Defining Behaviours for Solids in a Visual Design Environment

Omid Banyasad
Philip T. Cox

Technical Report CS 2002-07

Oct 7, 2002

Faculty of Computer Science
6050 University Ave., Halifax, Nova Scotia, B3H 1W5, Canada

Defining Behaviours for Solids in a Visual Design Environment

Omid Banyasad
banyasad@cs.dal.ca

Philip T. Cox
pcox@cs.dal.ca

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

Abstract

*The design of structured objects is frequently accomplished with the use of Computer-Aided Design (*CAD*) systems, some of which allow for parametrised designs, which represent families of objects. Some existing* CAD *systems permit parametrisation by providing interfaces to programming languages, leading to a sharp division between the visual and programming aspects of building complex designs. In previous work, a design language* LSD *was proposed, which used visual logic programming to overcome this separation and provides "assembly semantics" for building an instance of a parametrised design corresponding to specific parameter values.*

Also of interest to designers, however, is the behaviour of the objects they design. For example, the point of building a mechanical device such as an internal combustion engine is to obtain a particular mechanical behaviour. Here we extend the formal model for solid objects on which LSD *relies in order to support the definition of particular kinds of behaviours.*

1 Introduction

In today's competitive market, most manufacturing companies are hard pressed to deliver their products more cheaply and quickly. This forces them to seek more cost-effective and efficient design and production techniques. The need for better and faster design, in turn, requires new tools for design as well as new design methodologies [11]. Such new tools and design methodologies are primarily focused on reducing the design cycle time by allowing faster synthesis, early detection of flaws, rapid prototyping, and quick analysis. Integration of suitable design tools and new methodologies in a design environment is a key point in faster and better design if the integration facilitates a smooth and seamless transition of the design from one stage to the next.

One of the major objectives of integrating design tools in a Computer-Aided Design (CAD) system is to provide support for parametric design, whereby families of structured objects can be specified, such as an *n*-bit multiplexor, or internal combustion engine with *n* cylinders. We believe that there are also other significant advantages in integrating appropriate design tools in a CAD system. For example, a parametrised design environment for mechanical devices with capabilities for simulating physical behaviours of design parts would be a valuable tool for proving the viability of a design, detecting design flaws, or creating early demos for customers.

To allow for parametric design, commercial CAD systems typically either include a textual programming language or provide an interface to one. For example, AutoCAD includes AutoLISP [2], and ArchiCAD includes GDL as well as support for other textual programming languages [7].

Unfortunately, achieving parametric design capability by attaching a textual programming language, creates a system which requires the designer to switch view between two very different representations of the design. Or since the programming and design activities are so different, users with different skills may be required.

In an attempt to create an integrated programming/design environment for parametric design which does not suffer from this problem, a Language for Structured Design (LSD) was proposed in [6]. LSD, having its roots in logic, provides a homogeneous view of design objects and the operations that transform them. LSD captures the notion of solids as high-level programming constructs in a programming/design environment, both semantically and syntactically, however, it does not deal with the low-level computations required for characterising the geometry of the solids. This issue was addressed in [5], which proposed a formal model for design spaces, solids and operations on solids, and the link established between this model and LSD.

One of the most significant features of LSD is its declarative problem-solving capabilities for solving problems related to parametric design. The integration of programming capabilities directly into the design space provides a homogeneous view of design components along with algorithms applied on them for parametrising, constraining, and simulation. In [4] we have reported on this aspect of LSD through a detailed example.

As noted in [6] an important aspect of the design of structured solids is the simulation of the *behaviour* of solids. Conventional programming languages are used for designing algorithms, the behaviour of which is specified by their structure. In general, however, the relationship between the behaviour of an object and its structure is not so obvious. In fact, an object may have several different behaviours of interest. For example, a mechanical engineer may be interested in both the rotational behaviour of the gears in a transmission, as well as their reaction to heat.

Simulating the behaviour of solids in a design environment calls for appropriate support both in LSD and in the solid modeler. In this paper, we will show how the formal model for solids presented in [5] can be extended to characterise behaviours.

In the following, if x and y are sequences, we denote the concatenation of x an y by $x \bullet y$. For example, if x is $x_1,x_2,...,x_n$ and y is $y_1,y_2,...,y_m$, then $x \bullet y$ means $x_1,x_2,...,x_n,y_1,y_2,...,y_m$. If x is a sequence and y is not a sequence, for consistency, by $x \bullet y$ we mean $x_1,x_2,...,x_n,y$ and by $y \bullet x$ we mean $y,x_1,x_2,...,x_n$.

