PrintTalk: A Language for Constraint-Based 3D Modelling

Jef Jacobs
Software Languages Lab, Vrije Universiteit Brussel, Belgium

Wolfgang De Meuter
Software Languages Lab, Vrije Universiteit Brussel, Belgium

Jens Nicolay
Software Languages Lab, Vrije Universiteit Brussel, Belgium

Abstract

Programmatic CAD (PCAD) is an emerging alternative to traditional visual CAD software. However, state-of-the-art PCAD tools have limited or no support for constraints. Consequently, these tools depend solely on parametrisation for variability, reusability, and composition of shapes. This leads to problems such as parameter explosion, leaky compositional abstraction, and prevents a declarative approach to defining spatial patterns (linear, grid, circular, etc.) for the constituents of a composition. This paper describes the design of PrintTalk, a PCAD language that supports 3D modelling by composing shapes and expressing relations between them using first-class constraints. Evaluating PrintTalk against state-of-the-art PCAD tools demonstrates that its expressive abstraction and composition mechanisms facilitate the design and promotes the reuse of shapes.

2012 ACM Subject Classification Software and its engineering → Domain specific languages; Software and its engineering → Multiparadigm languages; Software and its engineering → Constraints

Keywords and phrases Programmatic 3D Modelling, PCAD, Domain specific language, Constraints

Digital Object Identifier 10.4230/LIPIcs.CP.2025.16

Supplementary Material *Software*: `https://doi.org/10.5281/zenodo.15593661`

1 Introduction

Computer-aided design (CAD) systems are central to the modern product design process, in which physical products are represented as digital CAD drawings. Designing digital 3D models often involves sophisticated CAD software, of which the vast majority is *visual* in nature. Visual CAD tools typically have built-in constraint mechanisms to aid designers in taming complexity. For example, *geometric constraints* [21] (parallelism, concentricity, etc.) control relationships between components, while *dimensional constraints* (length, radius, etc.) help specify precise dimensions of shapes in relation to each other. Designing parametrised components of which the parameters can subsequently be modified to adjust the entire design accordingly is called *parametric modelling*.

While visual CAD approaches can be accessible, intuitive, and well-suited for rapid prototyping, they lack several key advantages of programmatic approaches. Programmatic approaches are more amenable to software engineering techniques, enabling finer-grained control over the design process, facilitating the reuse of complex shapes, and offering richer automation, version control, and collaboration features. Consequently, *programmatic CAD* (PCAD) has emerged as a compelling alternative, where 3D models are constructed programmatically using parametric modelling. In addition to the inherent benefits of a code-based approach, PCAD leverages the full power of a programming language (variables, functions, loops, conditionals, modules, etc.), providing rich and expressive abstraction and composition mechanisms. This allows designers to think about 3D modelling at a higher

level, using custom abstractions tailored to their specific needs. For this reason, over the past few years there has been a growing interest in PCAD [10, 17], leading to the development of several PCAD libraries [5, 19, 14] and languages [13, 18].

1.1 Problem

Despite the benefits of a programmatic approach, a significant shortcoming of the state-of-the-art PCAD tools we surveyed is that, unlike their visual counterparts, they have limited or no support for constraints. In the absence of constraints, PCAD tools must *exclusively* rely on parametrisation for variability, reusability, and composition of shapes. In all but the most trivial designs, this reliance on parameters leads to three significant problems.

Parameter explosion. To achieve variability and reusability, designers must expose numerous parameters. As the complexity of the design and the points of variation increase, the number of parameters increases dramatically. Additionally, the programmer is responsible for assigning valid values to all parameters. Any dependencies that may exist between parameters remain implicit, which makes assigning valid values even more difficult.

Leaky compositional abstraction. Composing a shape out of other shapes using only parameters requires exposing internal details as "composition parameters" of constituent shapes whose only purpose is to make composition possible. For instance, to align a hinge with a door, the screw hole positions (normally an internal detail of the hinge) must be exposed as parameters. This breaks encapsulation, as programmers must track intricate details about composition hierarchies, hindering high-level reasoning and modular design. The need for composition parameters also contributes to parameter explosion.

Imperative compositional specification. Parameters alone cannot declaratively express the complex and reusable relations between components. Therefore, creating patterns (linear, grid, circular, etc.) without constraints requires imperative code using sequences and loops, coupling the creation of components with their arrangement. This results in inflexible, non-extensible compositions that cannot be properly abstracted over.

1.2 Solution

This paper introduces PrintTalk, a PCAD language that addresses the limitations of existing tools by tightly integrating constraint solving with parametric modelling. PrintTalk features three main concepts: parametrised shapes, constraints, and patterns (which are specialized constraints for spatial relationships). 3D modelling in PrintTalk involves programmatically composing shapes and expressing relations between them. Each shape definition declares a set of parameters that can partake in constraints and through which the shape can be modified. While shapes and constraints make up the core of the language, PrintTalk also features expressions that act as abstraction mechanisms and syntactic sugar for defining, instantiating, and composing shapes and constraints in a scalable and expressive manner.

PrintTalk overcomes the shortcomings of existing PCAD tools that we identified by meeting the following three criteria.

- **Criterium 1: Flexible variability and reusability.** PrintTalk shapes expose named parameters to which values can be assigned explicitly by the program, in which case they act as *constructor parameters*. Parameters that are not explicitly assigned a value become *constraint variables* whose values are determined by the constraint solver when running the program. Constraints and constraint variables provide a declarative and relational way to define shapes in PrintTalk. These declarative relationships, together with a reduced need for specifying explicit parameter values, avoid the problem of parameter explosion and provides a clean way to balance flexibility and complexity in a design.

- **Criterium 2: Flexible composition.** PrintTalk enables flexible composition by allowing constraints to be placed on the parameters of constituent shapes. A shape can constrain its own parameters and the parameters of its constituents. This avoids requiring detailed knowledge of a shape's internal structure before it can be composed with other shapes in a valid manner. PrintTalk features a set of primitive constraints (e.g., "equal", "greater than", etc.), but also allows users to define their own constraints as a combination of other constraints. For example, users can define their own "on-top-of" or "touching" constraints for composing their shapes.
- **Criterium 3: Declarative and reusable compositional abstraction.** PrintTalk decouples the logic for creating multiple shapes from arranging them in a particular spatial configuration. Multiple copies of shapes can be declaratively created by means of a `for` loop. In a separate step, multiple shapes can be arranged by adding constraints on their parameters. Because constraints are first-class citizens in PrintTalk, they can be composed and named, making them reusable. In PrintTalk, these reusable (compound) composition constraints expressing spatial arrangements are called *patterns*.

1.3 Overview

Section 2 introduces the PrintTalk language, detailing its core components and design principles. Section 3 surveys related work in programmatic CAD and constraint-based systems. Section 4 presents an evaluation of PrintTalk, comparing it to two state-of-the-art PCAD tools and demonstrating how it addresses the problems outlined in Section 1.1.

2 Overview of PrintTalk

This section introduces PrintTalk, a domain-specific language designed for constraint-based 3D modelling. PrintTalk enables the construction of complex 3D shapes by composing simpler shapes and expressing relations between them using constraints. To promote reusability, the language provides components for defining and instantiating new shapes and constraints.

