LSD: A Logic-Based Visual Language for Designing Structured Objects

Philip T. Cox *Trevor J. Smedley*

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

Abstract

Designing structured objects such as mechanical devices, buildings and electronic devices, involves two disparate activities. Visual representations of objects are drawn in two or three dimensions. Also, since the structure of more complex artifacts may depend on parameter values, coding akin to programming may be required. Consequently, design tools such as CAD systems usually provide sophisticated graphical drafting packages plus interfaces to textual programming languages. Those dealing with highly parameterised designs, such as electronic circuits, may abandon graphics altogether, expressing everything in textual terms.

Based on the observation that logic programming gives a uniform view of data and the algorithms that operate on it, we present a preliminary proposal for a declarative, logic-based, visual language for structured design (LSD). The advantage of this approach is seen to be the natural integration of concrete representations of design components with operations that specify how components are assembled from other components.

1. Introduction

The process of designing and building computer software provides a general model for a large class of processes which aim to produce artifacts with some required form and function. Some other examples of this class are the design of digital circuits [13, 27, 28], building structures [23], and parameterised devices [19]. The analogy with computer programming is stronger with some domains than others, but generally the process involves building a design for some object using some descriptive language: testing the design by simulating its behaviour, which should be a consequence of the design process, and finally using the design to generate an artifact which exhibits the same behaviour as the design under simulation. These activities correspond to programming, testing using a debugging interpreter and compiling to stand-alone code. In VLSI design, the steps are coding the design of an electronic device, testing it via simulation, and compiling low-level chip fabrication code.

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, 3, 14] 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 packages 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.

In contrast to tools for industrial design, tools for software design have always been dominated by textual means of expression. It is only recently that visual languages have been developed that combine the symbolic manipulation capabilities of textual programming languages with the expressiveness and ease of use of visual tools. It would appear then, that visual languages 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 [25]. This language was obtained by extending Prograph, a general purpose visual programming language [10, 20], by adding a new picture data type, rules for combining and transforming pictures, and a construct for iteratively aggregating pictures. Programs for building structured objects are represented using Prograph's standard dataflow diagrams, which provide a concrete visualisation of algorithm structure, while the objects themselves are concretely represented as pictures. However, even though all aspects of this language are visual, the visualisations of algorithms and objects do not mix. When viewing the algorithms, the objects are not visible, and *vice versa.* The only one exception to this strict separation is a new graphical rewrite rule in which productions that show how an object can be transformed are directly expressed in terms of the pictorial representation of the object.

In imagining how an object is assembled out of component objects, one is more likely to mentally visualise the components themselves and the relationships between them, rather than to picture the structure of an assembly algorithm, divorced from the components it operates on. It seems, therefore, that the construction of an object from components would be more intuitively conveyed by a language where components and operations are homogeneously represented. The sharp division between algorithm and data in the language described in [25] 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 [6] the visual logic programming language, Lograph, is described. Applications of Lograph to querying databases and solving graph problems are explored in [5] and [8]. Like Prolog and other logic programming languages, Lograph exposes the structure of data in programs: however, it goes further than such languages in two important respects. First, in Lograph the unification process, central to the execution of logic programs, is divided into individual steps, each performing an atomic transformation of some terms. Second, its semantics is defined as graphical transformation rules.

Since Lograph combines data and algorithms in a homogeneous visual representation manipulated by graphical transformation rules, it could provide a sound basis for a language for designing structured objects. This possibility, initially suggested in [11] is further explored here.

In Section 2 we give a brief introduction to Lograph, concentrating on those aspects that are relevant to the concepts that follow in Section 3 where we informally present LSD, a language for the design of structured objects. Section 4 provides the formal underpinnings for the language. In Section 5 we survey further issues that need to be addressed, and summarise results in Section 6. The presentation assumes some familiarity with first-order logic, logic programming and Prolog.

2. Introduction to Lograph

The use of pictorial representations for predicate logic formulae was originally investigated by Peirce [21], and more recently further developed by Sowa as a means for elucidating general conceptual patterns occurring in complex formulae [26]. Intense interest in logic programming in the 80s coupled with the advent of visual programming, naturally led researchers to devise various visual logic programming languages [6, 17, 22]. One of these, Lograph, provides the basis for our proposed design language [6]. In this section we give a brief introduction to the syntax and semantics of Lograph, restricting our attention to those aspects on which LSD depends.

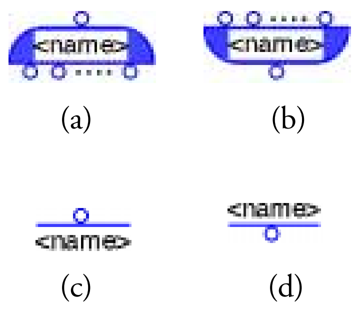


Figure 1: Representations of function cells

In the following, a *name* is a member of an arbitrary but fixed set of strings.

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 shown in Figure 1 (a) and (b). The terminals are represented by small circles on the faces, the root 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. Figure 1 (c) and (d) show abbreviated representations for function cells of arity 0, also called *constants*.

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. Figure 2 shows how 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 on the perimeter of the cell called the *origin* of the cell.

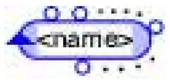


Figure 2: Representation of a literal cell

Note that the representation of a given cell may vary in length, the spacing of terminals, and in the case of literal cells, the location of the origin on the perimeter.

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. The *body* of a case is a set of cells, each of which is either a function cell or a literal cell. As shown in Figure 3, 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 name of the case. A terminal may occur several times in a case, and consequently may appear several times in the visual representation. Such multiple occurrences are indicated by lines called *wires* connecting the different occurrences. For example, in Figure 3, the first terminal of the head of the case occurs also as the second terminal of the literal cell **lit 2**, the last terminal of the literal cell **lit 1**, the second terminal of the function cell **f1** and both the second terminal and the root terminal of the function cell **f2**.

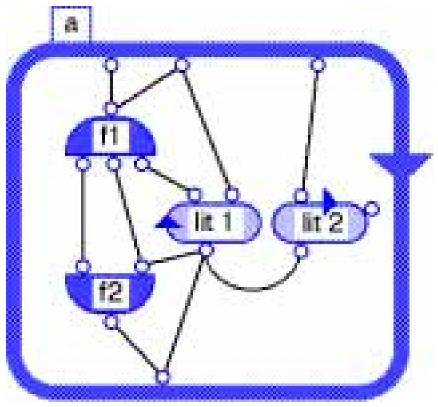


Figure 3: Visual representation of a case

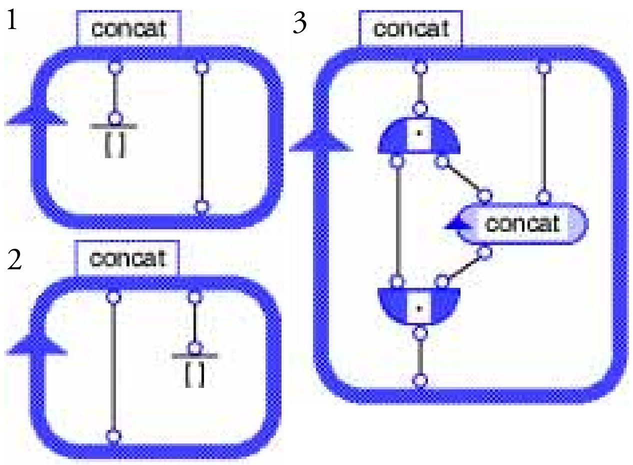


Figure 4: A definition consisting of three cases

The representation of a particular case may vary in the position of the origin, the spacing of the head literals, the size of the head rectangle, and the style, number and shape of the wires which connect the different representations of a terminal.

A *literal definition is* a set of cases with the same name, the same arity, and mutually disjoint sets of terminals. A *program* is a set of definitions with no terminals in common. Figure 4 shows a program consisting of one definition made up of three cases. The numbers beside the cases in this diagram are for reference, and are not part of the program.

To relate Lograph to textual logic programming, we note that a case corresponds to a Horn clause, the head and body of a case to the head and body of a clause, and so forth. Not surprisingly, there is a textual representation for Lograph programs. We will discuss this representation at the end of this section.