If x is a sequence of length n and $1 \leq i \leq n$, we denote by $\bar{x}_i$ the sequence of length n-1 obtained by removing the i^{th} element from x.

2 Solid Modeling

At the heart of a visual design environment with programming capabilities must lie a solid modeler. Solid modeling deals with the creation, manipulation and interaction of solids in a design space. A solid is a mathematical representation of a 3D object which includes its geometrical specifications and may also include other properties such as mass, material, centre of gravity, elasticity, and electromagnetic characteristics.

While most solid modelers are capable of performing complex geometrical computations on solids, they do not support the computations necessary for specifying behaviours. Since our solution to this problem involves an extension to the definitions given in [5], we will briefly summarise the core concepts presented there. The reader is encouraged to consult [5] for a thorough discussion.

Solids are modeled in a *design space*. The design space is a normal 3D space augmented with an arbitrary but finite fixed number of real-valued *properties*. A solid is a function that maps a list of *parameter values* to a set of points in the design space constituting the volume of the solid. Therefore, each solid in the design space represents a family of solids, each of which realised by a particular choice of parameter values. Note that a solid in the design space cannot exist until all of its parameter values are set to specific values within their respective ranges. Following is a more precise definition of a design space and a solid, not restricted to three dimensions.

Definition 2.1: A *design space in m dimensions over r properties* for some integers $m \geq 0$ and $r \geq 0$ is the set of all subsets of $\mathbf{R}^m \times \mathbf{R}^r$.

Definition 2.2: A *solid* in a design space $\mathcal{D}$ in *n variables* for some $n \geq 0$ is a function Φ: $\mathbf{R}^n \rightarrow \mathcal{D}$ such that, if $\mathcal{D}$ is in $|v|$ dimensions and (v, p) and $(v, q) \in \Phi(y)$ for some $y \in \mathbf{R}^n$, then $p = q$.

A variable of Φ is an integer i such that $1 \leq i \leq n$.

The intuition behind this definition is that something we normally think of as a solid can be characterised by a set of points in space, where each point has unique values for all properties associated with it.

This definition also provides parametrisation by defining a solid as a function. A completely specified object in the design space is obtained by applying the function to a particular list of values. Hence a solid represents a family of objects.

Definition 2.3: If Φ and Ψ are solids in n and k variables respectively, Φ and Ψ are said to be *equivalent*, denoted $\Phi \equiv \Psi$, iff $\{S \mid S = \Phi(y),\ S \neq \emptyset,\ y \in \mathbf{R}^n\} = \{S \mid S = \Psi(y),\ S \neq \emptyset,\ y \in \mathbf{R}^k\}$.

This definition recognises that an object in a design space may have more than one representation as a solid (function). For example, consider a two dimensional design space over the three properties *temperature*, *colour*, and *material* where *temperature* is determined by *colour* and *material*, and *colour is* determined by *temperature* and *material*. A square solid in this space can be defined be a function Φ of variables ($b,c,l.\alpha,temperature,colour$) or by another function Ψ of variables ($b,c,d,e,material,colour$) as illustrated in Figure 1. While both Φ and Ψ define the same family of objects, they *expose* two different sets of parameters of the solid.

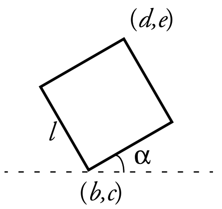

Figure 1: A **Square** in the design space

Example 2.4: To further illustrate the definition of a solid, let $\mathcal{D}$ be a design space in 2 dimensions over zero properties: then the function **Disk** is a solid in three variables which characterises the family of disks with radius r and centre at (b,c), where $\textbf{Disk}(b,c,r)=\{(x,y) \mid x,y \in \mathbf{R} \text{ and } (x-b)^2+(y-c)^2 \leq r^2\}$.

3 Behaviour

The word *behaviour* usually means the actions performed by some entity that we can observe: that is, a behaviour is a sequence of the states an object goes through over some period of time. For example, as a wheel rolls along the ground we observe its changing position.

The behaviour of an object need not be related to the passage of time, nor to the object's geometry. For example, the way an object reacts to heat is dependent on temperature and is not necessarily manifested by a change in position or size.

In general, therefore, behaviour involves the change of state that results from a change in some variable.