The following subsections detail the fundamental components of the PrintTalk language. First, the concepts of shapes, parameters, instantiation, and the structure for defining composite shapes using scripts and constraints are introduced (Section 2.1). The core mechanism behind PrintTalk's flexible, solver-driven composition in which parameters function dually as constructor arguments or constraint variables, is explained (Section 2.1.1). This discussion covers implicit parameters used for relative positioning (Section 2.1.2) and the language's scoping rules (Section 2.1.3). Next, composite constraints (Section 2.2.1) and spatial patterns (Section 2.2.2) are presented as mechanisms for structuring and reusing constraints. Finally, the process for generating a 3D model as output is described (Section 2.3), along with the integration details of the Z3 constraint solver (Section 2.3.1) and the methodology for determining the constraint hierarchy (Section 2.3.2). A complete overview of PrintTalk's syntax is provided in Appendix A.

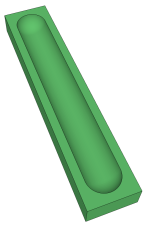
2.1 Shapes and Constraints

In PrintTalk, a shape has a name and a set of named, shape-specific parameters, such as dimensions or quantities. The language provides built-in primitive shapes, including cuboids, cylinders, cones, and spheres.

A shape is instantiated by referring to it by name and providing it zero or more values for its named parameters. For example, the expression `(sphere #:radius 10)` instantiates a sphere with radius `10`.

Listing 1 A PrintTalk program (left) and its resulting shape (right).

```
1  (shape: capsule (h d)
2   (script:
3    (named: cyl (cylinder #:diameter d #:height (- h d)))
4    (named: sp1 (sphere #:diameter d #:z (/ (- h d) -2)))
5    (named: sp2 (sphere #:diameter d #:z (/ (- h d) 2))))
6   (constraints: (assert: (>= h (+ 10 (* 2 d))))))
7
8  (shape: tray (l w)
9   (script:
10   (named: blk (cuboid #:length l #:width w #:height (/ w 2)))
11   (cut: (named: caps (capsule #:h (- l 10)
12    #:z (/ blk.height 4) #:rot-y 90))))
13  (constraints: (assert: (= caps.d (- blk.width 10)))))
14
15 (print: (tray #:l 150 #:w 30) "tray.stl")
```



New shapes are defined as a composition of other primitive or composite shapes using the `shape:` keyword. PrintTalk represents composite shapes as trees, where leaf nodes are primitive shapes.

```
1 (shape: name (<parameters>)
2  (script: <constituents>)
3  (constraints: <assertions>))
```

The `script:` section is used for instantiating constituent shapes and defining their basic composition. By default, shapes instantiated within a script are composed by means of a union, unless specified otherwise by a `(cut: <shape>)` operation, in which case the shape provided as argument to the `cut:` statement is cut from the preceding shapes in the script.

The order in which constituent shapes are instantiated within a script matters when a `cut:` operation is present in the script, as `cut:` operations are not commutative. The shape provided as argument to a `cut:` operation is only cut from previously instantiated shapes, leaving shapes instantiated after the `cut:` unaffected. A constituent shape can be given a name using a `(named: <name> <shape>)` statement, so that it can be referenced using that name. Listing 1 shows a PrintTalk program and its resulting shape, a pencil tray. Two composite shapes are defined: a capsule (lines 1–6) and a tray (lines 8–13). The pencil tray is obtained by cutting a capsule shape `cap` from a cuboid (or "block") shape `blk` (lines 9–12). A capsule itself is a composition of a cylinder with spheres at its top and bottom (lines 2–5).

The `constraints:` section in a `shape:` definition specifies the constraints used between the parameters of that shape. For example, the constraint on line 6 in Listing 1 sets a minimum height for the capsule shape based on its diameter to make sure the capsule has an "elongated" shape. Constraints are also used for defining the layout and precise relationships between constituent shapes (Section 2.1.1). An example of this is the equality constraint on line 13, which defines the relationship between the diameter of the capsule and the width of the block shape, ensuring that the diameter of the cutout capsule shape is smaller that the width of the block shape. PrintTalk provides **built-in primitive constraints** for comparing values (i.e., $=$, $<$, $<=$, $>$, $>=$).

Like shapes, constraints can also be composed as described in Section 2.2.1.

2.1.1 Constraint Variables and Composition

The expressive power of PrintTalk stems from the interaction between shape parameters and constraints. A shape parameter can function in two ways: as a *constructor parameter* if explicitly assigned a value during instantiation, or as a *constraint variable* if left unspecified. In the latter case, its value is determined by the constraint solver based on the relationships defined in the program (Section 2.3.1).

Consider the built-in `sphere` shape definition, which declares `radius` and `diameter` parameters linked by the constraint `(= diameter (* 2 radius))`. If a sphere is instantiated with only a diameter (as on lines 4 and 5 in Listing 1), the solver infers the radius, and vice versa. This bidirectional capability also allows for validation: if both parameters are provided with inconsistent values (e.g., `(sphere #:radius 10 #:diameter 23)`), PrintTalk signals a runtime error, as the constraint is violated.

Fundamentally, PrintTalk treats all parameters as constraint variables. This unified view enables flexible instantiation: programmers can provide explicit values for some parameters (which corresponds to adding an equality constraint), while allowing the solver to derive others based on declared constraints. This significantly promotes component reusability.

This constraint-based mechanism of deriving unspecified values extends naturally to shape composition. Relations between constituent shapes are expressed declaratively using constraints on their parameters, as demonstrated by the relation between the capsule diameter and block width in the tray example (Listing 1). As another example, to ensure two surfaces glue together correctly, constraints can enforce that their corresponding dimensions match, allowing the constituents to adapt to each other. PrintTalk's extensible constraint system facilitates defining such complex spatial relationships.

The bidirectional capabilities of constraint solvers make constraints a powerful tool for expressing relations between shapes. Its ability to both find suitable parameter values and check the consistency of provided ones enables versatile composition strategies that are difficult to achieve with purely manual parameter assignment. Without constraints, programmers must imperatively calculate and assign values for all parameters, often leading to less flexible and reusable designs.

However, not all parameter values can be derived from constraints, particularly when a parameter's value is required structurally *before* constraint solving commences, or when it defines the set of shapes upon which a constraint will operate. In such cases, PrintTalk requires these values to be provided explicitly during instantiation. It is therefore possible to *force* parameters to act strictly as constructor parameters by preceding their name with an exclamation mark in the parameter list. As illustrated later, the `!nr-teeth` parameter in the gear shape definition (Listing 2) must be explicit because it determines the number of teeth to instantiate within a loop. Similarly, the `!shapes` parameter in the circular spatial pattern (Listing 2) must be provided, as it specifies the collection of shapes the constraint acts upon. If an argument for such a forced constructor parameter is omitted during instantiation, PrintTalk raises a runtime error instead of treating it as a constraint variable whose value can be derived.

2.1.2 Relative Position and Rotation

In PrintTalk, every shape definition implicitly contains six parameters that determine its 3D position (`x`, `y`, `z`) and rotation (`rot-x`, `rot-y`, `rot-z`) *relative* to its parent. Root shapes, which have no parent, have an absolute position and rotation of `(0, 0, 0)`. Implicit parameters behave identically to explicit parameters, including how they are bound.

For each implicit parameter, an implicit soft constraint is asserted stating that the values of these constraint variables is `0` (i.e., equal to the parent's value). The underlying solver aims to find a solution that satisfies these soft constraints, but, as discussed in Section 2.3.1, no error is raised when no such solution exists. Default values for implicit parameters implements a flexible environmental acquisition mechanism [11] that enables expressing relative positions and rotations of constituent shapes only when required, while also combatting parameter explosion. This mechanism is illustrated by the capsule shape definition in Listing 1. A capsule shape is defined as a combination of a cylinder and two spheres. The z-component of the spheres is specified, while the other position and rotation parameters are not specified and thus acquired from the parent shape, meaning their offset is set to `0` through implicit soft constraints.