The semantics of the language are embodied in three execution rules, one of which involves definitions of 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. We will describe the execution rules with the aid of an example in which the query in Figure 5 is transformed using the program in Figure 4 which describes the concatenation of lists. This program assumes that a list is represented in the usual recursive fashion with function cells named • of arity 2, and a 0 arity function cell named **[ ]** to signify the empty list. For example, in Figure 5 the structure attached to the second terminal of the right-hand **concat** cell represents the list (1 2), and the second terminal of the other **concat** cell is attached to a structure representing a list containing one variable, which, since it is not connected to anything else, corresponds to an anonymous variable in a textual logic programming language. The purpose of the function cell named **?** will become clear later.

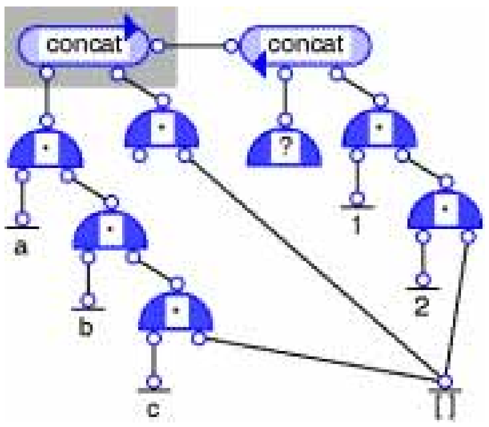


Figure 5: A query

In order to simplify the presentation of the example, in Figure 5 and following figures, the cells selected for transformation by application of a rule are placed against a shaded background. In Figure 6 and following figures, the part of the graph affected by the most recent transformation has an outline drawn around it.

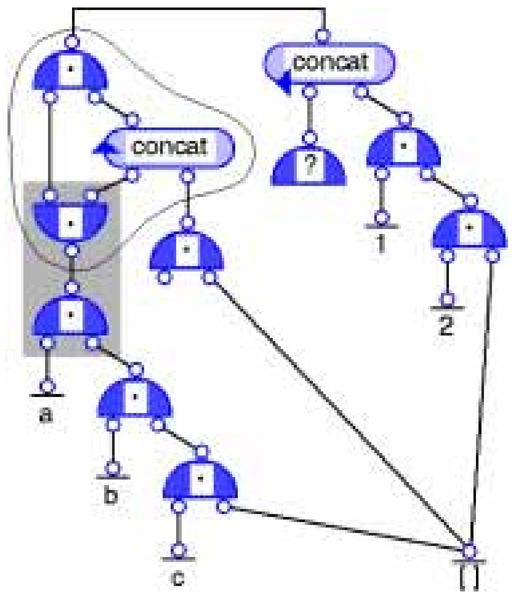


Figure 6: Result of Replacement rule

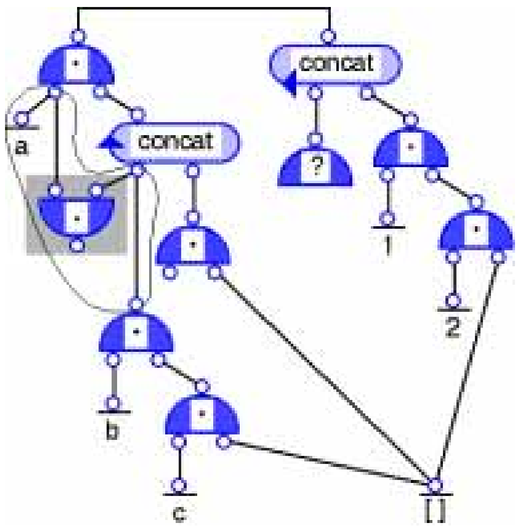


Figure 7: Result of Merge rule

In the first step of the execution, the left-hand **concat** cell of the query is transformed by an application of *Replacement*. This rule replaces a cell by a copy of the body of a case having the same name and arity as the replaced cell, and connects each terminal of the head of the case with the corresponding terminal of the cell, starting at the origins of both. By *connecting* two terminals, we mean that every occurrence of one of the terminals is replaced by a new occurrence of the other. The application of replacement that transforms the query in Figure 5 to the query in Figure 6 uses the case labelled 3 in Figure 4.

The next step is an application of the *Merge* rule, which applies 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, as illustrated by the transformation from Figure 6 to Figure 7.

The third rule, *Deletion*, used to transform the query in Figure 7 to that in Figure 8, removes a function cell, the root terminal of which has no other occurrences. Further application of replacement, merge and deletion transforms the query in Figure 8 to that in Figure 9.

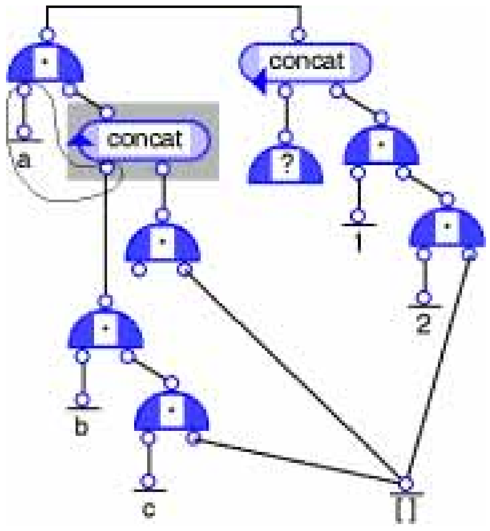


Figure 8: Result of Deletion rule

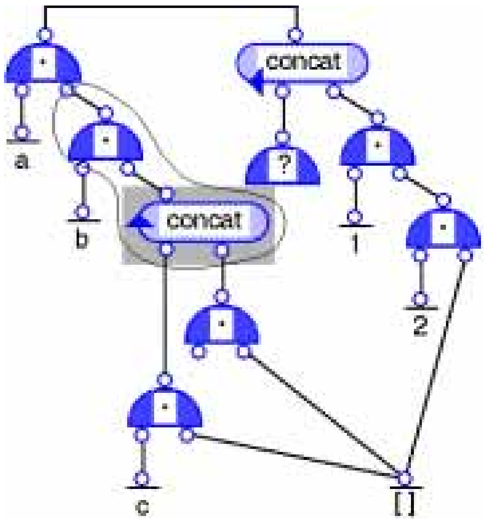


Figure 9: Result of Replacement with case 3, Merge and Deletion

Applying the replacement rule with case 1 to the indicated literal cell in Figure 9 results in the query shown in Figure 10. The reader is invited to verify that the query in Figure 11 is obtained from that in Figure 10 by an applications of merge to the indicated cells followed by two deletions.

Beginning with an application of replacement with case 3 to the **concat** cell, the query in Figure 11 is eventually transformed into the query in Figure 12 to which no further rules apply. The role of the function cell **?** is to prevent the deletion rule from destroying the “answer” to the query, which is a structure representing the list (a b 1 2).

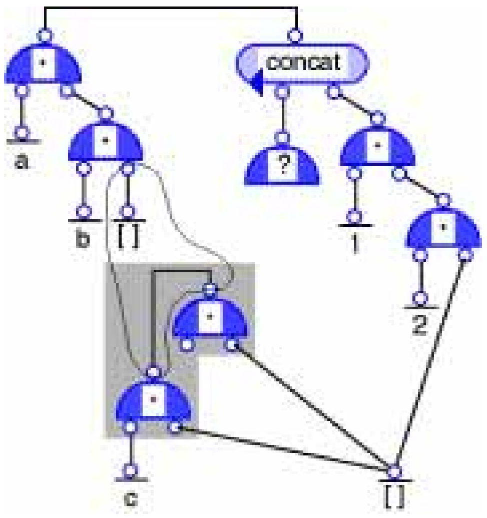


Figure 10: Result of Replacement with case 1

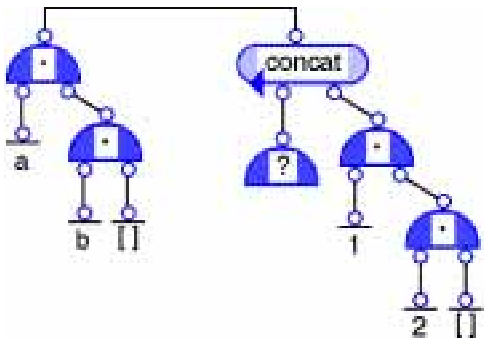


Figure 11: Result of a Merge and two Deletions

Note that the semantics of Lograph, informally presented in this example, are non-deterministic. In the example, we have chosen the "correct" cases of the definition to use in each application of replacement. There are many other executions leading to nontransformable queries, resulting from using other cases in replacement: however, if we define an *acceptable* result to be a query that would be reduced to an empty graph by deletion but for the presence of the **?** cell, then in this example there is only one acceptable result. Note that an acceptable result cannot contain any literal cells.

