PrintTalk: A Constraint-Based Imperative DSL for 3D Printing

Jef Jacobs
Jens Nicolay
Christophe De Troyer
Wolfgang De Meuter
Software Languages Lab
Vrije Universiteit Brussel
Brussel, Belgium

Abstract

We present PrintTalk, a DSL to "program" 3D objects, called "gadgets". PrintTalk also features "topologies", which are predefined spacial arrangements of gadgets. Gadgets are composed by executing a gadget script (possibly consisting of subscripts) that 'draws' the gadget in the 3D scene. However, executing the script also returns a number of constraint variables. These variables can be constrained inside the gadget and can also be bound outside the gadget in order to constrain the produced gadgets after the facts. This is the essence of the gadget composition mechanism of PrintTalk.

PrintTalk is implemented in DrRacket. Running a PrintTalk program generates a file that is sent to the 3D printer. We validate PrintTalk qualitatively by comparing the code for complex gadgets with the code needed to print those gadgets in existing languages.

CCS Concepts: • **Software and its engineering → Domain specific languages**; *Multiparadigm languages*; *Constraints.*

Keywords: PrintTalk, Domain Specific Language, DSL, Design Languages, 3D Modelling, CAD, Constraints

ACM Reference Format:
Jef Jacobs, Jens Nicolay, Christophe De Troyer, and Wolfgang De Meuter. 2021. PrintTalk: A Constraint-Based Imperative DSL for 3D Printing. In *Proceedings of the 18th ACM SIGPLAN International Workshop on Domain-Specific Modeling (DSM '21), October 18, 2021, Chicago, IL, USA.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3486603.3486778

DSM '21, October 18, 2021, Chicago, IL, USA

ACM ISBN 978-1-4503-9106-1/21/10…$15.00
https://doi.org/10.1145/3486603.3486778

1 Introduction

Ever since the invention of the 3D printer in 1983, 3D printing technology has evolved enormously. Hobbyist 3D printers can be bought for less than 1000 euro[1]. Professional 3D printers can print objects the size of a speedboat[2].

Instructing a 3D printer to print a shape is a software problem. Two approaches exist. **Direct Modelling** allows a designer to draw a 3D shape in an advanced editor. Such an editor is essentially the equivalent of 2D WYSIWYG drawing editors: a shape is manually designed, element by element, and then sent to the printer as soon as it is ready. **Parametric Modelling** allows a designer to program and/or describe the object by parameterising a suite of shapes in some advanced 3D design tool or in a textual language. In the latter case, the 3D designer writes a script and executing that script prints a 3D shape.

Direct modelling has the drawback that it hampers a systematization of the 3D printing process. Copying and pasting components and carefully assembling them "by hand" is the technique used for reuse and composition. The possibilities for putting shapes together are essentially limited by the number of menu items featured by the editor being used. Parametric modelling clearly has a larger potential to turn 3D programming into a true software engineering discipline where parts of a composite 3D design can be assembled, adapted and reused in a systematic way. However, as we will argue in this paper, existing languages are still rudimentary.

We present PrintTalk, an experimental DSL that fosters a high-level specification, composition, and reuse of 3D designs. The main characteristics of PrintTalk are:

- PrintTalk was designed in DrRacket [5] which makes it easy to extend and modify the language.
- PrintTalk focuses on the assembly *process* generating the objects (called 'gadgets' in PrintTalk) to be printed, rather than on the objects themselves.
- PrintTalk features both imperative and constraint-based constructs. The composition of gadgets is done

[1] https://www.prusa3d.com/
[2] https://composites.umaine.edu/3dirigo-the-worlds-largest-3d-printed-boat/

in an imperative way: subgadgets of a gadget are printed one after the other in a 3D scene. However, by associating every gadget with a number of constraint variables, some aspects of the composition are determined by a constraint solver.
- PrintTalk is compiled into STL, which can be used to generate so-called G-code [11] by publicly available slicing software. This means that PrintTalk can be deployed for many different brands of 3D printers.

The paper is structured as follows. In Section 2, we motivate PrintTalk. Section 3 presents its basic concepts, and the features it offers for reuse and composition. Section 4 explains PrintTalk's toolchain which turns PrintTalk into a practical language that enables developers to really print 3D objects. Section 5 evaluates PrintTalk.

2 State of the Art

We review the state of the art in Computer-Aided Design (CAD) technology for 3D printers. As explained in the introduction, two main philosophies for 3D modelling exist. **Direct modelling** can be thought of as sculpting 3D shapes in some editor before sending them to the printer. 3D models designed like this lack parameters and relations. Editing consists of drawing and scaling geometries visually. Direct modelling thus requires a 3D designer to compose a 3D shape 'by hand'. Reusing designs typically happens via 'copy-paste reuse', i.e., by copying and pasting an existing design and altering the copy in the editor [7]. Examples of direct modelling CAD software are Creo Direct[3] and Shapr3D[4]. **Parametric modelling** builds a history tree of a 3D shape. Designing happens by describing and manipulating this tree. A well-known parametric modelling technique is Constructive Solid Geometry (CSG) [9]. In CSG, complex 3D shapes consist of combinations of primitive shapes (such as cubes, spheres, and cylinders). In CSG, leaf nodes of the tree correspond to the primitive shapes. Non-leaf nodes represent binary operations such as intersections, unions and differences.