2.1.3 Scoping Mechanism

A shape's parameters are accessible throughout the entire body of its definition (the `script:` and `constraints:` sections). Parameters of named constituent shapes (created with `name:`) within a shape definition's script are visible in subsequent script statements and in the `constraints:` section. These parameters must be referenced using dot notation. However, shapes cannot directly access the parameters of their *grandchildren* (the constituents of their constituents). Composite shape definitions can introduce parameters that are bound to a selection of their constituents' parameters. This allows a grandparent shape to influence its grandchildren through intermediate parent parameters. Therefore, while PrintTalk's scoping rules promote information hiding, programmers retain control over which parts of a composite shape's constituents are exposed upon reuse.

To illustrate this scoping mechanism, consider the PrintTalk program in Listing 1 again. The composite shape definition `capsule` declares parameters `h` (height) and `d` (diameter) (line 1). These parameters are used in the script and constraints of the capsule shape definition. Parameter `d` is bound to the `diameter` parameters of the constituent cylinder and spheres in the capsule's script (lines 2–5). Next, a capsule is instantiated and named `caps` in the script of the `tray` shape (lines 11–12). The capsule's `d` parameter is referenced in the tray's constraint as `caps.d` (line 13). However, the `tray` shape cannot access parameters of the `capsule`'s constituents, such as `caps.sp1.diameter`.

2.2 Constraint Composition and Spatial Patterns

2.2.1 Composite Constraints

Not only shapes, but also constraints can be composed to express more high-level compositions such as "on-top-of", "touching", or "matching-screw-holes". Composite constraints are defined using the `constraint:` keyword followed by a name, a set of parameters, and **assertions of primitive or composite sub-constraints**.

```
1  (constraint: name (<parameters>) <assertions>))
```

Just like in the `constraints:` section of `shape:`, **the asserted sub-constraints are joined by logical conjunction**. Unlike composite shape definitions with only `constraints:` but without a `script:`, a composite constraint definition does not create a shape, and is purely used for combining constraints. As an example, the `valid-edge` composite constraint ensures that the edge between an inner and outer diameter has a value between 2 and 20.

```
1 (constraint: valid-edge (innerDia outerDia edge)
2  (assert: (= edge (- outerDia innerDia)))
3  (assert: (>= edge 2))
4  (assert: (<= edge 20)))
```

Instantiating composite constraints is similar to instantiating shapes, using the same syntax for binding parameter values, as discussed in Section 2.1. Composite constraints can be defined or called without specifying named parameters. However, PrintTalk's primitive comparative constraints, such as `=` and `<`, do not use named parameters. Instead, they operate on positional arguments, all of which are required.

2.2.2 Spatial Patterns Expressed as Constraints

Often, multiple shapes need to be arranged in a predefined pattern, such as the teeth of a gear in a circular pattern. PrintTalk's functionality for expressing spatial patterns is split into two parts: instantiation and positioning.

PrintTalk includes `for` loops through which multiple copies of a certain shape can be instantiated. A `for` loop contains an identifier, an iterable and a body. The iterable can either be a range of numbers or a shape. For each iteration of the loop, the identifier is bound to each number within that range, or each constituent of that shape.

```
1 (for <identifier> in <iterable>
2  <body>)
```

The shapes instantiated in the body of a `for` loop within a shape definition's script are composed with other shapes in the script by means of a union, unless specified otherwise by a `cut:` operation. To promote expressivity, `for` loops introduce pseudo-variables `i` and `n` representing the index and the total number of items to iterate over, respectively.

PrintTalk supports *anonymous shapes* that can be defined and instantiated in the script of other shapes. Hereby, the constituents instantiated in the body of the anonymous shape definition are "grouped" within a new shape, facilitating passing them as arguments to constraints. Apart from the absent name between the `shape:` keyword and the parameter list, the syntax for defining anonymous shapes is identical to defining named shapes.

Instantiating anonymous shapes is also identical to instantiating named shapes, by stating the shape and providing zero or more values for its named parameters. As an example, Listing 2 defines a gear (lines 1–11), in which a shape representing a single tooth is instantiated multiple times using a `for` loop (lines 7–8). These teeth are grouped within an anonymous shape definition without explicit parameters. The anonymous shape is instantiated (without arguments), and the instance is named `teeth` using a `named:` statement (lines 6–8).

To eliminate the need for repetitive manual positioning, PrintTalk allows spatial patterns to be expressed as composite constraints that place constraints on the positions of each of the involved shapes.

As an example, Listing 2 defines the `circular` pattern (lines 13–17) that is used for positioning the teeth of the gear. By means of a `for` loop, each shape in the pattern has its position along the x- and y-axes constrained, together with its rotation along the z-axis.

2.3 Output Statements

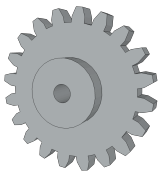
PrintTalk currently defines one output statement `print:` used for generating a 3D model and exporting it in the STL format.

Listing 2 Instantiating shapes multiple times using `for` loops in PrintTalk.

```
(shape: gear (!nr-teeth dia)
 (script:
  (cylinder #:diameter dia #:height 4)
  (cylinder #:diameter (/ dia 2) #:height 4 #:z 4)
  (cut: (cylinder #:diameter 4 #:height 8 #:z 2))
  (named: teeth ((shape: ()
                  (script: (for _ in (range nr-teeth)
                            (tooth #:l 4 #:w 3 #:h 4)))))))
 (constraints:
  (assert: (circular #:shapes teeth
                     #:radius (* (/ dia 2) 1.05)))))

(constraint: circular (!shapes radius)
  (for shape in shapes
   (assert: (= shape.x (* (cos (* (/ 360 n) i)) radius)))
   (assert: (= shape.y (* (sin (* (/ 360 n) i)) radius)))
   (assert: (= shape.rot-z (* (/ 360 n) i)))))

(print: (gear #:dia 32 #:nr-teeth 20) "gear.stl")
```



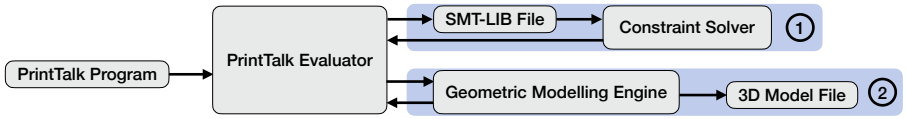


Figure 1 The PrintTalk toolchain.

```
(print: <shape> <filename>)
```

The `print:` statement conceptually ties together all necessary components for 3D model generation. When a shape is printed, that shape and all its constituents are instantiated, their constraints are checked and solved so that all constraint variables have valid values, and finally the 3D model is generated. Currently, `print:` is the only imperative, impure operation in PrintTalk, and is required to make the language practically usable. Imperative operations are executed solely for their side-effects and cannot participate in compositions with shapes or constraints.

2.3.1 Integrating the Z3 Solver

Constraints introduced in a PrintTalk program are solved using the Z3 solver [6]. To do so, the constraints are first compiled into an SMT-LIB file [2] that serves as the input for Z3. After solving these constraints, Z3 returns an S-expression containing the values to be assigned to the constraint variables. This process corresponds to the first part of the PrintTalk toolchain, depicted in Figure 1. Once all constraint variables have a value assigned to them, PrintTalk continues by building the 3D model to be exported as an STL file. PrintTalk makes use of the *Open CASCADE Technology (OCCT)* geometry library for generating STL files. This process corresponds to the second part of the PrintTalk toolchain.

