from Example 3.9 is applied twice. In the diagram, ellipse $\mathbf{E}_1$ is the first operand to both applications of **punch**. The **punch** operation is represented by a dark grey funnel-like connector terminating on the focus of the first operand to which the operation is to be applied.

Figure 10 illustrates the underlying Lograph structures of which the objects in Figure 9 are a concrete visual representation. The collections of cells labelled **A** are the *links* corresponding to the two **punch** operations in Figure 9. In each of these links, the **punch** literal cell is the *knot* of the link, and the $\mathbf{L}_1$ and $\mathbf{L}_2$ function cells are the *groups*. Note that, since Definition 5.3 does not dictate an ordering of the variables of a knot, we have chosen an ordering that simplifies the diagram by avoiding crossed wires.

The collections of cells labelled $\mathbf{E}_1$, $\mathbf{E}_2$ and $\mathbf{E}_3$ are the *e-components* corresponding to the ellipses. The *anchor* of the e-component $\mathbf{E}_1$ is the literal cell Θ_4, and represents the solid Θ_4 from Example 4.11. The $\mathbf{L}_1$ function cells are the *groups* of this e-component, and correspond to the $\mathbf{L}_1$-interfaces θ_4 and τ_4 in Example 4.11. Recall that Θ_4 is reduced with respect to $\{\theta_4,\tau_4\}$. In each of the e-components $\mathbf{E}_1$ and $\mathbf{E}_2$, the group $\mathbf{L}_2$ corresponds to the interface η defined by

$$\eta(r,u_2,v_2,u_1,v_1,material,temperature) = (r,u_2,v_2,u_1,v_1)$$

Clearly Θ_4 is also reduced with respect to $\{\eta\}$, so we can use Θ_4 for the anchor of ellipses $\mathbf{E}_1$ and $\mathbf{E}_2$. The two unconnected terminals on each of the Θ_4 literal cells correspond to the variables *material* and temperature. According to Definition 5.1, an e-component should contain one group for each exposed interface. In Figure 10, therefore, each of the ellipse groups should contain an $\mathbf{L}_2$ function cell and two $\mathbf{L}_1$ cells. We have omitted those that are not required for the two **punch** operations. Note, however, that if these cells were present their roots would not be attached to anything, so the deletion rule would remove them.

The semantics of e-components and links is analogous to that informally described at the end of Section 2.2. 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 3.8);
- 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 (Lemma 4.10);
- 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.

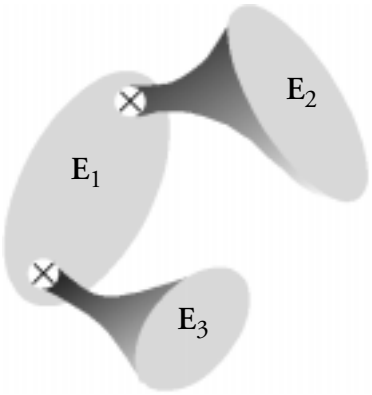


Figure 9: Applying an operation to some solids in LSD

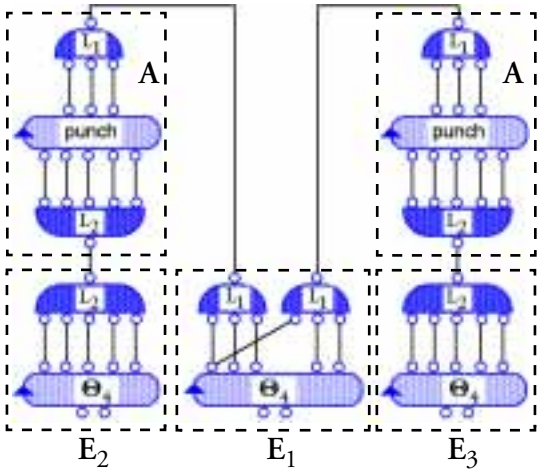


Figure 10: The Lograph underlying the LSD of Figure 9

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.

6. 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.

6.1 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 11. We do not have a feeling 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.

Figure 11: A free e-component **tooth**

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.

Our model for solids as functions gives 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.

6.2 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 [7] 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 of how such an interface-extracting editor might work, consider the open edge selector in the example of bonding in Example 3.10. **edge**$(x,u_1,v_1,u_2,v_2,)$ is 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 12 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.

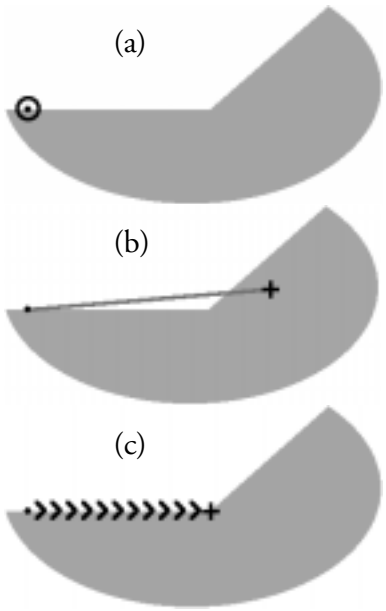


Figure 12: Editor for defining an open edge on a 2D solid.

Taking the view of selectors as syntax specifications for interface editors could be a promising approach to designing a language for them. However, it is important to note that some aspects of the process of extracting interfaces will depend on the solid. For example, in order to be able to distinguish between the two open edges of the e-component **tooth**, let us label them as "upper" and "lower" when the tooth is horizontal with its apex pointing right and its finished end, the actual tooth, pointing left. The tools for identifying an open edge must allow the user to ensure that the interface which is extracted does not correspond to "upper" for some parameter values and to "lower" for others, a possibility which cannot be ruled out by a generic definition such as 3.5 and 3.6.

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.

7. 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 systems 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.

8. References

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

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

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

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

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

[6] P.T. Cox & T.J. Smedley (1998) A Model for Object Representation and Manipulation in a Visual Design Language. *Proceedings of the 1998 IEEE Symposium on Visual Languages*, Halifax, IEEE Computer Society Press, 254-261.

[7] P.T. Cox & T.J. Smedley (1998) LSD: A Logic-Based Visual Language for Designing Structured Objects, *Journal of Visual Languages and Computing* v9, Academic Press, 509-534.

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

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

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

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