Parametric modelling can also be supported by editors similar to, but more powerful than, their direct modelling counterparts (e.g., FreeCAD[5]). However, things become much more interesting if an actual *program or script* can be used to generate the history tree of the 3D model. Two approaches exist. Either the script is written in some mainstream programming languages such as Python and performs a number of calls to some *library* that was developed in the same language, or the script is written in a *dedicated DSL* for 3D printing. We review the state of the art for both approaches in the forthcoming sections.

[3] https://www.ptc.com/en/products/creo/direct
[4] https://www.shapr3d.com/
[5] https://www.freecadweb.org/

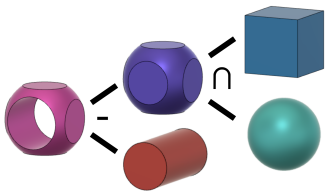

Figure 1. A CSG tree.

```
1  function main() {
2   return difference(
3    cube({size: 10, center: true}),
4    cylinder({r:3,h:10, center:true}),
5    cylinder({r:3,h:10, center:true}).rotateX(90),
6    cylinder({r:3,h:10, center:true}).rotateY(90));}
```

Listing 1. 3D model designed using JSCAD.

2.1 Parametric Modelling with Libraries

Many libraries for modelling 3D shapes with varying levels of maturity can be found on the internet. The following is a list of some popular ones that are each hosted by a different mainstream programming language. **CadQuery** is a Python library that supports common CAD file formats such as STEP and STL. **Lua Cad** is a collection of Lua-scripts that make it possible to generate OpenSCAD scripts (see Section 2.2). **JSCAD** enables 3D design in JavaScript. All such libraries are very similar mainly differ in technicalities and in the infrastructure they support. To give a feeling about the way they are used, Listing 1 illustrates a 3D model written in JSCAD. The model represents a cube from which three rotated cylinders are cut. The resulting 3D shape is illustrated in Figure 2.

The library approach has the advantage in that it can be used in combination with the entire host language, along with the infrastructure provided by the IDE that is used. Programmers that are already familiar with the language are not required to learn an entirely new programming language, which results in a less steep learning curve.

2.2 Parametric Modelling with Specific Languages

In contrast to libraries, a number of dedicated programming languages exist of which OpenSCAD[6] and ImplicitCAD[7] are the most advanced. Listing 2 expresses the same shape as in Listing 1 and is valid for both OpenSCAD and ImplicitCAD. OpenSCAD compiles 'scad scripts' into STL. A scad script describes a 3D shape using Constructive Solid Geometry (CSG) and extrusion of 2D outlines. OpenSCAD also features `assert` statements to be used when the values of the configurable parameters must meet some requirements, e.g., when a dimension determined by a variable may not exceed a certain length. Apart from the declarative definition of 3D shapes, OpenSCAD also includes `if` statements and `for` loops, requiring programmers to describe complex 3D models imperatively.

[6] https://openscad.org/
[7] https://implicitcad.org/

```
difference () {
   cube(10, center = true);
   cylinder(r=3, h=10, center = true);
   rotate([90, 0, 0]) cylinder(r=3,h=10, center=true);
   rotate([0, 90, 0]) cylinder(r=3,h=10, center=true);}
```

Listing 2. 3D model designed using OpenSCAD.

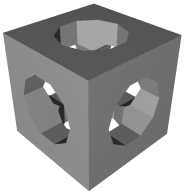

Figure 2. 3D model resulting from Listing 1 and Listing 2.

2.3 Comparing the Approaches

Describing 3D models by code has serious advantages when it comes to reuse via composition and abstraction.

Reusability. As soon as reuse of 3D models becomes an issue, tool-based approaches are restricted to copy-paste reuse. In code-based approaches, much richer reuse mechanisms can be conceived. The simplest form of reuse is *instantiation*: 'copies' of a 3D objects are created by calling some constructor multiple times, each time with different parameters. Constructors themselves emerge from using *abstraction* and *composition*: the ability to compose a number of 3D shapes into a composite 3D shape and then abstract over this 3D shape by giving it a meaningful name that can be used for composition in its turn. It is clear that this is the bread and butter of any modern programming language, which makes the possibilities for reuse much more advanced in the code-based approach.

Parameterisation. As said before, parametric modelling allows some aspects of 3D models to be modified by filling in parameters. In tool-based approaches, this happens by clicking and editing windows. In code-based approaches, parametric values can be passed on to constructors. However, combined with abstraction and composition, this results in an inherently more powerful mechanism because a composed 3D model can depend on a parameter and pass this parameter to one or more of its constituents.

Automation. A powerful aspect of programming languages is that they allow expressing an entire process with just one construct. For example, when building a composition, the repetition of a particular 3D shape can be done with iteration and selection (i.e., `if` tests) constructs. This is more powerful than some set of predefined menu items in an editor.

Testing and Static Analysis. Once 3D models are described by code, it becomes possible to write unit tests. For example, one could test whether the dimensions of an object comply with a certain specification. Additionally, it becomes possible to perform static analysis on the code in order to verify aspects such as manufacturability, rigidity and stability of the 3D model.