Z3 [6] was selected primarily because it is an open-source solver supporting non-linear constraints, a capability lacking in considered alternatives like Cassowary [1]. PrintTalk actually uses a version of Z3 to which minor modifications were made in order to better

Table 1 Performance metrics of PrintTalk's constraint solving and 3D model generation processes.

3D model	Solving time	STL generation time
Pencil tray (Listing 1)	96 ms	259 ms
Gear (Listing 2)	511 ms	3464 ms
Lamp (Section 4)	2087 ms	15646 ms

suit PrintTalk's requirements. PrintTalk employs a version of Z3 extended via its C++ API to efficiently process both non-linear constraints and the soft constraints generated by the relative positioning mechanism (Section 2.1.2). This extension handles hard constraints first, then iteratively tests and adds satisfiable soft constraints in order of decreasing weight. This approach implements a *locally-predicate-better* comparator for determining the *constraint hierarchy* [4]. For performance considerations, our approach does not aim to maximise the number of constraints of the same weight that can be satisfied. Instead, constraints are added in the specific order that they are retrieved from a PrintTalk program (as described in Section 2.3.2). Soft constraints are rejected when they cannot be satisfied due to incompatibility with previously added (soft) constraints.

Performance is currently not the primary concern for PrintTalk. However, the constraint solving time (part 1 of the toolchain) is typically exceeded by the STL generation time (part 2 of the toolchain), suggesting the solver performance is not a limiting factor. Table 1 visualises the constraint solving and STL generation times for the 3D models described in this paper, measured on an M1 MacBook Pro.

2.3.2 Determining the Constraint Hierarchy

PrintTalk establishes a *deterministic* sequence for constraints based on a depth-first traversal of the shape tree (Section 2.1). Constituent shapes within a node are visited according to their instantiation order in the parent's script. During traversal, upon visiting a shape, its explicit (hard) constraints are first appended to the sequence in their specified order. Subsequently, the implicit (soft) constraints for default position (`x`, `y`, `z`) and rotation (`rot-x`, `rot-y`, `rot-z`) parameters are appended in that specific order. The weight associated with the soft constraints of constituent shapes increases by 1 relative to their parent, making root node constraints the weakest and leaf node constraints the strongest.

Upon solving the constraints, PrintTalk sorts the soft constraints by weight using a stable sorting algorithm, ensuring that the order in which the constraints were originally added to the list of constraints is respected. These rules implement a clear semantics that helps reasoning over PrintTalk programs and ensures that the output of a PrintTalk program corresponds to the programmer's expectations. For example, programmers can anticipate the eventual positioning of constituent shapes, knowing that soft constraints on the positions of earlier instantiated constituents have precedence over subsequent constituents.

3 Related Work

This section situates PrintTalk relative to existing approaches. First, we review the state of the art in programmatic CAD (PCAD) tools, noting their general lack of robust constraint mechanisms which motivates PrintTalk's design (Section 3.1). Given the limited approaches for constraint-solving in PCAD, we explore the extensive body of work on constraints within the related field of 2D layout planning (Section 3.2). Finally, we cover object-constraint languages which, similar to PrintTalk, exhibit a multi-paradigm nature by combining declarative constraints with object-oriented or imperative features (Section 3.3).

3.1 Programmatic CAD Tools

Programmatic CAD tools exist both as stand-alone domain-specific languages (DSLs) and as libraries to general-purpose languages. This section describes OpenSCAD and CadQuery, representing the state of the art in PCAD DSLs and libraries, respectively.

3.1.1 OpenSCAD

OpenSCAD [18] is a stand-alone PCAD language that represents shapes as a tree structure, where the root node represents the complete 3D shape and other nodes represent constituents that are composed by means of set-theoretic operations (union, difference, or intersection). PrintTalk and OpenSCAD therefore employ a similar modelling technique that is based on instantiating shapes and composing instances into more complex shapes. However, OpenSCAD lacks bidirectional constraints capable of both generating valid parameter values and validating provided ones. It only offers `assert` statements to check if parameter values meet specified requirements, for example when a dimension may not exceed a certain length. Apart from the declarative definition of 3D shapes, OpenSCAD also includes `if` statements and `for` loops through which 3D models can be constructed imperatively.

3.1.2 CadQuery

CadQuery [5] is a PCAD library for Python. Compared to PrintTalk and OpenSCAD, CadQuery provides a different way of modelling that relies more on extruding 2D surfaces to 3D shapes. For example, instead of directly instantiating a cylinder, first a circle on plane must be drawn, followed by an extrusion operation to form a 3D cylinder. CadQuery [5] only supports a limited set of geometric constraints [21], and the functionality of these constraints is limited as Python is not a constraint-based language. While programmers can import a constraint-solving library for Python and manually implement constraints, we consider this beyond the scope of CadQuery's inherent functionality. Another limitation of CadQuery is that the programmer must keep track of all constraints that are added within a design and make sure that all constraints are solved at appropriate moments by invoking the `solve()` method.

3.2 Constraints for 2D Layout

Constraint-solving techniques are frequently applied in domains requiring the precise spatial arrangement of 2D objects, such as user interface design and graph layout.

3.2.1 UI Design

A plethora of related work describes tools and approaches for developing UI layouts using constraints that specify how elements should be arranged on a screen [3]. An example of this is Apple's UIKit, which integrates the Cassowary solver [1] to support constraints for designing interfaces of iOS applications. Another layout engine [16] allows UI layouts to be described using ordinal and linear constraints, so that the UI can be dynamically adjusted by resolving constraints. Other related work includes ALM [15], a constraint language through which UI elements can be constrained using both hard and soft constraints. Similar to PrintTalk, ALM makes use of soft constraints and constraint hierarchies to determine which constraints have precedence over others in order to find the most optimal layout.

3.2.2 Graph Layout

Another related domain concerned with positioning elements in a 2D space is the graph-layout domain. Several algorithms have been developed to represent 2D graphs in an optimal manner, where constraints are used for arranging the nodes and edges of graphs on a 2D canvas in such a way that none of the nodes overlap and none of the edges cross each other. One such algorithm takes suggested node positions and constraints on these positions as input [12]. The algorithm aims to find a solution in which the nodes are as close to their suggested positions as possible, while satisfying all constraints placed on these positions. The suggestions for node positions have an associated weight, and the algorithm prioritises suggestions with a higher weight. Another algorithm uses constraints for expressing aesthetic criteria such as symmetry and alignment [7]. These constraints also have an associated weight through which the graph layout's aesthetic can be optimized.

3.3 Object-Constraint Languages

Existing research explores languages that integrate both concepts of object-oriented programming and constraints. Two such languages, s-Comma [20] and Babelsberg [8], share many similarities with PrintTalk. PrintTalk shapes are similar to classes in s-COMMA, as constraints can be placed on the parameters of instances of s-Comma classes, but can also be part of the class definition. Babelsberg [8] is another object-constraint language that combines imperative programming with constraints. Babelsberg is implemented by extending Ruby with constraints and allowing variables to partake in these constraints. Upon evaluating a Babelsberg program, the interpreter switches between an imperative evaluation mode and a constraint evaluation mode at appropriate times. PrintTalk has similar characteristics, as a shape definition's script is evaluated first, after which all constraints are collected and solved so that values can be assigned to the constraint variables.