As noted above, a solid function actually represents a family of objects in a design space. Our definition of behaviour exploits this fact by taking the view that the family of objects represents all the states of one object, that can be observed as the value of a parameter is varied.

Definition 3.1: Let $\mathcal{D}$ be a design space, Φ be a solid in $\mathcal{D}$ in n variables and p be a variable of Φ. The *behaviour of* Φ *with respect to p* is the function ϕ_p: $\mathbf{R} \rightarrow (\mathbf{R}^{n-1} \rightarrow \mathcal{D})$ such that $\forall$y $\in \mathbf{R}^n$, $\phi_p(\text{y}_p)(\text{y}_{\bar{p}}) = \Phi(\text{y})$. Here $(\mathbf{R}^{n-1} \rightarrow \mathcal{D})$ denotes the set of all functions from $\mathbf{R}^{n-1}$ to $\mathcal{D}$. p is called the *controlling variable* of the behaviour.

Just as the definition of solid is a generalisation of the ordinary notion of solid, providing for parametrisation, our definition of behaviour is a generalisation of the ordinary notion of behaviour discussed above in that is applies to parametrised solids. In fact, given a particular value for the controlling variable, the behaviour picks out the family of objects that represent the states of the original family of objects corresponding to the chosen value for the controlling variable.

Example 3.2: Consider the **Disk** solid of Example 2.4. The behaviour of **Disk** with respect to its first variable is the function ϕ_1:$\mathbf{R} \rightarrow (\mathbf{R}^2 \rightarrow \mathcal{D})$ such that $\forall b \in \mathbf{R}$, $\phi_1(b)(c,r)$=$\{(x,y) \mid x,y \in \mathbf{R}$ and $(x\text{-}b)^2+(y\text{-}c)^2 \leq r^2\}$ $\forall c,r \in \mathbf{R}$. The controlling variable of this behaviour is the horizontal position of the centre of the disk. As this value varies, the state of the disk, that is, the particular selection of points that it consists of, moves horizontally. It is, of course, difficult to reconcile this notion of behaviour with our normal understanding of the term because we are dealing with a parametrised object. If we fix the values of all the variables except the controlling one, then we see that the behaviour in this example corresponds to the set of all possible positions of a disk of fixed radius, constrained to lie on a fixed horizontal line.

In the real world the behaviour of a solid may be defined with respect to parameters that are apparently not part of the definition of the solid itself. However, as we have noted above, there may be many different but equivalent solids corresponding to a particular family of objects in a design space. Each such solid may have variables which are not directly related to its geometry or properties, but are used by the function to determine them. We illustrate this as follows.

Example 3.3: Suppose that we would like the horizontal position of the centre of our disk in Example 2.4 to be computed from time t, horizontal speed v, and starting position b_0, so $b = b_0 + vt$. We can therefore represent our disk by the function **Disk'** where **Disk'**(v,t,b_0,c,r)=$\{(x,y) \mid x,y \in \mathbf{R}$ and $(x\text{-}b_0\text{-}vt)^2+(y\text{-}c)^2 \leq r^2\}$. Note that **Disk** and **Disk'** are equivalent. Clearly, we can now define a behaviour for the disk with respect to time, the first variable of **Disk'**.

As this example illustrates, to allow the definition of a behaviour for a solid to be defined with respect to a variable other than those over which the solid is defined, we need to be able to inject new variables into a solid, such as time in the disk example. This can be done by applying an operation to a solid that introduces new variables together with appropriate constraints. It is important to notice that injecting extra parameters into a solid does not change the fundamental nature of the family of objects the solid defines. The extra parameters just allow us to define relationships between variables that can be exploited to define a new behaviour by constraining the family to one of its subsets.

There are two possible ways for injecting new variables into a solid, both of which rely on the concept of operation defined in [5]. The first is to apply to the solid a unary operation that provides the necessary extra variables and constraint. The second method is to apply a binary operation to the target solid and a special solid which provides the required new variables.

Although the definition of operation in [5] allows us to introduce new variables to a solid using the second method above by using special solids, it cannot be used to directly inject new variables into a solid, as required by the first method. Accordingly, we provide a modified definition of an operation that accommodates the direct injection of new variables to a solid.

Definition 3.4: If $\mathcal{D}$ is a design space and n is a positive integer, an *n-ary operation in* $\mathcal{D}$ is a triple $(\mathcal{F}, \mathcal{L}, \mathbf{C})$ where

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

This definition of operation differs from that in [5] only in that it includes the extra variables. This allows an operation to inject new variables into its operands. This modification calls for minor changes to some other definitions from [5]. For example, the new variables need to be accounted for in the definition of the application of an operation to its operands.