We conclude that a code-based approach has clear advantages over editor-based approaches, especially if the goal is to turn 3D printing into a systematic engineering process that can simplify the production of variations of the same object and that can build objects by composing and parameterising the description of previously designed objects.

2.4 Towards More Powerful 3D Printing DSLs

One can wonder what the requirements are for powerful 3D printing DSLs. In 1976, Niklaus Wirth postulated the by now classic equation:

Programs = Algorithms + Data Structures

When it comes to the **data structures** being used, we envision a 3D printing DSL to be at least as powerful as OpenSCAD and its derivatives. In other words, a powerful 3D printing DSL will need:

1. a series of **built-in 3D shapes** such as spheres, cubes, etc., which form the primitive building blocks of the language.
2. a **composition mechanism** to compose primitive 3D shapes into more complex shapes that can, in their turn, be used for composition.
3. an **abstraction mechanism** that allows a 3D modeler to abstract over the composition (for example, by giving it a name), and that allows parameterisation of the composed 3D shape for future deployment.

The combination of these three mechanisms enables the construction of any complex 3D object as a composition of built-in objects. This is very similar to the CSG concept featured by languages like OpenSCAD.

When it comes to **algorithms**, the design of the DSL is less obvious. One could embed the data structures described above in a general purpose language with recursion and iteration. However, this is exactly what the library approach does. A potentially more powerful approach is to look at more declarative languages. In analogy to 2D GUI software, the family of *constraint programming languages* naturally comes to mind: we use constraints to specify how several constituents of a 3D shape should fit together. E.g., in a constraint programming language, it is very simple to specify things like "this object should be on top of that object and underneath this object". However, declarative programming sometimes results in very unnatural programs. E.g., given some n, drawing exactly n copies of the same object typically requires the programmer to rely on recursion. Languages like Prolog show that this easily results in unnecessarily complex programs that are typically expressed much simpler in a scripting language like Python.

We therefore argue for a **multiparadigm approach** in which the programmer can (i) write an (imperative) script that draws a 3D shape possibly relying on iteration, and (ii) have his objects parameterised by constraint variables that

can be filled in *later*, when the shape becomes part of a larger composite shape.

2.5 Mixing Constraints and Imperative Features

Constraint languages allow the programmer to use *constraint variables* and 'connect them' to one another by *constraints* [12]. A *constraint solver* is used to find bindings for the variables such that all constraints of the program are satisfied.

Many constraint solvers exists. Examples include Cassowary [1] and 'satisfiability modulo theories' (SMT) solvers such as the Z3 theorem prover [2]. Not all solvers are equally powerful. For example, Z3 is able to solve nonlinear constraints, while Cassowary is unable to do so.

Constraint programming languages typically result in *multidirectional programs*. When pegging a constraint variable to some value, the constraint solver will automatically determine the value for the other variables, irrespective of which variable was initially pegged by the programmer. It is not trivial to reconcile this mechanism with imperative programming since this style of programming is all about executing *unidirectional* programs that peg the value of their variables 'manually' by means of an assignment construct.

The work by Felgentreff et al. [3] discusses two main approaches for integrating constraints into an imperative language. **Constraint Satisfaction Libraries** can be used without requiring changes to the host programming language [3]. The 'constraint variables' of the library are actually data values of the host language. It is up to the user of the library to maintain the connection between both types of variables as needed. **Constraint-Imperative Programming (CIP) Languages** (e.g., [8] and [10]) reconcile constraint-based and imperative properties by design. CIP languages provide a dedicated syntax for asserting and defining constraints. The semantics of the variables is an integral part of the semantics of the programming language.

In the next section, we present PrintTalk, a CIP DSL with both imperative and constraint-based characteristics.

3 PrintTalk: Concepts and Examples

PrintTalk is DSL that enables abstraction, composition, and reuse of 3D shapes. In addition to primitive values such as numbers and strings, PrintTalk features two concepts that are specific to 3D modelling: *gadgets* and *topologies*. These can be combined with one another by a mixture of imperative and constraint-based concepts.

In the remainder of this section, we present an overview and examples of PrintTalk and its concepts, and explain how they enable abstraction, composition, and reuse.

3.1 Gadgets

Gadgets represent 3D shapes. Gadgets can be parameterised and instantiated with arguments. PrintTalk contains native gadgets for primitive 3D models that represent cubes, cuboids, spheres, and cylinders. For example, the instantiation `(cube x y z size)` creates a gadget that represents a cube of size `size` with its centre at position `(x, y, z)`.

A new type of gadgets is defined using the `gadget:` construct whose general form is as follows:

```
1 (gadget: <name>
2         (<parameters>)
3         (<constraint-vars>)
4         (script: <scripts>)
5         (constraints: <constraints>))
```

The definition of a gadget consists of a name, a list of (ordinary) parameters, a list of constraint variables, a sequence of script statements and a set of constraints. An example can be found in Listing 3.