4 Evaluation

We evaluate PrintTalk against OpenSCAD and CadQuery, representing the state of the art in stand-alone PCAD languages and libraries. Our evaluation focuses on how each tool addresses the problems outlined in Section 1.1: parameter explosion, leaky compositional abstraction, and imperative compositional specification. We compare PrintTalk, OpenSCAD, and CadQuery programs for modelling the same 3D model – a desk lamp illustrated in Figure 2 – with the same level of detail and features, and offering similar control over shape parameters. The lamp is composed out of a lamp shade, a base, arms, and caps for the hinge points. Each program uses the most suitable features that each tool offers for modelling the lamp. The complete code listings are available as supplementary material,[1] and only the most relevant snippets are included in this section. Appendix B contains an extended evaluation of PrintTalk, in which two additional 3D models are covered.

4.1 Criterium 1: Flexible Variability and Reusability

PrintTalk

PrintTalk permits parametrised components, such as shapes and composite constraints, to be instantiated with only a subset of their parameters explicitly assigned values. The constraint solver subsequently determines values for the remaining unbound parameters.

[1] https://doi.org/10.5281/zenodo.15593661

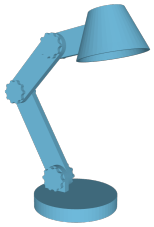


Figure 2 3D model of a lamp used for the evaluation of PrintTalk.

Listing 3 PrintTalk constraints through which valid parameter values can be determined.

```
40 (shape: lamp-base (dia height screw-dia screw-x screw-y screw-z mnt-l mnt-w mnt-th)
41  (script:
42   (named: base (cylinder #:diameter dia #:height height))
43   (cuboid #:height mnt-l #:width mnt-w #:length mnt-th
44    #:x (/ mnt-th -2) #:y (- (/ dia 2) (+ (/ mnt-w 2) 10)) ... )
45   (cylinder #:diameter mnt-w #:height mnt-th
46    #:x (/ mnt-th -2) #:y (- (/ dia 2) (+ (/ mnt-w 2) 10)) ... )
47   (cut: (named: sh (cylinder #:diameter screw-dia #:height mnt-th
48    #:x (/ mnt-th -2) #:y (- (/ dia 2) (+ (/ mnt-w 2) 10)) ... ))))
49  (constraints:
50   (assert: (>= mnt-l (/ mnt-w 2)))
51   (assert: (>= dia (+ mnt-w 5)))
52   (assert: (>= height 5))
53   (assert: (= screw-x (- x (/ mnt-th 2))))
54   (assert: (= screw-y (+ y (- (/ dia 2) (+ (/ mnt-w 2) 10)))))
55   (assert: (= screw-z (+ z (/ height 2) mnt-l)))))
```

Constraints incorporated into a component's definition serve to guide the solver towards valid solutions, for example ensuring positive dimensions for shapes. These constraints also improve reusability by validating explicitly provided parameter values, triggering an error if a value violates an established condition. Responsibility for defining adequate constraints rests with the programmer, as PrintTalk's validation capability is confined to the constraints specified within the component definition.

As an example, Listing 3 defines a shape named `lamp-base` containing parameters `dia` and `height` representing the diameter and height of the base of the lamp, and constraints are in place to set minimum dimensions (lines 51–52).

OpenSCAD

Parametrised OpenSCAD shapes can be defined using the `module` keyword and can, similar to PrintTalk, also be instantiated with optionally bound parameters. Unbound parameters in OpenSCAD get an `undef` value, and a warning message is raised when `undef` values are used within the body of the shape's definition. Like PrintTalk, this body can contain assertions for ensuring that parameter values are valid. Assertions in OpenSCAD take expressions representing conditions as argument, and raise an error when these conditions cannot be satisfied. Contrary to PrintTalk, there is no constraint solver that can improve the reusability of 3D components by using these assertion for determining suitable parameter values automatically. To generate valid shapes, programmers must manually find suitable values for all of a shape's parameters upon instantiation. In Listing 4, the `lamp_base` shape definition contains `assert` statements modelling the same parameter requirements as the equivalent PrintTalk `lamp-base` shape definition in Listing 3.

Listing 4 OpenSCAD assertions through which invalid parameter values can be detected.

```
41 module lamp_base(dia, height, screw_dia, mnt_l, mnt_w, mnt_th) {
42  assert(mnt_l >= mnt_w/2);
43  assert(dia >= (mnt_w + 5));
44  assert(height >= 5);
45  ... }
```

CadQuery

To modularise CadQuery shapes, programmers must make use of Python's language features such as functions or classes for defining new shapes. These allow shapes to be modularized through function parameters or parametrised constructor methods, respectively. While Python functions and constructor methods support default parameter values, there is no mechanism in place that allows programmers to leave parameters unbound, relying on a constraint solver to find suitable parameters. Furthermore, invalid parameter values cannot be detected without resorting to writing tedious `if` tests and raising errors accordingly.

4.2 Criterium 2: Flexible Composition

PrintTalk

Generally, complex 3D shapes are defined by composing simpler constituent shapes. For example, the lamp is designed out of individual components representing the base, arms, hinges, and a lamp shade. To correctly compose these shapes, anchor points indicating where other shapes can be attached in a composition must be defined. In PrintTalk, these points can be defined within a shape's definition, and exposed as parameters of that shape.

In the case of our lamp, the `lamp-base` component exposes compositional parameters `screw-x`, `screw-y`, and `screw-z` as the anchor point of the base to which a lamp arm can be attached. Constraints are in place to express the relations between these compositional parameters and the dimensional parameters of the `lamp-base` shape (lines 53–55 in Listing 3).

Although anchor points represented as compositional parameters increase the parameter count of composable shapes, we find this approach to be the sweet spot between reusability and composability. Because parameters can be left unbound when instatiating PrintTalk shapes, this provides flexibility in the way that shapes can be reused:

- **Anchor points are derived from other parameters:** By specifying the constituent's other parameters, its anchor points can be derived, and are essentially "locked" in place. In PrintTalk, the anchor point defined through composition parameters can be accessed and used to align other constituents in the composition with them.
- **Other parameters are derived from anchor points:** By constraining the constituent's anchor points, and leaving other parameter values such as dimensions *underconstrained*, these dimensions can be derived upon composition, resulting in a shape that fits the composition, without requiring the designer to manually determine all dimensions.

OpenSCAD

OpenSCAD has no support for anchor points, and the modules through which shapes can be abstracted and modularised offer no support for exposing information such as anchor points upon composition. Instead, anchor points must be calculated manually, external to the shape. Yet, calculating the anchor points of a constituent requires information a how the constituent is composed itself. This form of "leaky abstraction" complicates the reuse of shapes in OpenSCAD.

Listing 5 Additional calculations needed to determine the anchor point of the `lamp_base` shape.

```
77 base_screw_y=(base_dia/2)-(arm_width/2)-10;
78 base_screw_z = base_height/2+(arm_width-screw_dia);
79
80 lower_arm_pos_delta_y = sin(lower_arm_angle)*(lower_arm_length/2);
81 lower_arm_y=base_screw_y-lower_arm_pos_delta_y;
82 lower_arm_pos_delta_z = cos(lower_arm_angle)*(lower_arm_length/2);
83 lower_arm_z=base_screw_z+lower_arm_pos_delta_z;
```

Listing 6 Tagging CadQuery geometries in order to use them as composition parameters.

```
47 hole = base_shape.faces("%PLANE").faces(">Z").faces(">X")\
48     .edges("%CIRCLE").edges(">>Z[-2]")
49 hole.tag("hole")
```

For example, in Listing 5, additional calculations for determining the anchor point of the `lamp_base` shape are needed in order to correctly use that shape in a composition where the screw hole of the base needs to line up with the screw hole of the lower arm.