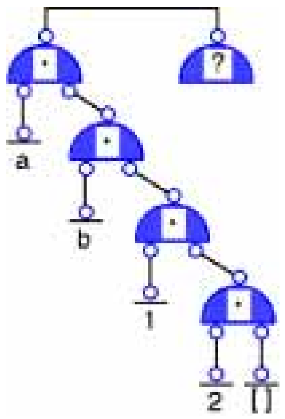


Figure 12: Final untransformable query

Clearly, a practical implementation of the language would have to deal with this nondeterminism in some way, for example by imposing an ordering on cases and performing a backtracking search like Prolog, or by allowing concurrent processing of some kind [2, 17, 24]. It is likely that any solution for Lograph, and therefore for LSD, would necessitate some modifications to the semantics and as a consequence, some syntactic additions.

As mentioned earlier in this section, Lograph programs have a textual representation. The textual equivalent of a Lograph case is a flat clause, a clause consisting of flat literals. A *flat literal* is either a literal of the form $p(x_1, \ldots, x_n)$ where p is a predicate symbol and $x_1, \ldots, x_n$ are variables, or an equality of the form $x = T$ where T is either a variable or T is a term of the form $f(x_1, \ldots, x_n)$ where f is a function symbol and $x, x_1, \ldots, x_n$ are variables. A *flat clause* is a Horn clause consisting of flat literals such that a variable occurring as the right-hand side of an equality cannot have any other kind of occurrence. Each function cell in the body of a Lograph case corresponds to an equality in the body of the corresponding flat clause; each literal cell corresponds to a literal constructed with a predicate symbol; and each terminal corresponds to a variable. To illustrate this correspondence, Figure 13 shows the flat clauses equivalent to the definition in Figure 4 and the query in Figure 5.

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

(a)

:- **concat**(x, y, z),
    $x = \bullet(x_1, u)$,
    $u =$ **[ ]**,
    $y = \bullet(y_1, y_2)$,
    $y_1 =$ **a**,
    $y_2 = \bullet(y_3, y_4)$,
    $y_3 =$ **b**,
    $y_4 = \bullet(y_5, u)$,
    $y_5 =$ **c**,
    concat(z, v, w),
    $v = \bullet(v_1, v_2)$,
    $v_1 =$ **1**,
    $v_2 = \bullet(v_3, u)$,
    $v_3 =$ **2**
    $w =$ **?**.

(b)

Figure 13: Flat clauses equivalent to (a) the definition in Figure 4; and (b) the query in Figure 5.

The three Lograph execution rules described above induce three corresponding deductive rules on flat clauses. In [9] a procedure is defined for converting an arbitrary set of Horn clauses into a set of flat clauses. This flattening procedure together with the three deduction rules constitute a proof system for first order logic called *Surface Deduction*, the soundness and completeness of which is proven in [9], thereby establishing the sufficiency of the Lograph execution rules.

In applying surface deduction rules, variables are unified as in other resolution-based proof procedures: however, because of the flat structure of clauses, these unifications involve simply substituting variables for variables. The details of unification and the discovery of nonunifiability are dealt with by the rules corresponding to Lograph's merge and deletion. A consequence of this is that with some additions, surface deduction, and therefore Lograph, deals with equality and can generate residual equational structures useful in abductive reasoning [7, 9, 18].

3. A Declarative Language for Structured Design (LSD)

The feature of Lograph that is particularly appealing in the current context is the consistent view it gives the user of data and operations on data. Representations of data and operations occur together in the same graphical structure, and execution can be viewed as a sequence of transformations of that graphical structure. Thus, Lograph places the focus directly on data. In fact, just as a network of function cells explicitly represents data, a network including function and literal cells can be seen as an implicit representation of data that can be made explicit by applying the transformation rules. This view is well suited to the design of structured objects since in that domain we are primarily concerned with viewing and transforming objects. Accordingly, to define LSD, a language for structured design, we extend Lograph by adding three new syntactic entities similar to functions cells, terminals and wires respectively, and one new execution rule. While acknowledging that designing three-dimensional objects is significantly more complex than designing two-dimensional ones, we restrict our attention to design in two-dimensions in order to introduce the fundamentals of our approach as straightforwardly as possible.

3.1 Extending Lograph to LSD

Here we introduce LSD informally via an example. To emphasise the view of program entities as design entities, we introduce new nomenclature for some Lograph concepts as we proceed. In such cases will follow the new name with the Lograph name in square brackets.

Figure 14 depicts an LSD *program* consisting of two designs, **partial cog** and **cog**. A *design* [definition] is a declarative specification for a type of component, instances of which are represented in other designs by *implicit components* [literal cells] or *i-components* for short. For example, **partial cog** defines an incomplete cog with a particular number of teeth as consisting of an extra tooth bonded on to an incomplete cog with one less tooth, represented by the i-component **partial cog**. The case of **partial cog** in the centre of Figure 14 corresponds to an incomplete cog with no teeth, providing the base case of this recursive design. The other design in this program, **cog**, consists of a single case defining a complete cog as the component that results from bonding together the open edges of a partial cog.

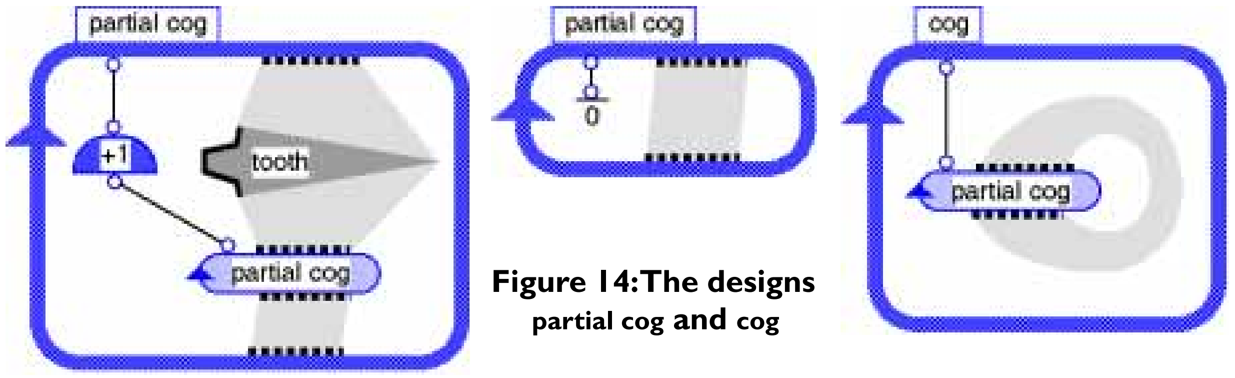


Figure 14: The designs partial cog and cog

Each of the two cases of the design **partial cog** has three terminals: the first is a simple terminal, while the other two are *edge terminals*. Simple terminals are as in Lograph, while an edge terminal, shown as a broken heavy line ▪▪▪▪▪▪▪▪ , is an implicit representation for an open edge along which a component can be bonded to another component. We will go into this in more detail below. The requirement that different cases of a Lograph definition have the same name and arity is strengthened in LSD by requiring that they must have the same name and *signature*, where the signature is a sequence of terminal types. For example, the signature of each of the two cases of **partial cog** is *(s, e, e)*, where *s* and *e* indicate simple and edge terminals respectively. The body of the recursive case consists of a function cell, one i-component, a component named **tooth**, and three bonds. The body of the other case consists of a function cell and a bond. Like the heads of the cases of the design **partial cog**, the i-component **partial cog** has the signature *(s, e, e)*. This component implicitly represents an instance of the design **partial cog**.