The script of a gadget specifies the instantiation and composition of one or more "subgadgets". Subgadgets can be instantiated with both ordinary 'parameter' variables and constraint variables in their argument expressions. The number of argument expressions of each subgadget instantiation must be the same as the number or ordinary variables (i.e. *not* constraint variables) defined by those subgadgets. Hence, gadget instantiation only peggs the value of the ordinary variables and leaves the value of the constraint variables open.

PrintTalk implements the Constructive Solid Geometry (see Section 2) modelling technique to design complex 3D shapes by combining multiple simpler 3D shapes. By default, all subgadgets instantiated in a gadget's script are combined by means of a 'union'. When these gadgets are instantiated, their gadget scripts are executed as well, thereby possibly instantiating other subgadgets in their turn. It is possible to subtract gadgets from other gadgets by using the `cut:` statement. While PrintTalk does not directly support intersections, the same can be achieved by combining `cut:` statements, using the following formula: `A intersect: B = A cut: (A cut: B)`. Gadget scripts may not be empty and may not consist solely of `cut:` statements.

Constraint variables are variables of which the value is not specified when instantiating a gadget. Instead, values for constraint variables will be assigned by the underlying constraint solver at a later moment in time, when a 3D model consisting of various gadgets is finally generated using PrintTalk's `print:` statement. It is only when calling `print:` that a 3D model is rendered, thus it is only required to solve constraints just before this rendering takes place. We will discuss the exact semantics of the constraint variables in Section 3.2.

Example: Cube with a Hole. To exemplify the concepts explained so far, Listing 3 contains the definition of a gadget named `cube-hole` which represents a cube with a hole in it. `cube-hole` takes four parameters (line 2). The first three parameters (`x`, `y`, and `z`) represent the absolute 3D position (i.e., The position relative to the origin at `(0, 0, 0)`) of the gadget. The fourth parameter (`cube-size`) represents the length

```
1  (gadget: cube-hole
2          (x y z cube-size)
3          (cyl-dia)
4   (script: (cube x y z cube-size)
5            (cut: (cylinder x y z cyl-dia cube-size)))
6   (constraints: (constraint:<! cyl-dia cube-size)))
```

Listing 3. A gadget that represents a cube with a hole.

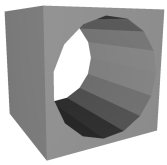

Figure 3. 3D model representing the `cube-hole` gadget in Listing 3.

of every rib. `cube-hole` can be instantiated and converted into an `.stl` file using a `print:` statement, as shown below. The resulting shape, stored in an `.stl` file, is depicted in Figure 3. Gadget `cube-hole` introduces one constraint variable, `cyl-dia`), that represents the diameter of the hole (line 3). The value of this variable is determined automatically by `print:`. Line 6 represents the constraints that are local to the gadget. In this example, there is one constraint that states that the diameter of the hole `cyl-dia` must be smaller than the size of the cube `cube-size`. Lines 4–5 represent the gadget's script. This script instantiates two subgadgets: a cube with size `cube-size`, and a cylinder with size `cyl-dia`. The cylinder is cut from the cube by means of the `cut:` statement.

```
1  (print: (cube-hole 0 0 0 10) "cube-hole.stl")
```

3.2 Gadget Constraints

In addition to the ordinary variables that correspond to a gadget's parameters, PrintTalk also features constraint variables. While an ordinary variable is explicitly given a value in a program, constraint variables get a value assigned by a constraint solver (discussed further below). This happens when gadgets are printed. At that moment, the constraints of a gadget are solved, thereby taking into account the constraints of its subgadgets as well.

To constrain the possible values a constraint variable may attain, declarative constraints must be specified. Constraints in PrintTalk can only be specified as part of a gadget definition. By convention, identifiers used for constraints end with an `!` to indicate that a condition is being asserted. A constraint's arguments can either be numbers, constraint variables or solver-expressions. A solver-expression is an expression that is not evaluated by PrintTalk, but by the underlying constraint solver instead. It is up to the programmer to ensure that the provided expression is supported by the underlying solver, and that the number and type of arguments matches with what is expected by the underlying solver. Identifiers of solver-expressions start with `solver:`. For example, `(constraint:=! radius (solver:/ diameter 2))` expresses a constraint that `radius` must always be equal to half of `diameter`.

Type of Constraints. PrintTalk supports three types of constraints: comparative constraints, geometric constraints, and user-defined constraints.

Comparative constraints, such as `=!`, `>!` and `<!`, are used to constrain the values of constraint variables based on the values of ordinary and/or other constraint variables. For example, in Listing 3 on line 6, gadget `cube-hole` asserts a `<!` constraint between constraint variable `cyl-dia` and ordinary variable `cube-size`.

Geometric constraints constrain geometric elements such as distances or angles in an Euclidean space. In its current implementation, PrintTalk supports `distance!`, `angle!`, `parallel!`, and `perpendicular!` geometric constraints. For example, constraint `(constraint:distance! u v w x y z 42)` constrains the Euclidean distance between positions (u, v, w) and (x, y, z) to be equal to `42`.

Finally, *user-defined constraints* are defined as composition of other constraints.

For example, Listing 10 in Section 5 introduces two user-defined constraints that are used in the remainder of that section.