Definition 3.5: If D $\in \mathcal{D}$ then $\downarrow$D = $\varnothing$ if $\exists$ (v, p), $(v, q) \in$ D such that $p \neq q$, otherwise $\downarrow$D = D.

Definition 3.6: 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 $t = e + \sum_{i=1}^{n} n_i$ variables, where e is the number of extra 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_e the first e elements of y, denote by y_1 the next n_1 elements of y, denote by y_2 the next n_2 elements of y and so forth, then we define

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

Again, this definition differs from that in [5] only in that it accounts for the extra variables, including them as variables of the solid computed by the operation.

Example 3.7: Let $\mathcal{D}$ and **Disk** be as defined in Example 2.4. An operation that defines a horizontal move behaviour for **Disk** can be defined by **H-Move** = $(I, \mathcal{L}, \mathbf{C})$ where I is the identity operator, $\mathcal{L} = (\mathbf{L}_1)$ where $\mathbf{L}_1(a,b,c)$ is valid iff a is a solid with a reference point (b,c), and $\mathbf{C}(v,t,b_0,b,c) = [b=b_0+vt]$. Here, v, t, and b_0 are the three extra variables of the operation representing horizontal speed, time and starting position respectively. By "reference point" we mean some point in a solid to which all other points can be related, such as the centre of a disk, or the left upper corner of a square. Applying **H-Move** to **Disk** creates a solid Ψ such that for all $v,t,b_0,b,c,$ and r, $\Psi(v,t,b_0,b,c,r) = \{z \mid z \in \downarrow\mathbf{Disk}(b,c,r) \text{ and } b = b_0+vt\}$. Substituting for **Disk**, we obtain $\Psi(v,t,b_0,b,c,r) = \{(x,y) \mid x,y \in \mathbf{R} \text{ and } (x-b_0-vt)^2+(y-c)^2 \leq r^2\}$. Since b no longer appears on the right hand side, this function can be reduced to **Disk'**, where **Disk'**$(v,t,b_0,c,r) = \{(x,y) \mid x,y \in \mathbf{R} \text{ and } (x-b_0-vt)^2+(y-c)^2 \leq r^2\}$. As noted in Example 3.3, **Disk'** has a behaviour with respect to variable t. When the other variables of the solid **Disk'** are fixed, this behaviour with respect to t is the set of disks with a fixed radius r anchored on a horizontal line $y=c$. This can be pictured as a disk that moves with speed v on a horizontal line $y = c$ where b_0 is the initial horizontal position of the centre of the disk at time $t=0$.

An operation may be sufficiently generic to be applicable to more than one kind of solid. For example, the **H-Move** operation in Example 3.7 can be applied to any solid that has a reference point, as illustrated below.

Example 3.8: Let **Box** be the solid over the design space of Example 3.7 defined as **Box**$(b,c,w,h)=\{(x,y) \mid b \leq x \leq b+w \text{ and } c \leq y \leq c+h\}$. Applying **H-Move** to **Box** creates the solid Ψ defined by $\Psi(v,t,b_0,b,c,w,h) = \{z \mid z \in \downarrow\mathbf{Box}(b,c,w,h) \text{ and } b = b_0+vt\}$. Inserting the definition of **Box** we obtain $\Psi(v,t,b_0,b,c,w,h) = \{(x,y) \mid x,y \in \mathbf{R} \text{ and } b \leq x \leq b+w \text{ and } c \leq y \leq c+h \text{ and } b = b_0+vt\}$, which can be reduced to the equivalent solid **Sliding-window** defined by **Sliding-window**$(v,t,b_0,c,w,h) = \{(x,y) \mid x,y \in \mathbf{R} \text{ and } b_0+vt \leq x \leq b_0+vt+w \text{ and } c \leq y \leq c+h\}$.

Although most behaviours of solids in the real world are with respect to time, the definition of a behaviour as presented here is sufficiently general to capture non-temporal behaviours.