The component in the recursive case of **partial cog** of Figure 14 is an example of an *explicit component*, or *e-component* for short, and explicitly represents a single tooth of a cog. For simplicity, some details are omitted from the representation of this component in Figure 14. A more complete picture of it is shown in Figure 15, revealing in particular its two open edges, each marked by a row of chevrons called an *indicator*. The "inside" of the object lies on the right of the indicator. An *open edge* is a line along which one e-component can be "bonded" to another to form a new e-component. A component is said to be *open* or *closed* depending on whether or not is has any open edges. An open component is considered to be "unfinished" in the sense that it does not represent a real object, but can exist only as a part of a larger object.

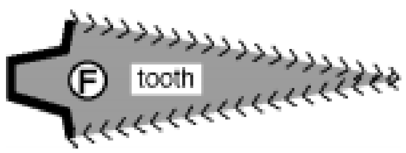


Figure 15: An explicit component

E-components are similar to function cells in that both are constituents of data structures. An e-component differs from a function cell in that it can be connected to other structures in a program only via open edges, it has a parameterised internal state and has a customised visual representation that may vary depending on the values of its parameters, which may correspond to characteristics such as size, position, orientation in the plane or colour. Because of this parameterisation, an e-component represents a family of components, and its appearance is actually a "typical" one chosen to be representative of that component family. Substituting values for parameters produces an e-component defining a more constrained family of e-components. When all parameters have been replaced by constants, a *bound* e-component is obtained, fully specified in all respects. Such a component will be locked into a particular position, will not be deformable, will have a fixed colour, and so forth. An e-component may have several sets of parameters, providing alternate ways of specifying its characteristics. For example, the definition of an e-component may include variables for fixing its position according to polar coordinates, and others that accomplish this in cartesian coordinates. An e-component is *free* if and only if it is not bound, in which case its visual representation includes the icon Ⓕ as shown in Figure 15.

The third new construct added to Lograph is the *bond*, a grey band each end of which is incident on an edge terminal or open edge. A bond is either *external* or *internal*, depending on whether or not it is incident on a terminal of the head of the case. An internal bond indicates open edges to be joined in defining a new component. An external bond indicates an open edge to be "exported" from the design. Bonds appear in all three cases in Figure 14.

Components that are implicitly represented, such as the component **partial cog** in the leftmost case of Figure 14, have edge terminals rather than open edges. During the execution process described in the next section, implicitly represented components are made explicit, at which time their terminals become open edges. Hence an edge terminal is an implicit representation for an open edge.

3.2 Assembly

Execution of an LSD program involves applying execution rules to transform an *assembly request* (or simply *request*), which is similar to a Lograph query except that it may also include bonds. Execution transforms the request using the Lograph rules together with a new rule called *bonding*, with the aim of obtaining a graph which contains no implicit components, and cannot be further transformed. We call this execution process *assembly*. To illustrate, we consider the assembly of a nine-toothed cog starting with the request in Figure 16(a). Note that we are using the constant **9** is an abbreviation for the structure in Figure 16(b) built from the constant 0 and function cells which we interpret as the successor function.

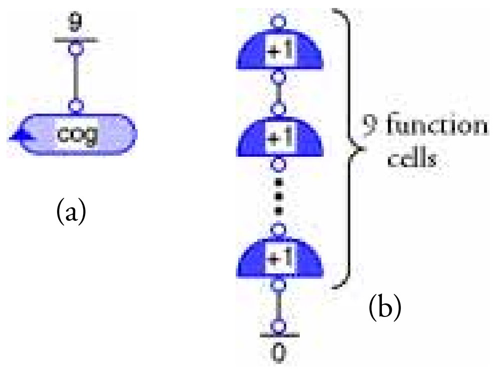
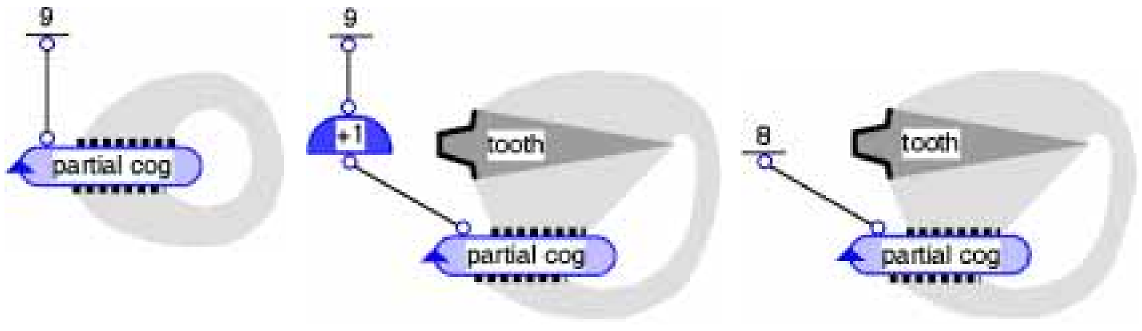


Figure 16:
(a) Specification of a 9-toothed cog
(b) Expansion of 9

Figure 17 shows the first few steps of the assembly. The request in Figure 16(a) is transformed into the request in Figure 17(a) by replacement with the case **cog**. Figure 17(b) is obtained from Figure 17(a) by replacement with the recursive case of **partial cog**.

Note that during this replacement the bond in the request and the upper and lower bonds from the copy of the case body are blended into one. From a graphical transformation viewpoint, this is consistent with the way wires and simple terminals would behave during replacement: the underlying mechanism is also identical.

If we were to choose the base case of **partial cog** for this replacement step, the bond from the request in Figure 17(a) and the bond from the copy of the case would blend together and disappear, leaving the constants **9** and **0** connected by their roots, a graph which cannot be further transformed. This is clearly an "unacceptable" result.

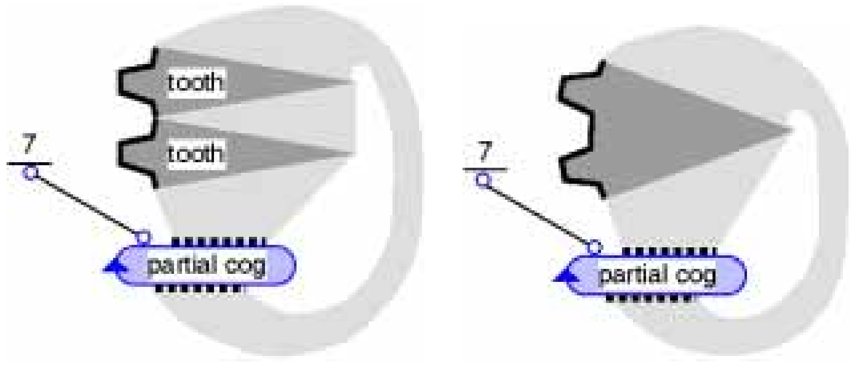


(a) (b) (c)

Figure 17: Two applications of Replacement, followed by Merge and Deletion

Recalling that **9** is an abbreviation for the structure in Figure 16(b), it is clear that merge followed by deletion applied to the request in Figure 17(b) results in 17(c), where like **9, 8** abbreviates a linear structure of function cells.

Applying replacement with the recursive case of **partial cog** to the i-component in the request in Figure 17(c), followed by merge and deletion, results in the request in Figure 18(a) where an internal bond connects two open edges. This is the situation in which the new LSD execution rule *Bonding* applies, producing the request in Figure 18(b). Bonding creates one new e-component by joining the two e-components along their open edges. The internal state and parameters of the new e-component are determined from the states and parameters of the constituent e-components. Clearly, certain parameters of the two compo-



(a) (b)

Figure 18: Replacement, Merge and Deletion followed by Bonding

nents will become mutually constrained as a result. Here for example, the two components are rotated relative to each other in order to accomplish the bonding.

Repeating this pattern of execution steps eventually leads to the request in Figure 19(a), to which only replacement applies. Replacement with the base case of **partial cog** followed by merge and deletion produces the request in Figure 19(b). Finally, bonding results in the request in Figure 19(c), consisting of one closed explicit component.