Constraint Combination across Subgadgets. The ordinary variables of a gadget can be thought of as variables that are part of PrintTalk's imperative side. They are bound when instantiating a gadget. Whenever a gadget instantiates subgadgets, the ordinary parameters of these subgadgets get bound as well. Hence, an entire tree of gadgets is instantiated and all ordinary parameters of the gadgets occuring in that tree get a value that is pegged forever.

The **semantics of the constraint variables** of a gadget is totally different. Constraint variables can be thought of a part of the 'return value' of the gadget script. Obviously that return value is the gadget itself but the important thing to understand is that this gadget is automatically 'decorated' with its constraint variables. They are an **integral part of the gadget**. This mean that it must also be possible to refer to the constraint variables of a gadget if that gadget becomes part of a composition. This is accomplished by the `g@c` notation which can be used to designate a constraint variable `c` that resides in a gadget `g`.

This entire mechanism is illustrated by Listing 4, which describes three gadgets. Notice that this gadget script uses the `named:` form for the first time in the paper. `named:` can be used in a gadget script to introduce local gadgets that can be referred to further up in the same script. Hence, `named:` corresponds to the `let`-form that exists in many programming languages. The `block` and `ball` gadgets have ordinary variables `(x, y, z)` that represent their positions. Their sizes are represented by constraint variables on which constraints are placed from within the gadget-definitions. The size of `block` gadgets is constrained to be smaller than 10 (line 3).

The size of `ball` gadgets is constrained to be greater than 8 (line 7).

The `block` and `ball` gadgets are instantiated and combined by the `combine` gadget (lines 10–11). This gadget further constrains the sizes of the newly instantiated gadgets by stating that they must be equal (lines 12–13). This is where the `@` notation is used to refer to the `size` constraint variables of both subgadgets.

```
1  (gadget: block (x y z) (size)
2   (script: (cube x y z size))
3   (constraints: (constraint:<! size 10)))
4
5  (gadget: ball (x y z) (size)
6   (script: (sphere x y z size))
7   (constraints: (constraint:>! size 8)))
8
9  (gadget: combine () ()
10  (script: (named: block (block 0 0 0))
11           (named: ball (ball 10 0 0)))
12  (constraints: (constraint:=! (block@size)
13                               (ball@size))))
```

Listing 4. Combining Constraints in PrintTalk.

Constraint Solving. When a 3D model is generated from a gadget, ordinary variables such as gadget parameters will have a value that was explicitly assigned by the program at some point, but the constraint variables will not have a value, even though they may be heavily constrained by the entire constraint-network that emerges by composing the tree of subgadgets comprising the gadget.

We call a constraint variable without a value "unresolved". It is the role of the constraint solver to try to satisfy all constraints and assign values to all constraint variables. It is up to the programmer to ensure that all constraint variables that are necessary for generating the 3D model are sufficiently constrained, so that they can be resolved by the constraint solver. When a gadget is underconstrained, the overal final gadget can be given extra constraints (e.g. `(constraint:=! convar 42)`) to pegg the value of a constraint variable and force the constraint solver to trickle that value through the entire gadget (and all the subgadgets that transitively depend on that variable).

PrintTalk raises an error when an unresolved constrained variable is involved in generating a 3D model upon running the `print:` statement.

3.3 Topologies

3D shapes often contain a pattern or repeated shapes in their design that are straightforward to construct in an imperative style. We invite the reader to have a look at Figure 6 which shows a table with three legs. PrintTalk supports such repetitions of gadgets in the form of **topologies**. A topology creates a gadget that consists of a union of repetitions of the same gadget instantiated by the topology. At the time of writing, PrintTalk supports three types of topologies. Figure 4 illustrates them visually.

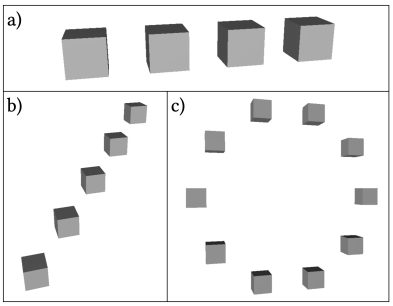

Figure 4. PrintTalk's topologies: a) `linear-repeated:`, b) `linear-spaced:`, and c) `ring-repeated:`.

The `linear-spaced:` topology evenly distributes a certain number of instantiations of the same gadget between two Euclidean positions.

```
(linear-spaced: <number>
                (<beginX> <beginY> <beginZ>)
                (<endX <endY> <endZ>)
                <gadget>)
```

The topology takes four parameters that specify the number of gadgets to be instantiated between the two provided positions (with an additional gadget being instantiated at these positions as well), the begin-position, end-position and the gadget instantiation. In the gadget instantiation argument of the `linear-spaced:` topology, PrintTalk offers the programmer three **pseudo-variables**[8] that are automatically filled in by PrintTalk, namely `X`, `Y` and `Z`, as the x, y, and z coordinates of the gadget instantiation.

An example illustrating the use of the `linear-spaced:` topology is provided in Listing 5. The resulting 3D model is illustrated in Figure 4(b). The gadget resulting from using a

```
1  (linear-spaced: 3 (0 0 0) (60 50 70) (cube X Y Z 10))
```

Listing 5. PrintTalk's `linear-spaced:` topology.