Example 3.9: Let **Cylinder**$(b, c, d, v, h) = \{(x,y,z) \mid x,y,z \in \mathrm{R} \text{ and } (x-b)^2+(y-c)^2 \leq (v/h)\pi \text{ and } d \leq z \leq d+h\}$, then **Cylinder** defines a vertical cylindrical solid in a 3D design space $\mathcal{D}$ over zero properties where (b,c,d) is the centre of the base of a cylinder with volume v and height h. The behaviour of this solid with respect to its volume v is the function $\phi_4:\mathbf{R} \to (\mathbf{R}^4 \to \mathcal{D})$ such that $\phi_4(b,c,d,h)=\{(x,y,z) \mid x,y,z \in \mathrm{R} \text{ and } (x-b)^2+(y-c)^2 \leq (v/h)\pi \text{ and } d \leq z \leq d+h\}$. Note that changing the volume v will make a cylinder become thinner or thicker while its height and position remains intact.

As we mentioned earlier, there are two different ways for defining a new behaviour. We have presented one of them, namely, applying a unary operation that imposes new constraints on the parameters of a solid and possibly introduces new parameters if necessary.

In the second method, the new variables required for a behaviour, time for example, could be introduced by some special solid, and an associated special operation could extract the appropriate variables from the solid to which the behaviour is being attached and constrain them to variables supplied by the special solid. In fact, the special solid, rather than being just a technical device for accomplishing the result, could represent a physical controller, for example a slider like that in the QuickTime movie player, that varies time over an interval. The binary operation that modifies the target solid so that a required behaviour can be defined for it, provides the means for attaching such a slider to the target solid. The operation extracts an appropriate variable from the target solid, for example "angle to horizontal axis", and constrains it to the time variable from the slider in an appropriate way. Hence applying the operation to the special solid and target solid is analogous to wiring a slider to an interface element, as is done in some graphical user interface builders.

Example 3.10: Let $\mathcal{D}$ be a design space in 2 dimensions over zero properties. For simplicity, we represent points in $\mathcal{D}$ in clockwise polar coordinates, where the angle component is measured in degrees. Let **Beam**$(a,l) = \{(\alpha,r) \mid \alpha,r \in \mathrm{R},\ \alpha = a \text{ and } 0 \leq r \leq l\}$ be a solid in $\mathcal{D}$, a straight line at angle from the vertical a and length l with one end at the origin. Let **Arc** be a solid in $\mathcal{D}$ defined by **Arc**$(d,t) = \{(\alpha,1) \mid \alpha \in \mathbf{R} \text{ and } 0 \leq \alpha \leq$

$dt\}$, also in polar coordinates. Note that **Arc** defines an arc on the unit circle centred at the origin. We use this second solid to inject time (t) and a coefficient (d) into other solids. Now let **Add-time** = (a_1, $\{L_1(a_1,b), L_2(a_2,d,t)\}$, $[b = dt]$) be an operation with two operands a_1 and a_2, where $L_1(a, b)$ is valid iff a is a beam with angle b (the first parameter of **Beam**) and $L_2(a,d,t)$ is valid iff a is the arc **Arc**(d,t). Applying **Add-time** to the two operands **Beam** and **Arc** results in the solid **Hand**(l,d,t) $=\{(\alpha,r) \mid \alpha,r \in \mathrm{R}, \alpha=dt, 0 \le r \le l\}$. Applying **Add-time** to **Beam'** and **Arc'**, where **Beam'**(a) = **Beam**(a,1) for all a, and **Arc'**(t) = **Arc**(6,t) for all t, will create a solid **Hand'** such that **Hand'**(t) = **Hand**(1,6,t), representing a clock hand with unit length. When t represents time in seconds, **Hand'** exhibits a behaviour analogous to the behaviour of a clock's second hand. When t is in minutes, **Hand'** will behave in a fashion similar to clock's minute hand. Similarly, an application of **Add-time** to **Beam'** and **Arc"** where **Arc"**(t) = **Arc**(30,t) for all t, will create a solid **Hand"** such that **Hand'**(t) = **Hand**(1,30,t), which behaves like a clock's hour hand when t is in hours.

In order to create a clock with three hands, let **Pin** = ($a_s \cup a_m \cup a_h$, $\{L_1(a_s,t_s), L_1(a_m,t_m), L_1(a_h,t_h)\}$, $[t_s = 60t_m,\ t_m = 60t_h]$) be an in $\mathcal{D}$ with three operands a_s, a_m, and a_h, where $L_1(a,t)$ is valid iff a is a hand solid and t is its time variable. Applying **Pin** to three solids **Hand'**, **Hand'** and **Hand"** corresponding to the second, minute, and hour hands of a clock respectively, will create a solid **Clock** over $\mathcal{D}$ such that **Clock**(l_s,l_m,l_h,t) = $\{(\alpha,r) \mid \alpha,r \in \mathrm{R}$ and $[(\alpha=6t, 0 \le r \le l_s)$ or $(\alpha=t/10, 0 \le r \le l_m)$ or $(\alpha=t/120, 0 \le r \le l_h)] \}$

4 Defining Behaviours in a Design Environment

Having defined behaviours formally, we will now consider how they might be implemented in a design environment. Defining a new behaviour for a solid could be achieved in three possible ways.