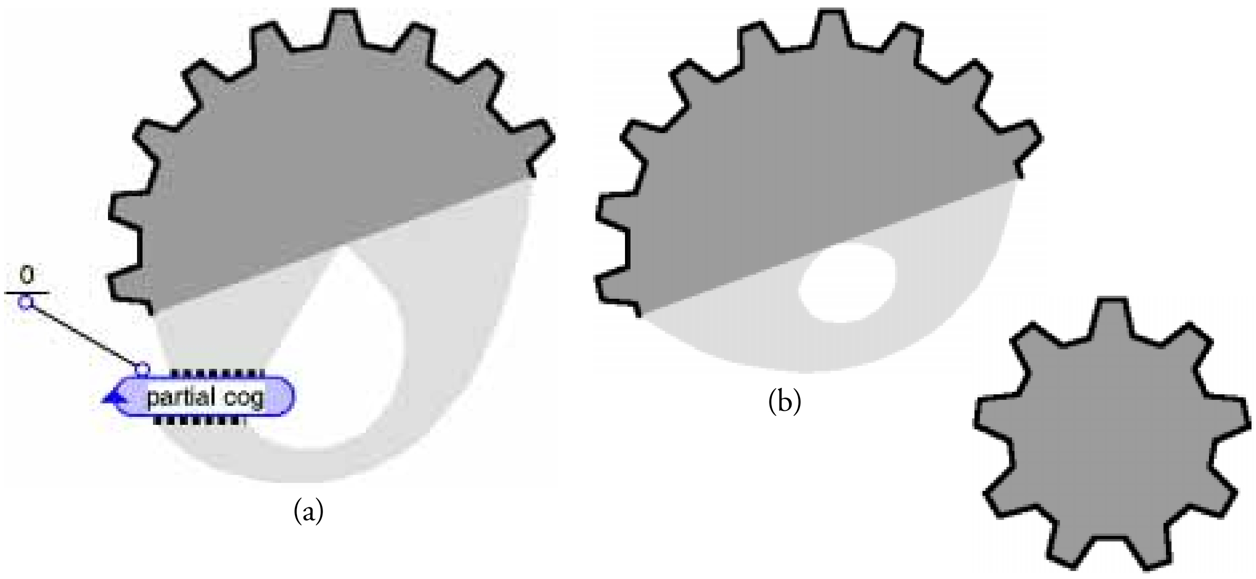


Figure 19: Completing the assembly

When the bonding rule is applied, it may happen that the resulting e-component represents an object which for some reason cannot physically exist. For example, suppose we attempted to assemble a cog with one tooth. Obviously this could be prevented by appropriate constraints on the extent to which a tooth can be deformed. This will be dealt with in Section 4.

The e-component produced by the above assembly is closed. However, this would not be the case if an execution were to begin with a request containing a component with an open edge or edge terminal not attached to a bond. To ensure that in the visual representation of an i-component all open edges that would result from execution are apparent, we require that all edge terminals must be attached to a bond within a case.

Although we have used nomenclature in LSD suggestive of the main purpose of the language, it is important to emphasise that the language is an extension of Lograph and can therefore accomplish the same symbolic computations. We filter the results of computation depending on what we are trying to achieve. If, for example, we are interested in abductive reasoning in some design domain, we might perform computations that terminate in some explicit components together with residual equational structures embodying conditions that need to be satisfied for such components to built.

4. Underlying Formalism

To put the language described above on a solid foundation, we must not only fit the new concepts into the formal logic that underlies Lograph, but also show how this formalisation relates to real design domains and maps back into visual representation. This last issue is important since LSD deals with both abstract entities and entities with significant physical properties. At the logical level LSD, like Lograph, simply performs symbolic manipulations with both. At the interface level, however, abstract entities such as function cells and i-components are represented by generic icons, while representations for e-components are generated by drawing functions which are necessarily outside the scope of the formalism, but rely on values and constraints computed by it. We deal here only with the logical level of LSD. The extra-logical aspect, that is, the model for representing solid objects, is discussed in [11].

The following definitions 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*. For convenience, we assume the existence of special function symbols $\mathbf{f}_2$, $\mathbf{f}_3$ … of arity 2, 3 … respectively, which we will use for grouping terms. For each $i \geq 2$ and terms $t_1, \ldots, t_i$ we denote the term $\mathbf{f}_i(t_1, \ldots, t_i)$ by $[t_1, \ldots, t_i]$.

The reader is encouraged to relate the entities defined below to the example in the preceding section, noting in particular, how the parts of an entity correspond to features in its representation. We will make comments about aspects of visual representation as appropriate to clarify the relationship between formalism and appearance.

Definition: A *function cell* is a literal of the form $x = \mathbf{f}(y_1, \ldots, y_k)$ where $\mathbf{f}$ 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: An *implicit component* (*i-component*) is a literal of the form $\mathbf{p}(v_1, \ldots, v_k)$ where $\mathbf{p}$ 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 $(\mathrm{F}_1, \ldots, \mathrm{F}_k)$ where $\mathrm{F}_i = \mathbf{s}$ if v_i is a simple terminal, and $\mathrm{F}_i = \mathbf{e}$ otherwise, for each i $(1 \leq i \leq k)$.

Definition: 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 $\mathbf{q}$ 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.

If $w = [[x_1, y_1, x_2, y_2], [u_1, \ldots, u_m]]$ is an open edge, $[x_1, y_1, x_2, y_2]$ and $[u_1, \ldots, u_m]$ are respectively called the *vector* and *bonding conditions* of the edge.

Definition: For each explicit component **e**, there exists a formula $\mathbf{K_e}$ called the *specification* of **e** such that if x is a variable occurring in **e**, then x occurs in $\mathbf{K_e}$ if and only if x is not an edge terminal. An e-component is *valid* iff its specification is satisfiable; otherwise it is *invalid*.

Definition: If **e** is an e-component, let $\mathbf{V} = \{w_1, \ldots, w_n\}$ be the set of free variables of $\mathbf{K_e}$ and let **W** be some subset of **V**. We assume, without loss of generality that $\mathrm{W} = \{w_1, \ldots, w_t\}$ for some $t \leq n$. Let us denote by $\mathbf{K_e}[\mathrm{a}_1, \ldots, \mathrm{a}_n]$ the formula obtained by replacing all occurrences of w_i by some term a_i for each i $(1 \leq i \leq n)$, then **W** is said to be *sufficient* iff $\forall w_1, \ldots, \forall w_n \forall x_{t+1}, \ldots, \forall x_n$ $(\mathbf{K_e} \wedge \mathbf{K_e}[w_1, \ldots, w_t, x_{t+1}, \ldots, x_n] \supset w_{t+1} = x_{t+1} \wedge \ldots \wedge w_n = x_n)$ is valid. Clearly **V** itself is sufficient. A subset of **V** is *necessary* iff it has no sufficient subset. A necessary and sufficient subset of **V** is called a set of *parameters* for the component.

It is easy to show that every superset of a sufficient set is sufficient, and that $\varnothing$ is a set of parameters for an invalid component.

Definition: An e-component is *bound* iff $\varnothing$ is a set of parameters, otherwise the component is said to be *free*.

Definition: An e-component is *open* iff it has at least one open edge, otherwise it is said to be *closed*.

Our discussion so far has been concerned with a logic for characterising the generic aspects of designing structured objects, in which certain assumptions have been made about the properties of components. For example, it is assumed that if $[x_1, y_1, x_2, y_2]$ is the vector of an open edge, then (x_1, y_1) and (x_2, y_2) are points in the plane, that an open edge lies on the perimeter of the visual representation of the component, and that the inside of the component is to the right of an edge. Ensuring that such conditions hold is beyond the scope of LSD, and must be accomplished by the component specification.

Specifications for explicit components provide the connection to the actual design domain. Consequently, by "satisfiable" in the preceding definition, we mean satisfiable in conjunction with some set of axioms that characterise the design domain. This set undoubtedly includes axioms for real numbers and geometry, and may include axioms that describe manufacturing processes, the properties of materials for fabricating objects and so forth.