`linear-spaced:` topology contains three constraint variables (`deltaX`, `deltaY` and `deltaZ`), which represent the difference between adjacent gadgets' x, y, and z-coordinates, respectively. The code presented in Listing 6 uses this feature to state that the value of `deltaX` should be equal to 10.

```
1  (gadget: spacedCubes () (endX)
2   (script:
3    (named: spaced (linear-spaced: 3
4                   (0 0 0) (endX 50 70)
5                   (cube X Y Z 10))))
6   (constraints: (constraint:=! (spaced@deltaX) 10)))
```

Listing 6. Specifying the distance between gadgets instantiated by the `linear-spaced:` topology.

The `linear-repeated:` topology instantiates a gadget a number of times, where the position of each gadget instance is determined by the position of the previous gadget instance or by the iteration count. Pseudo-variables `PREVIOUS` and `COUNT` are available to reference the previous gadget and to obtain the iteration count, respectively.

[8]Pseudo-variables are variables like `this` or `self`. They can be used but not given a value by the programmer.

```
(linear-repeated: <number> <first-gadget> <gadget>)
```

The topology takes three arguments that represent the number of gadgets to be instantiated (apart from the first gadget), the first gadget to be instantiated and the remaining gadgets to be instantiated. The `PREVIOUS` and `COUNT` pseudo-variables can only be used in the (final) parameter that represents the remaining gadgets. The code presented in Listing 7 uses the `COUNT` pseudo-variable to specify the z-coordinate of cubes. The resulting 3D model is illustrated in Figure 4(a).

```
1 (linear-repeated: 3 (cube 0 0 0 5)
2                     (cube 0 0 (* 10 COUNT) 5))
```

Listing 7. PrintTalk's `linear-repeated:` topology.

The `ring-repeated:` topology instantiates the same gadget a number of times and places them in a ring.

```
(ring-repeated: <centreAxis1> <centreAxis2>
                <radius>
                <amount>
                <gadget>)
```

This topology takes five parameters that represent the centre coordinates and the radius of the ring, the number of gadgets to be instantiated and the gadget instantiation expression itself. The topology exposes two pseudo-variables (`X` and `Y`) that automatically contain the values for the parameters that represent the coordinates of the gadgets on the axes that make up the plane of the ring. The code presented in Listing 8 illustrates how the `ring-repeated:` topology can be used to construct a ring of cubes on the xy-plane with the centre of the plane on position `(0, 0, 0)`. The resulting 3D model is illustrated in Figure 4(c).

```
1 (ring-repeated: 0 0 50 10 (cube X Y 0 10))
```

Listing 8. PrintTalk's `ring-repeated:` topology.

Discussion. The list of topologies that are currently supported has emerged organically from using PrintTalk on various examples. We therefore do not claim that this is the ultimate set of topologies that is necessary and sufficient to generate all possible gadgets whose constituents contain patterns of repeated subgadgets.

The point of PrintTalk's topologies is that they expose an iterative process without polluting the language with imperative loops, iteration variables and variables that need to be updated manually by the programmer using the assignment statement. Our conjecture is that, in specifying 3D shapes, such variables are only justified when calculating the way subgadgets generated by the loop relate to one another spatially. In PrintTalk, updating the variables is hidden in the language run-time. The variables themselves are exposed to the programmer as pseudo-variables.

4 Implementation

We implemented PrintTalk as an embedded DSL on top of Racket [4], a powerful and extensible language that makes

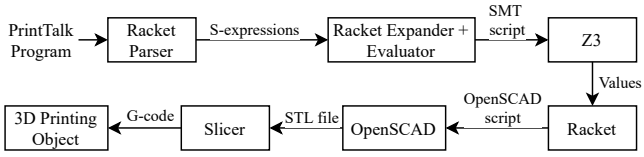

Figure 5. The PrintTalk toolchain.

it highly suitable for implementing DSLs. PrintTalk inherits Racket's host features such as garbage collection and interaction with the operating system. Racket was chosen as a base for PrintTalk for its powerful macro system, which allows for the definition of new syntax without requiring the modification of lexers and parsers. Our implementation uses Z3 for solving constraints, and OpenSCAD for generating STL files that represent 3D models that can be visualized and printed. We discuss these tools and how they work together in our current implementation in the remainder of this section. The entire toolchain is shown in Figure 5.

4.1 The Racket Front-End

Upon execution, a PrintTalk program is first transformed into a set of Racket S-expressions that then are executed by a Racket interpreter. It is the role of PrintTalk's parser to perform this transformation, and it relies on two mechanisms built into Racket for doing so: macros, and a parser library.

Pattern-based macros are used to transform the S-expression-based portion of PrintTalk's syntax, using Racket's `define-syntax`.

The PrintTalk syntax contains three language elements that are not S-expressions: constraint variable access, constraint assertions, and solver expressions. We used Racket's `parser-tools-lib` package to implement a LALR(1) parser for transforming this portion of PrintTalk's syntax into S-expressions as well.