CadQuery

CadQuery allows for shapes to be composed in "Assemblies", and composition happens either by means of constraints, or by manually specifying the location, rotation, and scale of each constituent in the composition. CadQuery provides a limited set of constraints that can be used for expressing relations between parameters of constituent shapes. To determine their composition parameters, shapes can be queried to obtain their geometries such as surfaces, edges, and points, which can be tagged using the `tag` method. For example, the code snippet in Listing 6 is used to tag the screw hole of the base of the lamp.

After being tagged, geometries can be queried from their shape by using their tag, and can be used as composition parameters that can partake in composition constraints. Listing 7 illustrates how the screw hole of the base is constrained to line up with the screw hole of the lower arm of our lamp using CadQuery's built-in `Plane` constraint.

It is only possible to tag geometries that are present in a shape, and it is not possible to tag arbitrary anchor points relative to the shape. For example, it is not possible to tag a specific spot where an on-off switch can be added to our lamp, without there being a geometry that can be tagged on that exact spot.

4.3 Criterium 3: Declarative and Reusable Compositional Abstraction

PrintTalk

Defining shapes that contain constituent shapes placed in a pattern, such as the caps used for the hinges in our lamp, requires the functionality for instantiating a number of shapes, and placing each shape in the desired location. This functionality is split over two components in PrintTalk, facilitating the reuse of spatial patterns through constraints. Similar to shapes, constraints expressing spatial patterns can be defined once and reused later. Moreover, the

Listing 7 Composing CadQuery shapes by constraining their tagged geometries.

```
113 .constrain("base?hole", "lower_arm?front_bottom_hole", "Plane", param=0)\
```

Listing 8 Use of the `circular` spatial pattern in PrintTalk for placing shapes in a circular pattern.

```
1 (shape: cap (dia height)
2  (script:
3   (cylinder #:diameter dia #:height height)
4   (cut: (named: cutouts ((shape: ()
5                          (script:
6                           (for _ in (range 10)
7                            (cylinder #:diameter (/ dia 4) #:height height))))))))
8  (constraints:
9   (assert: (circular #:shapes cutouts #:radius (* dia 0.55)))))
```

Listing 9 OpenSCAD code using `for` loops for placing shapes in a circular pattern.

```
1 module cap(dia, height){
2  difference(){cylinder(h=height, d=dia, center=true);
3   union(){
4    for(i=[0:9]){
5     translate([cos(36*i)*dia*0.55, sin(36*i)*dia*0.55, 0])
6      cylinder(h=height, d=dia/4, center=true);
7 }}}}
```

pattern definition is independent of the shapes to be placed in that pattern.

For example, the `cap` shape definition in Listing 8 reuses the `circular` pattern defined in Listing 2 for modelling the 10 cutouts of the caps of the lamp's hinge-points (line 9).

OpenSCAD and CadQuery

In contrast, OpenSCAD and CadQuery require `for` loops to achieve the same, hampering the abstraction of patterns. This is illustrated in the OpenSCAD snippet in Listing 9 that uses a single `for` loop for instantiating and positioning the cutouts of the caps of the lamp's hinge-points (lines 4–6). As the parameters of shapes are immutable, and it is impossible to reposition shapes after their instantiation, shapes must be positioned correctly upon instantiation, which makes it impossible to split the functionalities in a way that allows for abstracting spatial patterns.

5 Conclusion

We presented PrintTalk, a constraint-based language for programmatic 3D modelling. PrintTalk enables 3D models to partake in compositions, as constituents to more complex shape, and features an composition mechanism that allows relations between constituents and their parameters to be expressed in a declarative manner, using constraints. We implemented PrintTalk as a domain-specific language on top of Racket. In order to evaluate PrintTalk, we compared it to state of the art in programmatic modelling tools, represented by OpenSCAD and CadQuery. From this evaluation, we conclude that PrintTalk's composition mechanism promotes the reusability of 3D shapes, with an underlying constraint solver that helps programmers with finding suitable parameter values for constituents, such that they can be composed into valid 3D models.

References

1 Greg J. Badros, Alan Borning, and Peter J. Stuckey. The cassowary linear arithmetic constraint solving algorithm. *ACM Transactions on Computer-Human Interaction (TOCHI)*, 8(4):267–306, December 2001. doi:10.1145/504704.504705.

2 Clark Barrett, Aaron Stump, Cesare Tinelli, et al. The smt-lib standard: Version 2.0. In *Proceedings of the 8th international workshop on satisfiability modulo theories (Edinburgh, UK)*, volume 13, page 14, 2010.

3 Alan Borning and Robert Duisberg. Constraint-based tools for building user interfaces. *ACM Trans. Graph.*, 5(4):345–374, October 1986. doi:10.1145/27623.29354.

4 Alan Borning, Robert Duisberg, Bjorn Freeman-Benson, Axel Kramer, and Michael Woolf. Constraint hierarchies. In *Conference Proceedings on Object-Oriented Programming Systems, Languages and Applications*, OOPSLA '87, pages 48–60, New York, NY, USA, 1987. Association for Computing Machinery. doi:10.1145/38765.38812.

5 GitHub - CadQuery/cadquery. https://github.com/CadQuery/cadquery. [Accessed 02-04-2025].

6 Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg. doi:10.1007/978-3-540-78800-3_24.

7 E. Dengler, M. Friedell, and J. Marks. Constraint-driven diagram layout. In *Proceedings 1993 IEEE Symposium on Visual Languages*, pages 330–335, 1993. doi:10.1109/VL.1993.269619.

8 Tim Felgentreff, Alan Borning, and Robert Hirschfeld. *Babelsberg: Specifying and solving constraints on object behavior*, volume 81. Universitätsverlag Potsdam, 2014.

9 Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, Shriram Krishnamurthi, Eli Barzilay, Jay McCarthy, and Sam Tobin-Hochstadt. A programmable programming language. *Commun. ACM*, 61(3):62–71, February 2018. doi:10.1145/3127323.

10 Jessie Frazelle. A new era for mechanical cad. *Commun. ACM*, 64(10):36–39, September 2021. doi:10.1145/3464909.

11 Joseph Gil and David H. Lorenz. Environmental acquisition: A new inheritance-like abstraction mechanism. *SIGPLAN Not.*, 31(10):214–231, October 1996. doi:10.1145/236338.236358.

12 Weiqing He and Kim Marriott. Constrained graph layout. *Constraints*, 3(4):289–314, 1998. doi:10.1023/A:1009771921595.

13 ImplicitCad — implicitcad.org. https://www.implicitcad.org/. [Accessed 02-04-2025].

14 GitHub - jscad/OpenJSCAD.org. https://github.com/jscad/OpenJSCAD.org. [Accessed 02-04-2025].

15 Christof Lutteroth, Robert Strandh, and Gerald Weber. Domain specific high-level constraints for user interface layout. *Constraints An Int. J.*, 13(3):307–342, 2008. doi:10.1007/S10601-008-9043-2.

16 Christof Lutteroth and Gerald Weber. User interface layout with ordinal and linear constraints. In *User Interfaces 2006, 7th Australasian User Interface Conference (AUIC 2006), Hobart, Tasmania, Australia, January 16-19 2005*, volume 50 of *CRPIT*, pages 53–60. Australian Computer Society, 2006. URL: https://dl.acm.org/doi/10.5555/1151758.1151764.