Execution of the bonding rule, illustrated in the example in Section 3 and formalised below, requires that the validity of e-components be verified. One could use a general theorem-prover for this purpose, but that approach is unlikely to be tractable for a design domain complex enough to be useful. Tools such as constraint solvers together with domain-specific verification mechanisms are more likely to be successful.

The specification for a component defines it fully within allowable variations of parameter values, and therefore will include such information as solid modelling data, fabrication materials and constraints on deformation and movement. In particular, the specification determines how the component is drawn. To this end we assume the existence of a drawing function Δ, dependent on the axioms that characterise the design domain, that maps closed formulae to two-dimensional drawings.

The visual representation of an e-component consists of a *name*, a *body*, and a set of *indicators*. The name is the explicit symbol corresponding to the component. The body and indicators depend on the component's classification according to the above definitions. If the component is invalid, it has no indicators, and its body is represented by an octagon containing question marks as shown in Figure 20.

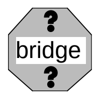
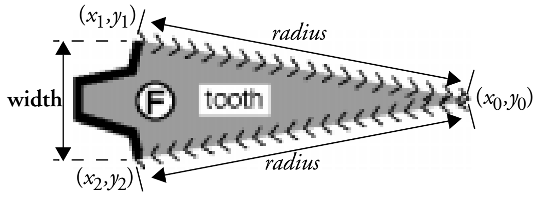


Figure 20: Invalid component

If an e-component **T** is valid, it has one indicator for each open edge, and its body is a two-dimensional drawing. If **T** is bound, its body is $\Delta(\mathbf{K_T})$. Otherwise, the body consists of the symbol Ⓕ superimposed on an "average" representation. This average representation is computed by the following process. First a "reasonable" set of values is chosen for some set of parameters of **T**, and a formula $\mathbf{K_{T1}}$ obtained by replacing the parameters in $\mathbf{K_T}$ by these values. Next $\mathbf{K_{T1}}$ is closed by existentially quantifying all free variables occurring in it, obtaining $\mathbf{K_{T2}}$. Finally $\Delta(\mathbf{K_{T2}})$ is evaluated to produce the required two-dimensional drawing. In a later section we will consider what constitutes a reasonable choice of parameter values for an average representation for a valid, free e-component. As an example, consider the valid, free, open e-component **T** in Figure 21(a). A visual representation for this component is shown in Figure 21(b) with some annotations to relate it to (a), and to the specification in Figure 21(c).

$$\begin{aligned}\mathbf{T} = \{&u = [[x_1, y_1, x_0, y_0], [colour]],\\ &\mathrm{v} = [[x_0, y_0, x_2, y_2], [colour]],\\ &\mathbf{tooth}(x_0, y_0, x_1, y_1, x_2, y_2, colour)\}\end{aligned}$$

(a) Component T



(b) Visual representation

$$\begin{aligned}\mathbf{K_T} = (&colour = \mathbf{grey} \vee colour = \mathbf{black})\\ &\wedge \mathbf{real}(x_0) \wedge \mathbf{real}(x_1)\\ &\wedge \mathbf{real}(x_2) \wedge \mathbf{real}(y_0)\\ &\wedge \mathbf{real}(y_1) \wedge \mathbf{real}(y_2)\\ &\wedge (x_1 - x_0)^2 + (y_1 - y_0)^2 = radius^2\\ &\wedge (x_2 - x_0)^2 + (y_2 - y_0)^2 = radius^2\\ &\wedge radius \geq \mathbf{min}\\ &\wedge (x_2 - x_1)^2 + (y_2 - y_1)^2 = \mathbf{width}^2)\end{aligned}$$

(c) Specification

Figure 21: A free, open explicit component

It is important to note that the representation in Figure 21(b) reveals only some of the characteristics of a component. For example, although the superimposed Ⓕ indicates that the component is free, there is no indication of exactly which characteristics may vary, or how they are constrained. Furthermore, some parameters may simply have no sensible visual representation: for example one of the open edges of **tooth** may have a bonding condition indicating the type of steel that a cog is to be fabricated from. Such issues are discussed in later sections.

The specification in Figure 21(c) defines allowable colours, ensures that the heads and tails of the open edges are points in the plane, and deals with and some aspects of the geometry of the component. Since cogs must mesh together, there needs to be some regularity in their teeth. If we assume that the shape of the tooth is defined by some function which ensures that the "valleys" between teeth are at right angles to radial lines, and that the "hills" of one cog will fit into the "valleys" of another, then the width of a tooth sector must be fixed and the radius must be greater than some minimum value. Clearly this specification includes only a small fraction of the information that would required to properly define such a component

Definition: 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.

For simplicity, the definitions of internal bonds and e-components are expressed in terms of literals that are not flat. Replacing nested terms by variables and adding corresponding equalities, as described in [18] produces equivalent flat literals to which surface deduction can be applied.

Definition: 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.

Definition: An LSD *program* is a set of uniquely named designs with no variables in common.

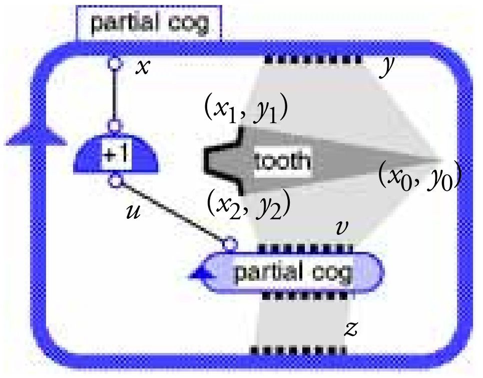
Figure 22 illustrates the connection between a case and its visual representation, In (a) the underlying representation of the LSD program in Figure 14 is shown as a set of clauses which are

```
partial_cog(x, y, z) :-
      x = +1(u),
      y = [[x1, y1, x0, y0], [colour]],                ⎤
      w = [[x0, y0, x2, y2], [colour]],                ⎥ explicit
      tooth(x0, y0, x1, y1, x2, y2, colour),          ⎦ component
      w = [[x3, y3, x4, y4], r],                       ⎤ internal
      v = [[x4, y4, x3, y3], r],                       ⎦ bond
      partial_cog(u, v, z).

partial_cog(x, y, y) :- x = 0.

cog(x) :- partial_cog(x, y, y).
```

(a)



(b)

Figure 22: Textual representation of LSD program in (almost) flat clauses, and the visual representation of one case

almost flat except that for clarity we have left the nested terms in some of the equality literals. Figure 22(b) depicts the recursive case of the design **partial cog** annotated with variable names to relate it to the first clause in (a). Every simple terminal in the clause is explicitly represented in the case. The grey band between the two components in (b) represents the internal bond, the two labelled equality literals in (a). Note that since the open edges along which components are bonded have a direction, it is necessary to ensure that the bonding matches the head of one edge with the tail of the other. This is accomplished by the two equality literals of an internal bond . The upper grey band in (b) is like a wire in that it represents the simple sharing of the edge terminals between the head of the clause and the first equality literal of the e-component. Similarly the lower grey band represents the fact that the edge terminal z occurs both in the head and the i-component.

It remains only to explain the semantics of bonding, which are partially embodied in surface deduction, but require one small extension.To do this we will demonstrate the surface deduction equivalent of the bonding step illustrated in Figure 18. The request in Figure 18(a) is represented by the set of literals in Figure 23(a), where the literals that constitute the internal bond between the two e-components are outlined. To execute this bond we first unify the two equalities with root w_1: this corresponds to merging two literal cells in Lograph. The root w_1 of the resulting single equality does not occur anywhere else, so the equality is deleted. Similarly, the equalities with root y are merged and deleted. The resulting set of literals is shown in Figure 23(b).