The resulting S-expressions form a Racket program that generates an SMT script. This program internally represents PrintTalk's gadgets as Racket objects, and gadget variables are fields of these objects. A constraint variable expression such as `(cube@x)` is converted into the application `(dynamic-get-field 'x cube)`, using Racket's `dynamic-get-field` procedure to access object fields. A constraint assertion such as `(constraint:<! a b)` is converted into the application `(make-constraint '< a b)`, and similarly a solver expression such as `(solver:+ a b)` is converted into `(solver-expression '+ a b)`, where both applied procedures are implemented by us to return a Racket object that represents the specific element.

4.2 The Z3 Constraint Solver

PrintTalk requires a constraint solver that supports both nonlinear equations (e.g., trigonometric functions for the circular topology) and geometric constraints. Therefore, we use Z3. Some constraints are easy to express in Z3. For example, an equality-constraint `(constraint:=! a b)` in PrintTalk is

translated into a single assert-statement (`assert (= a b)`) in Z3. More effort is required for (more complex) geometric constraints. While Z3 does not support geometric constraints directly, some geometric constraints can be supported by combining other constraints. E.g, a distance-constraint is implemented using the Pythagorean theorem and an angle-constraint is implemented using the law of cosines.

In our current implementation, constraints are solved when the `print:` form is encountered. Constraint solving consists of four steps. First, All constraints that are placed on gadgets are collected. Because gadgets and their subgadgets are organized in a tree structure, this is implemented as a straightforward tree traversal. Second, all collected constraints are converted into an SMT script that is saved to the file. Third, Z3 is called with the SMT script as input, using Racket's built-in `subprocess` procedure to invoke Z3 as a subprocess. The final step consists of examining Z3's output, which is based on S-expressions and therefore straightforward to read using Racket's `read` procedure. If Z3 successfully solves the constraints, the resulting values are assigned to their corresponding constraint variables in PrintTalk. When the constraints are not satisfiable an error is raised.

4.3 Generating 3D Models

The output of a PrintTalk program is an STL file that contains a triangle mesh representation of the gadget. The STL file can be used as input for other tools to render and visualize the 3D model on screen, or to actually print the model in 3D using slicer software [11]. Our Racket code thus generates a scad script which can be sent to OpenSCAD for generating that STL file.

5 Evaluation

In this section, we evaluate PrintTalk by comparing it to OpenSCAD, as it is one of the most widely used programmatic CAD languages. We compare an OpenSCAD script to a PrintTalk program that describes the same 3D model. The model represents a round table with a ball on top of it, as illustrated by Figure 6. The ratio between the diameter of the ball and the diameter of the tabletop is equal to the golden ratio. The ratio between the thickness of the tabletop and the length of the table's legs is also equal to the golden ratio.

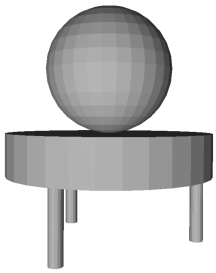

Figure 6. 3D model representing a round table with a ball on top of it.

```
1  module table(x, y, z, dia, height, nrOfLegs){
2   legHt = height / ((1.0 + sqrt(5)) / 2.0);
3   tabtopHt = height - legHt;
4   legZ = z - (tabtopHt/2) - (legHt/2);
5   legRadius = (dia/2) - 10;
6   union(){
7    translate([x, y, z]){
8     cylinder(h=tabtopHt, d=dia, center=true);
9    }
10   for(count = [0 : nrOfLegs-1]){
11    angle = ((2*PI*count)/nrOfLegs) * 180 / PI;
12    cx = x + (cos(angle) * legRadius);
13    cy = y + (sin(angle) * legRadius);
14    translate([cx, cy, legZ]){
15     cylinder(h=legHt, d=10, center=true);
16 }}}}
17
18 module scene(x, y, z, tabDia, height){
19  ballDia = tabDia / ((1.0 + sqrt(5)) / 2.0);
20  ballZ = z + (ballDia/2) +
21     ((height-(height/((1.0 + sqrt(5))/2.0)))/2);
22  union(){
23   translate([x, y, ballZ]){
24    sphere(d= ballDia);
25   }
26  table(x, y, z, tabDia, height, 3);
27 }}
28
29 scene(0, 0, 0, 150, 80);
```

Listing 9. OpenSCAD code representing a round table with a ball on top of it.

5.1 OpenSCAD Script

Listing 9 contains the OpenSCAD code that represents the 3D model illustrated in Figure 6. This code consists of two modules. The `table` module (lines 1–16) represents a round table. The total height of the table is provided as a parameter. Given this height, the thickness of the tabletop and the length of the legs can be calculated (lines 2–3). This calculation makes use of the number determined by $(1.0 + \sqrt{5})/2.0)$, which represents the algebraic notation of the golden ratio. As a consequence, the ratio between the tabletop's thickness and the length of the legs is equal to the golden ratio. The table consists of a tabletop (line 8) and a number of legs, which are placed in a circular pattern under the tabletop by a `for` loop (lines 10–15). The `scene` module (lines 18-27) represents the entire scene (i.e., A table with a ball on top of it). The z-coordinate and the diameter of the ball must be calculated manually, so that the ball is on top of the table and the ratio between the diameter of the ball and the diameter of the table is equal to the golden ratio. Finally, the scene is instantiated by calling the `scene` module with the correct number of parameters (line 29).