First a programming language, such as the language underlying a solid modeler, could be employed to directly define a behaviour for a family of solids. We are at present investigating the viability of Lograph, the visual logic programming language on which LSD is based, as an underlying language for our proposed solid modeler and also for defining operations and behaviours on solids [3]. This calls for extending Lograph to support constraint specification and numerical computation beyond the scope of pure Lograph.

A second approach is to apply predefined operations to solids to obtain a desired behaviour. In this case, the design environment would be equipped with a set of icons, each representing a separate operation with its own selectors and constraint. Dragging and dropping such an icon on to a solid in the design space would trigger the application of the operation to the target solid, extracting the appropriate variables from the solid, adding necessary variables as described above, and applying the operation's constraint. This scheme relies on the extended definition of operation which adds necessary variables to a solid. A variation of this would rely on the other means for adding variables, by using special solids to supply them. In this case, the environment would provide sliders and other such controller solids which would be connected to the target solids by an appropriate wiring operation.

In the third, more direct approach, the designer explicitly adds new variables to a solid, then creates new relationships between the variables of the solid. Creating these new relationships between variables might be achieved by either using predefined generic constraints, or directly defining constraints by demonstrational techniques.

In the remainder of this paper we will discuss the third approach. In contrast to preceding sections, we will present the concepts informally, to provide a more intuitive description of how the formal notion behaviour might be implemented in a design environment. First, we will describe some basic requirements in a design space that are necessary for our discussion.

4.1 Solid Palette

A design environment is typically a 3 dimensional space in which a designer can create solid models by using a set of graphical tools and user interfaces. We aim to add features and capabilities to enable the creation of behaviours by demonstration.

Creating a new behaviour for a solid calls for access to the variables of the solid and also support for adding extra variables. To deal with these issues, we propose a *solid palette* that enables a designer to access variables, as well as other components that are significant during the process of programming solids.

A solid palette in the design environment contains information about a solid that can help a designer create new behaviours or allow the designer to observe how a solid performs in response to changes in its controlling variables. A solid palette contains a list of the solids variables, an Add Var button for adding new variables, a Training button for defining a new constraint on the variables of the solid by demonstration, and a Sample button.

The process of creating a new behaviour for a solid starts with defining a new set of variables for the solid, including the variable with respect to which the behaviour is to be defined. Adding a variable is achieved by double-clicking the solid to open its solid palette, then pressing the Add Var button on the palette. This opens another palette from which an appropriate control component for the parameter can be selected. This will add the new control component to the solid palette, as shown in Figure 2. The designer can name the parameter, and set certain other characteristics such as its size and shape, or its scale if it is a numerical selector such as a knob as shown in the figure. Note that adding a new variable will not have any effect on the solid until the parameter is bound to other parameters of the solid through some constraints. Figure 2 illustrates the solid palette of the **Beam** solid of Example 3.10.

4.2 Predefined Constraints

Constraints play a key role in the creation of new behaviours. A constraint imposes specific restrictions on the values that the variables of a solid can assume. New constraints can be added to a solid using a set of parametrised constraint tools in the design space.

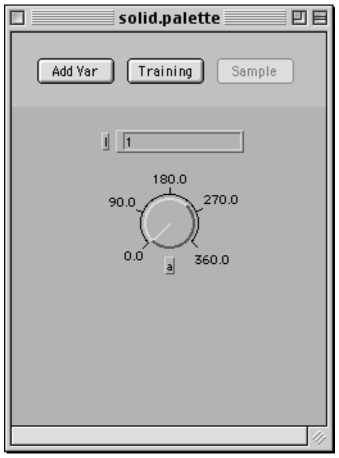

Figure 2: A Solid Palette.

Constraint tools are graphic icons that a designer can select and apply to solids. While some constraints may impose specific restrictions on the values that some of the variables of a solid can assume, others may well be parametrised. For example, an **Anchor** constraint may anchor a point to a specific coordinate location in the design space, while an **Anchor-along-line** constraint may constrain a point to lie on a parametrised line. The parameters of such a constraint can be set by a designer.