So far, execution of bonding has been accomplished purely by applying surface deduction rules without modification. However, since our goal is to formulate a rule that allows us to construct new e-components out of existing ones, it is necessary that the structure that results after applying it fits the formal definition of e-component. Hence, one final step is necessary, requiring a minor addition to

$$\begin{array}{l}
\{\ u = \mathbf{7}, \\
\quad z = [[x_6, y_6, x_5, y_5], [hue]], \\
\quad w_1 = [[x_5, y_5, x_7, y_7], [hue]], \\
\quad \textbf{tooth}(x_5, y_5, x_6, y_6, x_7, y_7, hue), \\
\quad \boxed{\begin{array}{l} w_1 = [[x_8, y_8, x_9, y_9], p], \\ y = [[x_9, y_9, x_8, y_8], p]. \end{array}} \\
\quad y = [[x_1, y_1, x_0, y_0], [colour]], \\
\quad w = [[x_0, y_0, x_2, y_2], [colour]], \\
\quad \textbf{tooth}(x_0, y_0, x_1, y_1, x_2, y_2, colour), \\
\quad \textbf{partial\_cog}(u, v, z). \\
\}
\end{array}$$

(a)

$$\begin{array}{l}
\{\ u = \mathbf{7}, \\
\quad \boxed{\begin{array}{l} z = [[x_6, y_6, x_0, y_0], [colour]], \\ \textbf{tooth}(x_0, y_0, x_6, y_6, x_1, y_1, colour), \\ w = [[x_0, y_0, x_2, y_2], [colour]], \\ \textbf{tooth}(x_0, y_0, x_1, y_1, x_2, y_2, colour), \end{array}} \\
\quad w = [[x_3, y_3, x_4, y_4], r], \\
\quad v = [[x_4, y_4, x_3, y_3], r], \\
\quad \textbf{partial\_cog}(u, v, z). \\
\}
\end{array}$$

(b)

$$\begin{array}{l}
\{\ u = \mathbf{7}, \\
\quad z = [[x_6, y_6, x_0, y_0], [colour]], \\
\quad w = [[x_0, y_0, x_2, y_2], [colour]], \\
\quad \textbf{two\_teeth}(x_0, y_0, x_6, y_6, x_2, y_2, colour), \\
\quad w = [[x_3, y_3, x_4, y_4], r], \\
\quad v = [[x_4, y_4, x_3, y_3], r], \\
\quad \textbf{partial\_cog}(u, v, z). \\
\}
\end{array}$$

(c)

Figure 23: Execution of Bonding in Surface Deduction

the surface deduction semantics: that is we must create a new e-component out of the literals that remain from the two involved in the bonding. These literals are outlined in Figure 23(b). To accomplish this step, the two literals with the explicit symbol **tooth** are replaced by a single literal constructed with a new explicit symbol **two_teeth**, and the variables which occur in the remaining open edges. This results in the set of literals in Figure 23(c). Finally, 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.

Let us define an acceptable execution as one that terminates in a single closed e-component. Clearly the soundness and completeness of surface deduction guarantees that, if the proof procedure used to establish the validity of e-components is sound and complete, then there exists a valid e-component corresponding to a request iff there is an acceptable execution of that request, in which case the e-component constructed by the execution is an example.

Note that, given the correspondence between Lograph and surface deduction, we could use Lograph directly as our visual design language, in which case internal bonds and e-components would be represented by networks of function and literal cells. For example the Lograph query in Figure 24 corresponds to the LSD request in Figure 18(a) and the flattened form of the set of literals in Figure 23(a). Although Lograph provides a clear picture of abstract logical structures, the visualisations supplied by LSD map much more directly to the structures in a design domain and therefore, according to the "closeness of mapping" criterion proposed in [15], are likely to contribute significantly to understanding visual design problems. This example illustrates the magnitude of the cognitive distance between the two visual representation.

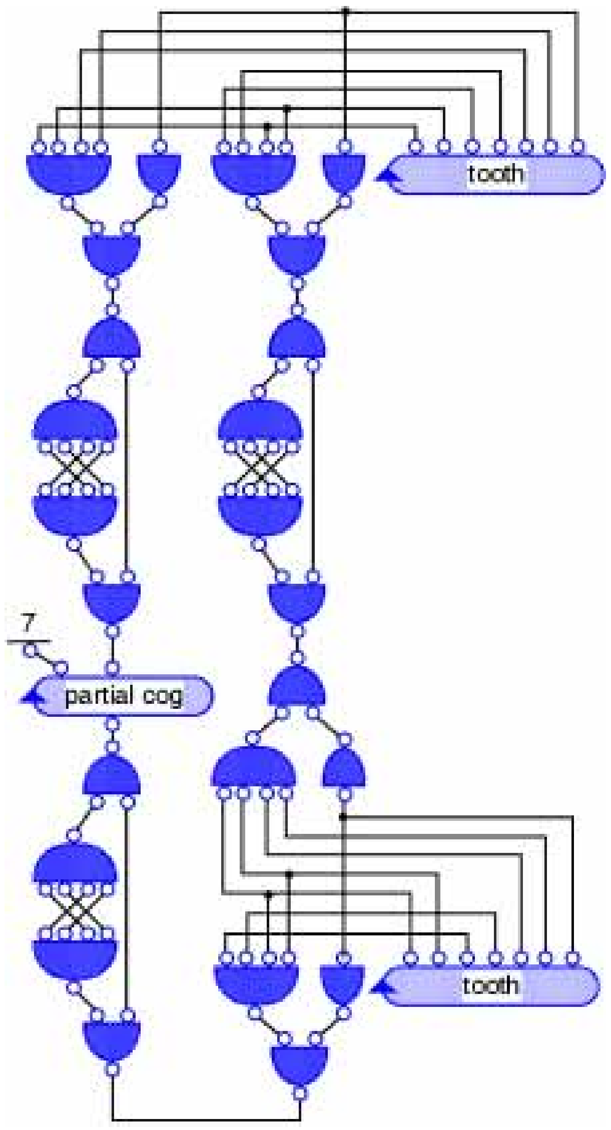


Figure 24: Lograph query equivalent to the LSD request in Figure 23(a). Unnamed function cells represent grouping functions of various arities

5. Discussion - research directions

5.1 Further extensions

To obtain LSD from Lograph we have added some new syntactic elements and one new execution rule. Bonding, however, is insufficient for describing any structure. For example, the pistons of an internal combustion engine bear a particular relationship to the block; they slide within the confines of the cylinders but are not attached to them. Clearly other rules for combining objects are necessary.

The solid modelling portions of CAD/CAM systems usually provide some very basic operations for building objects based on set operations together with such operations as extrusion. We will investigate the possibility of incorporating a primitive set of such rules into LSD from which to construct higher level rules such as bonding. It may then be possible to devise metarules with which an LSD "superuser" could customise the language for a particular design domain.

5.2 Semantics and Execution

As discussed above, the semantics of LSD is provided by surface deduction with a minor addition to cope with bonding. We have also noted some interesting properties of surface deduction beyond those needed to support LSD. In particular, surface deduction provides a uniform mechanism for incorporating equality into the deductive process, so is well equipped to deal with many of the constraints that are likely to arise [9, 18]. It might be possible, therefore to soften the hard line we have drawn between LSD semantics and the verification of validity by using surface deduction to deal with some of the equalities that occur in the constraints of a component. Since constraint solving and constraint satisfaction are such key issues, it also seems likely that constraint logic programming would be applicable [16]. A promising direction for further investigation, therefore, is how surface deduction and constraint logic programming might interact in the component verification process.

An important related issue is efficient execution, and the order of application of execution rules to a graph. As noted in Section 2, Lograph deduction rules have no imposed order, so both AND-parallelism and OR-parallelism are possible. Clearly, the sample LSD execution in Section 3 relies on ordered execution, or at least on the ability to impose an ordering. While on the one hand we acknowledge that research is necessary to determine the degree to which ordering is necessary, we also conjecture that designs for structured objects are quite deterministic in nature, so that a designer/programmer will always know the order in which cases of a design should be tried in an execution. This would suggest that in a practical LSD-based system, the order of execution of cases and components, like the order of clauses and body literals in Prolog, should be determined by the programmer. It might be possible, however, to automatically order components within cases to guarantee the earliest possible unification failure, similar to the way Prograph orders operations in cases [20].