17 Aman Mathur, Marcus Pirron, and Damien Zufferey. Interactive programming for parametric cad. *Computer Graphics Forum*, 39(6):408–425, 2020. doi:10.1111/cgf.14046.

18 OpenSCAD — openscad.org. https://openscad.org/. [Accessed 02-04-2025].

19 GitHub - jeff-dh/SolidPython. https://github.com/jeff-dh/SolidPython. [Accessed 02-04-2025].

20 Ricardo Soto and Laurent Granvilliers. *On the Pursuit of a Standard Language for Object-Oriented Constraint Modeling*, pages 123–133. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008. doi:10.1007/978-3-540-79355-7_12.

21 Hiromasa Suzuki, Hidetoshi Ando, and Fumihiko Kimura. Geometric constraints and reasoning for geometrical cad systems. *Computers & Graphics*, 14(2):211–224, 1990. doi:10.1016/0097-8493(90)90033-T.

A Formal Syntax of PrintTalk

Figure 3 gives an overview of PrintTalk's formal syntax. PrintTalk is built on top of Racket [9] and inherits its s-expression syntax. Similar to Racket, PrintTalk makes use of prefix-notation, where function applications are structured with the operator preceding the operands: `(operator operand1 operand2 ...)`.

Identifiers are represented as an arbitrary number of characters, excluding white spaces and colons. Also similar to Racket, identifiers starting with `#:` are reserved for named parameters. Additionally, identifiers starting with an exclamation mark are reserved for parameters that are forced to act as constructor parameters, as discussed in Section 2.1.1.

Strings are represented as an arbitrary number of characters between double quotes (e.g., `"this is a string"`), and are primarily used for representing filenames that are provided as arguments to `print:` statements.

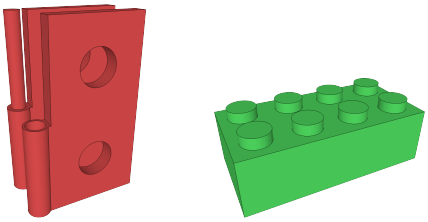
Numbers are either integers (e.g., `42`), or floating point numbers (e.g., `0.33`).

⟨*program*⟩ ::= ⟨*top-level expression*⟩*	*PrintTalk program*
⟨*top-level expression*⟩ ::=	*Top-level expressions*
⟨*expr*⟩	*Expression*
| ⟨*definition*⟩	*Definition*
| `(print:` ⟨*expr*⟩ string`)`	*Exporting shapes*
⟨*expr*⟩ ::=	*Expressions*
⟨*var*⟩	*Variable*
| ⟨*literal*⟩	*Literal*
| ⟨*application*⟩	*Function application*
| ⟨*expr*⟩`.`⟨*var*⟩	*Variable reference*
| `(named:` ⟨*var*⟩ ⟨*expr*⟩`)`	*Naming shapes or drawings*
| `(for` ⟨*var*⟩ `in` ⟨*expr*⟩ ⟨*expr*⟩*`)`	*For loop*
| `(shape: (`⟨*var*⟩*`) (script:` ⟨*expr*⟩*`))`	*Shape constructor*
| `(shape: (`⟨*var*⟩*`) (script:` ⟨*expr*⟩*`) (constraints:` ⟨*assertion*⟩*`))`	*Shape constructor*
⟨*application*⟩ ::= `(`⟨*expr*⟩ ⟨*expr*⟩*`)`	*Function application*
⟨*definition*⟩ ::=	*Definitions*
`(shape:` ⟨*var*⟩ `(`⟨*var*⟩*`) (script:` ⟨*expr*⟩*`))`	*Shape definition*
| `(shape:` ⟨*var*⟩ `(`⟨*var*⟩*`) (script:` ⟨*expr*⟩*`) (constraints:` ⟨*assertion*⟩*`))`	*Shape definition*
| `(constraint:` ⟨*var*⟩ `(`⟨*var*⟩*`)` ⟨*assertion*⟩*`)`	*Constraint definition*
⟨*assertion*⟩ ::= `(assert:` ⟨*expr*⟩`)`	*Assertion*
⟨*var*⟩ ::= identifier	*Variables*
⟨*literal*⟩ ::= number or string	*Literals*

Figure 3 PrintTalk's formal syntax.

B Extended Evaluation

This section provides an extended evaluation in which we consider hinges and building blocks (Figure 4) as two additional cases for comparing PrintTalk to OpenSCAD and CadQuery based on the three criteria of Section 1.1. The complete code listings are available as supplementary material,[2] and only the most relevant snippets are included in this section.



(a) hinge (b) brick

Figure 4 3D model of (a) a hinge, and (b) a building brick.

B.1 Criterium 1: Flexible Variability and Reusability

The hinge shape declares several parameters representing dimensions that must be compatible with each other. For example, the diameter of the screw holes must be smaller than the widths and heights of the leafs, the diameter of the knuckle must be larger than the thickness of the leaf, and the diameter of the pin must be in proportion to the length and width of the leaf.

PrintTalk

In PrintTalk, these dimensional properties can be modelled by constraints, so that the underlying constraint solver can raise an error when the provided parameter values do not satisfy the constraints. If a PrintTalk program does not assign a value to a parameter, it is treated as a constraint variable, and its value is then computed by the constraint solver, based on applicable constraints, before the 3D model is generated.

Listing 10 PrintTalk constraint for ensuring valid dimensions.

```
(shape: leaf (w t h knuckle-dia screw-dia)
 (script:
  (cuboid #:length w #:width t #:height h)
  (cut: (cylinder #:diameter screw-dia #:height t ...))
  (cut: (cylinder #:diameter screw-dia #:height t ...))
  (cuboid #:length (/ knuckle-dia 2) #:width t #:height (/ h 2) ...)
  (cylinder #:diameter knuckle-dia #:height (/ h 2) ...))
 (constraints:
  (assert: (>= screw-dia (/ w 8)))
  (assert: (< screw-dia w))
  (assert: (< screw-dia (/ h 2)))
  (assert: (>= knuckle-dia t))))
```

[2] https://doi.org/10.5281/zenodo.15593661

OpenSCAD

In OpenSCAD, it is possible to add `assert` statements for "constraining" parameters and raising an error when a provided parameter value does not meet an asserted condition. However, as there is no underlying constraint solver, OpenSCAD is not able to automatically compute values for parameters based on assertions.

Listing 11 OpenSCAD assertions for modelling conditions on parameter values.

```
1  module leaf (w, t, h, knuckle_dia, screw_dia){
2     assert(screw_dia>=w/8);
3     assert(screw_dia<w && screw_dia<h/2);
4     assert(knuckle_dia>=t);
5     ...
```

CadQuery

While CadQuery itself does not provide features for asserting conditions that parameter values must adhere to, this functionality can be achieved by manually programming `if` tests and raising Python exceptions accordingly. Similar to OpenSCAD, CadQuery is unable to automatically compute values of parameters for which no value was provided upon shape instantiation.