4.3 Defining Constraints by Demonstration

Programming by Demonstration (PBD) is viewed by many as an elegant way to create agents with behaviour by direct manipulation of representations of domain entities. PBD entails the use of concrete examples to teach an agent how to behave under various situations when there is an analogy between the current circumstances and those of the examples. A PBD system records the actions of the programmer and uses various generalisation techniques to infer the behaviours of the agent [1,8,9,10].

We believe that PBD can have two possible applications in a design environment; creating new operations on solids and creating new behaviours.

One major difference between PBD for creating automated agents and creating behaviours for solids is that in the former, an agent responds to some environmental changes while in the latter case, a solid does not have the notion of perceiving its surrounding environment. The behaviour of a solid is defined with respect to some of its own variables.

4.3.1 Inferring Constraints

In the case of agent behaviours, inferred behaviour is often represented as a set of rules, finite automata, or a grammar. In the case of solids, we aim to infer constraints that realise a behaviour. While there are different types of constraints, for simplicity we will focus our attention on linear constraints in order to demonstrate how constraint tools might operate in a design environment.

4.3.2 Inferring Linear Constraints

A linear constraint over *n* variables $x_0,x_1,...,x_{n-1}$ is an equation of the form

$$x_0 + a_1x_1 + ... + a_{n-1}x_{n-1} + a_n = 0$$

Given *n* solutions to this equation, we can solve the resulting set of *n* linear equations to obtain the coefficients $a_1,a_2,...,a_n$.

A tool that uses PBD to define constraints would employ an appropriate constraint solver, such as a simple linear equation solver in the case of a linear constraint. In order to create a linear constraint that realises a behaviour, the PBD tool needs to know which variables of a solid are also the variables of the constraint. The variables it can choose from are those new variables introduced by the designer as described above, and the current variables of the solid.

The constraint inference mechanism maintains a pool of variables to be included in the constraint, and in training mode, monitors all variables, both in the pool and not, checking for those the values of which have changed. If there is a difference between the current and the previous values of any variable, that variable will be added to the variable pool. At any point, the inference mechanism may deduce the constraint from the gathered data, if the data is sufficient. We now briefly describe how a tool implementing such a mechanism might operate.

In Example 3.10 we showed how special solids can be used to inject new variables into other solids and then constrained those solids with respect to the injected variables. Here we will show how a design space with PBD features can be used to define the same behaviour for **Hand** solids.

Example 4.4: In order to define a new behaviour for the **Beam** solid of Example 3.10 to obtain the solid **Hand**, we first introduce a new variable corresponding to time (*t*). A double click on the **Beam** solid in the design space opens the corresponding solid palette as shown in Figure 2, which already contains the variable *a*. Pressing the Add Var button opens another palette from which we choose a slider to represent the new variable. We name this new parameter *t* and adjust its limits to 0 and 60 as shown in Figure 3(b).

Clicking the Training button starts the definition of a new linear constraint for the solid, changes the name of the Training button to Solve, and enables Sample button. We set *t* to

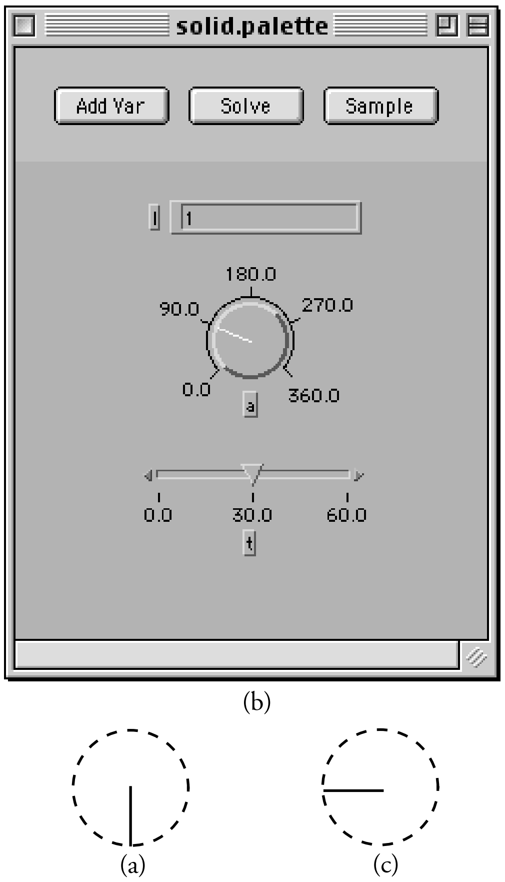

Figure 3: Creating a constraint

30 and in the design space, turn the **Beam** solid 90° clockwise to point towards the bottom of screen as shown in Figure 3(a), and click the Sample button. This adds both the *t* and *a* variables of **Beam** to the pool of variables maintained by the constraint engine, since the values of both are different from their values before the sampling. Figure 3(b) shows the solid palette at this point.