5.3 Environment

As is the case with any programming language, and especially visual languages, the characteristics of a development environment which could be built around LSD are as important as the language itself. For example, we noted above that our visual representation for the e-component tooth is inadequate in various respects. To generate an average picture, a set of "reasonable" values for a set of parameters is chosen. This could be done automatically by, say, choosing a median value from an ordered set of possible values for a parameter. A more refined choice could be based on considering the constraints on a component rather than just the range of values of individual parameters. For example, if the contraints were hierarchically organised, a "best" solution might be chosen according to some appropriate comparator predicate [4]. In practice, however, it may turn out that the users who design the initial, primitive e-components and place them in a library are most likely to know what good average representations are. It would seem sensible therefore for the development environment to allow the user to choose average parameter values for these primitive components, then combine these averages in some way to obtain "reasonable" parameter values for generated e-components.

Similarly, the environment should provide tools for the user to investigate a component's properties in an intuitive way. For example, the user might be able to test the spatial constraints of a component by trying to drag or deform it. Or more sophisticated tools for viewing and specifying constraints could be incorporated, along the lines of those described in [29]. Facilities could be supplied for defining views of different kinds for primitive components. As components are combined, corresponding kinds of views could also be combined where possible, for the user to display as desired.

One of the important principles in our proposed language is that the objects being manipulated should be represented as concretely as possible. The environment should, therefore, attempt to replace abstract representations with more concrete ones whenever they can be computed. One possibility might be to perform some background analysis or partial executions of a design while the user is editing in an attempt to build an icon for it. Test executions of the tail recursive **partial cog** design in Figure 14 could relatively easily reveal the execution pattern, which, combined with the use of reasonable parameter values, could be used to construct an appropriate icon: a cog with a missing wedge.

Another important visualisation tool that should be supplied by an LSD environment is animated execution. The Lograph execution rules and the bonding rule described above all lend themselves well to animation given that they have a natural expression as graph transformations. We have produced an animated simulation of the execution in Section 3, and feel that such animations would be a significant aid in understanding and debugging a design (http://www.cs.dal.ca/~smedley/papers/cog.mov).

5.4 Simulating Behaviour

A particularly important aspect of the design of structured objects is the simulation of the behaviour of objects. With programming languages, the structures being designed are algorithms, the behaviours of which are specified by their structures. So in this case the form and the function are synonymous. An example in which form and function are somewhat separated is the design of electronic circuits. In a language for VLSI design such as VHDL, visualisation is very indirect, to the extent that designs are represented textually. In VHDL "execution" means simulating the behaviour of a design, rather than computing a visual representation. Nevertheless, there is a strong relationship between form and function in this domain, since VLSI circuits are somewhat akin to dataflow diagrams, and executing them involves the flow of signals through them. In many domains, form and function are widely separated. For example, in a structure such as a bridge, one possible behaviour might be reaction to stress, which is related to the structure in a far less obvious way.

In our investigations so far, we have concentrated on structure. Clearly, our proposed language must be able to build the behaviour of compound components from the behaviours of its constituents, and to execute these behaviours to provide working simulations. Note that, since our language is used to describe families of objects (for example, a family of gears, where the number of teeth is specified as a parameter), our facilities for describing behaviour must also apply to parameterised descriptions. We expect that finding suitable visual representations for use in defining behaviours will present a challenging visual language design problem.

5.5 Compilation

One of the observations discussed in Section 1 leading to LSD was the strong analogy between the design of software and the design of other structured objects. In drawing this analogy, we noted that the design process usually concludes with some kind of compilation, producing a usable device. In the case of software, the result is an application. In VLSI design, compilation produces code for a chip fabricator to build a hardware device. It seems unlikely that any kind of generic compilation is possible for programs in a general design language such as LSD. It may be possible, however, to generate some kind of intermediate code, equivalent to the abstract machine code produced by the first pass of a two-pass compiler.

6. Concluding remarks

Based on the observation that logic programming languages give a uniform view of data and the algorithms that operate on it, we have presented a preliminary proposal for a declarative, logic-based, visual language for structured design (LSD). This language is derived from Lograph, a visual logic programming language, by adding a new structure called an "explicit component" and rules with which to transform it. The main advantage of this approach is seen to be the natural integration of concrete representations of components with code that specifies how a component is composed of constituent components.

Further work in this area will concentrate on the formal basis of this language, based on the surface deduction semantics of Lograph, extended to account for the new constructs. We also expect that constraint logic programming will play a part in the formal semantics of LSD. Other important issues to be examined are how to specify, combine and execute component behaviours, and what tools might be appropriate in an LSD-based design environment.

7. Acknowledgments

This work was supported in part by Natural Sciences and Engineering Research Council grants OGP0121312 and OGP0000124.

8. References

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

[2] A. Beaumont & G. Gupta (eds.) (1991) Parallel Execution of Logic Programs. *Proceedings of Int'l. Conference on Logic Programming, Pre-Conference Workshop*, Paris, Lecture Notes in Computer Science, vol. 569, Springer-Verlag.

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

[4] A. Borning, B. Freeman-Benson & M. Wilson (1992) Constraint Hierarchies. *Lisp and Symbolic Computation*, 5(3), 223-270.

[5] P.T. Cox, S. Matwin & T. Pietrzykowski (1985) Using LOGRAPH to query databases. *Proceedings IEEE COMPINT 85*, Montreal, pp 168-171.

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

[7] P.T. Cox & T. Pietrzykowski (1985) Surface Deduction: a uniform mechanism for logic programming. *Proceedings of Second Int'l. Symposium on Logic Programming*, Boston, pp 220-227.

[8] P.T. Cox & T. Pietrzykowski (1985) Solving graph problems using LOGRAPH. *Proceedings of a Symposium on the Role of Language in Problem Solving* 1, B. Hamill, D. Weintraub, R. Jernigan (eds.), North Holland, pp 221-233.

[9] 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.

[10] 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.

[11] 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, to appear.

[12] 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.

[13] E.J. Golin, M.J. Haney, E. Huges, D. Miller-Karlow & G. Tharakan (1992) *Visual design with vVHDL.* University of Illinois at Urbana-Champaign, Department of Computer Science Technical Report #1745.

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

[15] T.R.G. Green & M. Petre (1996) Usability Analysis of Visual Programming Environments: A 'Cognitive Dimensions' Framework. *Journal of Visual Languages and Computing* 7(2), 131-174.

[16] J. Jaffar & M. Maher (1994) Constraint Logic Programming: A Survey. *Journal of Logic Programming*, 19(20), 503-581

[17] K. M. Kahn & V. A. Saraswat (1990) Complete Visualizations of Concurrent Programs and Their Executions. *Proceedings of the 1990 IEEE Workshop on Visual Languages*, Skokie IL, pp 7-15.

[18] E.H. Knill, P.T. Cox & T. Pietrzykowski (1993) Equality and Abductive Residua for Horn Clauses. *Theoretical Computer Science* 120, 1-44.

[19] A. Paoluzzi & C. Sansoni, Programming Language for solid variational geometry. *Computer Aided Design* **24**, (1992), 349-366.

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

[21] C.S. Pierce (1933) *Collected papers of Charles Sanders Pierce*. C. Hartshorne and P. Weiss (Eds.), Harvard University Press, v3.

[22] J. Puigsegur, W.M. Schorlemmer & J. Agustí (1997) From Queries to Answers in Visual Logic Programming. *Proceedings of the 1997 IEEE Symposium on Visual Languages,* Capri, pp 102-109.

[23] A. Rau-Chaplin, B. MacKay-Lyons & P. Spierenburg (1996) The LaHave House Project: Towards an Automated Architectural Design Service. *Proceedings of the International Conference on Computer-Aided Design (CADEX'96)*, IEEE Computer Society Press, pp 25-31.

[24] E. Shapiro (Ed.) (1988) *Concurrent Prolog: Collected Papers (Logic Programming Series)*. MIT Press.

[25] 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.

[26] J.F. Sowa (1984) *Conceptual structures: information processing in mind and machine*. Addison-Wesley .

[27] *Standard* VHDL *Language Reference Manual* — Std 1076-1987. IEEE (1988).

[28] D.E. Thomas & P.R. Moorby (1991) *The Verilog Hardware Description Language*. Kluwer Academic Publishers.

[29] A. van Dam (1993) Visual Communication Via 3D User Interfaces, Keynote address. *IEEE Workshop on Visual Languages,* St. Louis.