Listing 12 CadQuery `if`-tests and exceptions for modelling conditions on parameter values.

```
3  def leaf(w, t, h, knuckle_dia, screw_dia):
4      if screw_dia < w/8:
5          raise Exception("minimum value for screw_dia")
6      if screw_dia > w or screw_dia>h/2:
7          raise Exception("maximum value for screw_dia")
8      if knuckle_dia < t:
9          raise Exception("minimum value for knuckle_dia")
10     ...
```

B.2 Criterium 2: Flexible Composition

A `leaf` shape, representing the leaf and the solid part of the knuckle, is reused twice for constructing the complete `hinge` shape, with a pin attached to one leaf and a hole cut from the other leaf. To model a valid hinge, the diameters of the pin and the hole must be determined so that these fit together. Furthermore, the diameter of the knuckle must be sized accordingly, so that it is larger than the pin and larger than the thickness of the leaf.

PrintTalk

In PrintTalk, parameters and constraints can be used for representing dimensions and expressing relations between them. As an example, the `hinge` shape of Listing 13 contains parameters for representing the diameter of the pin (`pin-dia`), the diameter of the knuckle (`knuckle-dia`) and the rim around the pin (`pin-rim`). From within the `hinge`'s `constraints:` section (lines 22–26), composition constraints are placed on these parameters to ensure that they are sized appropriately, relative to each other. One of these parameters – `knuckle-dia` – is passed as an argument to the corresponding parameter of the `leaf` constituents (lines 17 and 20). Through PrintTalk's scoping and parameter-passing mechanism, additional

Listing 13 PrintTalk constraint for expressing a linear pattern.

```
(shape: hinge (w t h screw-dia pin-dia knuckle-dia pin-rim)
 (script:
  ;; Leaf with pin
  (named: leaf1 (leaf ... #:knuckle-dia knuckle-dia #:screw-dia screw-dia))
  (named: pin (cylinder #:diameter (- pin-dia 1) #:height (/ h 2) ...))
  ;; Leaf with hole
  (named: leaf2 (leaf ... #:knuckle-dia knuckle-dia #:screw-dia screw-dia ...)))
  (cut: (named: pin-hole (cylinder #:diameter pin-dia #:height (/ h 2) ...))))
 (constraints:
  (assert: (>= pin-dia (* 0.8 knuckle-dia)))
  (assert: (< pin-dia knuckle-dia))
  (assert: (= pin-rim (/ (- knuckle-dia pin-dia) 2)))
  (assert: (>= pin-rim (/ pin-dia 8)))))
```

constraints can be added to the `knuckle-dia` constraint variable from within the constituent `leaf` shapes. In this case, an additional constraint stating that the diameter of the knuckle (`knuckle-dia`) must be larger than the thickness of the leaf (`t`) is added from within the `leaf` shape (line 12 of Listing 10). In PrintTalk, if the `hinge` shape is instantiated without values specified for the `pin-dia`, `knuckle-dia`, and `pin-rim` parameters, these parameters are treated as constraint variables. Their values are then determined based on constraints originating from both the `hinge` shape itself and its constituent `leaf` shapes.

OpenSCAD and CadQuery

In OpenSCAD and CadQuery, it is not possible to add composition constraints that express relationships between parameter values of constituents, and automatically derive parameter values based on these constraints. Instead, parameter values must be determined manually, and provided as arguments upon shape instantiation. For the `hinge` shape, this means that values for `pin_dia` and `knuckle_dia` must be provided upon instantiating the shape. Determining valid values for these parameters, however, requires inspecting the code of the constituents, taking into account conditions (expressed as assertions or `if`-tests) on parameter values and any exceptions these conditions potentially raise.

B.3 Criterium 3: Declarative and Reusable Compositional Abstraction

The brick shape (Figure 4(b) & Figure 5) includes two rows of cylinders on its top surface and a row of hollow cylinders on its bottom surface. The cylinders within each row are arranged in a linear pattern.

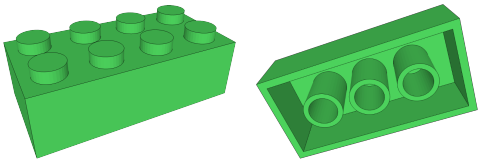


Figure 5 Brick with linear patterns on its top (left image) and on its bottom (right image).

Listing 14 PrintTalk constraint for expressing linear patterns.

```
(constraint: linear-x (!shapes x0 xn dx)
 (for shape in shapes
  (assert: (= xn (+ x0 (* (- n 1) dx))))
  (assert: (= shape.x (+ x0 (* dx i))))))
```

Listing 15 The brick shape modelled by reusing the `linear-x` constraint in PrintTalk.

```
(shape: bottom-cyl (height)
 (script:
  (cylinder #:diameter 6.5 #:height height)
  (cut: (cylinder #:diameter 4.8 #:height height))))

(shape: block (l w h clearance)
 (script:
  (cuboid #:length (- (* 8 l) clearance) #:width (- (* 8 w) clearance) ...)
  (cut: (cuboid #:length (- (- (* 8 l) clearance) 2.4) ...))))

(shape: brick (!l)
 (script:
  (block #:l l #:w 2 #:h 9.6 #:clearance 0.2)
  (named: bot-cyls ((shape: () (script:
           (for i in (range (- l 1))
              (bottom-cyl #:height 8.4 #:z -0.6))))))
  (named: top-cyls-c1 ((shape: () (script:
           (for i in (range l)
              (cylinder #:diameter 4.8 #:height 1.8 #:y -4 #:z 5.7))))))
  (named: top-cyls-c2 ((shape: () (script:
           (for i in (range l)
              (cylinder #:diameter 4.8 #:height 1.8 #:y 4 #:z 5.7)))))))
 (constraints:
  (assert: (linear-x #:shapes bot-cyls #:x0 (- (* (- (/ l 2) 1) 8)) #:dx 8))
  (assert: (linear-x #:shapes top-cyls-c1 #:x0 (- (-(* (/ l 2) 8) 4)) #:dx 8))
  (assert: (linear-x #:shapes top-cyls-c2 #:x0 (-(-(* (/ l 2) 8) 4)) #:dx 8))))

(print: (brick #:l 4) "brick-4.stl")
```

PrintTalk

In PrintTalk, the logic for arranging shapes in a pattern can be abstracted as constraints. This abstraction eliminates the need to repeat positioning calculations each time shapes are subsequently placed in that pattern. For example, the `linear-x` constraint in Listing 14 expresses the pattern used for modelling the rows of cylinders on the top and bottom surfaces of the brick. As a result, the code for instantiating cylinders can be separated from the code for positioning them.

OpenSCAD and CadQuery

OpenSCAD and CadQuery do not feature abstraction mechanisms that allow the code for instantiating shapes to be decoupled from the code for positioning these shapes. As a result, the calculations for linearly positioning the cylinders on the top and bottom surfaces of the brick must be repeated for each row of cylinders.

Listing 16 In OpenSCAD, calculations for determining the position of shapes must be repeated for each row of cylinders.

```
module brick (l){
    cyl_delta = 8;
    bottom_cyl_begin_x = -(l/2-1)*cyl_delta;
    top_cyl_begin_x = -((l/2)*cyl_delta)+cyl_delta/2;
    union(){
        block(l=l, w=2, h=9.6, clearance=0.2);
        for(i=[0:l-2]){
            translate([bottom_cyl_begin_x+i*cyl_delta, 0, -0.6])
                bottom_cyl(height=8.4);
        }
        for(i=[0:l-1]){
            translate([top_cyl_begin_x+i*cyl_delta, -4, 5.7])
                cylinder(h=1.8, d=4.8, center = true);
        }
        for(i=[0:l-1]){
            translate([top_cyl_begin_x+i*cyl_delta, 4, 5.7])
                cylinder(h=1.8, d=4.8, center = true);
        }
    }
}
```