Next we set *t* to 45, turn the solid 180° to the left as in Figure 3(c) and click the Sample button again. At this point, the inference engine has collected two sample points (90°,30) and (180°,45) for the *a* and *t* variables. We now click Solve, initiating the solution of the linear constraint $a + a_1t + a_2 = 0$ where *a* corresponds to the angle of the **Beam** and *t* corresponds to time. Solving the system will give $a + 6t - 90 = 0$. Note that the constant -90 here corresponds to the phase difference between the base of the angles in a clock and that of the polar coordinate system that we have employed (clockwise). If we had pressed the Solve button after the first sampling, the system would have given us an *insufficient data* error message and would have asked us for more data samples.

The inferred constraint $a + 6t - 90 = 0$ is added to the definition of the **Beam** solid, creating a new solid that exhibits a behaviour similar to that of a clock's second hand. A similar approach can be taken to defining two more solids that correspond to the minute and hour hands of a clock.

5 Summary and Future Work

A formal definition for the behaviour of solids in a solid modeler has been presented. The definitions are consistent with the earlier proposal for solid modeling provided in [5] and are intended to provide support for simulation in design environments. We have also provide some informal insights into how a design environment might incorporate tools for building behaviours into structured objects.

We are currently investigating the viability of the visual logic programming language Lograph as the basis for a logic-based solid modeler, consistent with our definitions of design space, solid and behaviour. Such a solid modeler will provide the necessary modeling capabilities for our implementation of LSD.

The definition of behaviour we have presented above characterises a behaviour as variations in state resulting from changes in a single variable. This is probably sufficient for many practical design applications. However, it is important to note that behaviours with respect to two or more variables cannot be defined by the obvious generalisation of our definition. For example, if we consider a wheel rolling along the ground at a fixed speed (dependent on time) and shrinking and expanding (dependent on radius), then the horizontal position of the wheel will be determined notonly be the current values of time and radius, but by the sequence of states of the wheel during the observation of its behaviour. Although such behaviours (i.e. dependent on more than one variable) may seem to be pathelogically unimportant, they may be significant in certain design domains.

6 Acknowledgements

This research was supported Natural Sciences and Research Council of Canada Individual Research Grant OGP0000124.

7 References

[1] R. St. Amant, H. Lieberman, R. Potter, and L. Zettlemoyer. Visual Generalization in Programming by Example, *Communications of the ACM*, March 2000/ Vol. 43, No. 3, pp. 107-114

[2] Autodesk Inc. (1992) *Auto*LISP *Release 12 Programmers Reference Manual*.

[3] O. Banyasad, P. T. Cox, *Logic-Based Solid Modeling in a Visual Logic Programming Language*, Report CS-2002-?, Faculty of Computer Science, Dalhousie University, (2002).

[4] O. Banyasad, P. T. Cox, *Solving design problems in a logic-based visual design environment*, Report CS-2001-

04, Faculty of Computer Science, Dalhousie University, (2001).

[5] P.T. Cox, T. Smedley, A Formal Model for Parametrised Solids in a Visual Design Language, *Journal of Visual Languages and Computing*, 6(6), Academic Press (2000), 687-710.

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

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

[8] T. Lau. *Programming by Demonstration: a Machine Learning Approach*, PhD thesis, University of Washington, 2001.

[9] A. Michail. *Imitation: An Alternative to generalization in Programming by Demonstration Systems*, University of Washington, Technical Report UW-CSE-98-08-06.

[10] B. Myers and R. McDaniel. Demonstrational Interfaces: Sometimes You Need a Little Intelligence; Sometimes You Need a Lot. *Your Wish is My Command*. Henry Lieberman, Ed. San Francisco: Morgan Kaufmann, 2001. pp. 45-60.

[11] K. J. Waldron, Drafting a New Plan for Design, *Mechanical Engineering, Design Supplement*, November 1999, 37-38.