5.2 PrintTalk Program

This section discusses a PrintTalk program that describes the same 3D model as the OpenSCAD script discussed in Section 5.1. We first introduce two user-defined constraints in Listing 10. The `golden-ratio` constraint (lines 1–6) constrains two parameters (numbers or constraint variables) so that the ratio between them is equal to the golden ratio. The `on-top-of` constraint (lines 8–11) can be used to constrain the

```
1  (constraint: golden-ratio (a b) ()
2   (constraints:
3    (constraint:>! a 0)
4    (constraint:>! b 0)
5    (constraint:=! (solver:/ a b)
6                   (solver:/ (solver:+ a b) a))))
7
8  (constraint: on-top-of (aZ aHt bZ bHt) ()
9   (constraints:
10   (constraint:=! (solver:- aZ (solver:/ aHt 2))
11                  (solver:+ bZ (solver:/ bHt 2)))))
```

Listing 10. User-defined constraints in PrintTalk.

z-coordinates of two shapes, so that one shape is placed on top of the other shape. The constraint takes four parameters, representing the z-coordinates and the heights of two shapes. Notice that these constraints are not part of the definition of the 3D model that we are describing. Instead, `golden-ratio` and `on-top-of` are general reusable constraints that can be imported and that can also be used in the definition of other 3D designs.

Listing 11 contains the actual PrintTalk code that represents the 3D model illustrated by Figure 6. The code consists of two gadgets, and the constraints defined in Listing 10 are reused in their definition. The `table` gadget (lines 1–13) represents a round table. The total height of the table is provided as an argument. Given this height, the thickness of the tabletop and the length of the legs are determined by constraints (lines 9–10). Other constraints determine the radius of the ring of legs (lines 11–12) and the Z-coordinate of the legs (line 13). The table consists of a tabletop (line 5) and a number of legs, which are placed in a circular pattern under the tabletop by a `ring-repeated:` topology (lines 6–7). The `scene` gadget (lines 15–24) represents the entire scene, i.e., a table (line 20) with a ball (line 19) on top of it. The z-coordinate and the diameter of the ball are represented by constraint variables (lines 22–24). Finally, the scene is converted into an STL file using the `print:` statement (line 26).

5.3 Comparison

While the overall structure of the OpenSCAD code and the PrintTalk code is similar, there are three key differences:

First, PrintTalk provides a `ring-repeated:` topology that makes it straightforward to instantiate gadgets in a circular way. In OpenSCAD programmers are condemned to using a `for` loop with complicated trigonometric calculations.

Second, one of the key features of PrintTalk is its support for constraints. PrintTalk makes it possible to describe the golden ratio by its equation using a user-defined constraint. As a consequence, not all calculations must be performed explicitly by the end-programmer. Again, in OpenSCAD, programmers are condemned to performing calculations using the algebraic representation of the golden ratio.

Finally, in addition to constraints, PrintTalk includes constraint variables. Using constraint variables in the definition of constraints increases the reusability of gadgets, as demonstrated by the example below.

```
1  (gadget: table
2   (x y z dia height nrOfLegs)
3   (legZ legRadius legHt tabtopHt)
4   (script:
5    (cylinder x y z dia tabtopHt)
6    (ring-repeated: x y legRadius nrOfLegs
7                    (cylinder X Y legZ 10 legHt)))
8   (constraints:
9    (constraint:golden-ratio! legHt tabtopHt)
10   (constraint:=! height (solver:+ legHt tabtopHt))
11   (constraint:=! legRadius
12                  (solver:- (solver:/ dia 2) 10))
13   (constraint:on-top-of! z tabtopHt legZ legHt)))
14
15 (gadget: scene
16   (x y z tabDia height)
17   (ballZ ballDia)
18   (script:
19    (named: ball (sphere x y ballZ ballDia))
20    (named: tab (table x y z tabDia height 3)))
21    (constraints:
22     (constraint:on-top-of! ballZ ballDia
23                            z (tab@tabtopHt))
24     (constraint:golden-ratio! tabDia ballDia)))
25
26 (print: (scene 0 0 0 150 80) "table.stl")
```

Listing 11. PrintTalk code representing a round table with a ball on top of it.

Variation: The ball must have a specific size. This example illustrates how PrintTalk's constrained gadgets foster the reusability of 3D designs. We explain how the OpenSCAD and PrintTalk code, as described by Sections 5.1 and 5.2 respectively, is reused to generate a 3D model of which the ball on top of the table has a specific diameter.

The `scene` module (resp. gadget) specifies a parameter to represent the diameter of the tabletop. In contrast, the diameter of the ball is not provided explicitly. Instead, it is derived from the diameter of the table.

In OpenSCAD, we are condemned to calculating the value of the parameter that represents the diameter of the table explicitly so that the ball has the desired diameter. This is shown in Listing 12.

```
1  module scene2(x, y, z, ballDia, height){
2   tabDia = ballDia * ((1.0 + sqrt(5)) / 2.0);
3   scene(x, y, z, tabDia, height);
4  }
5  scene2(0, 0, 0, 100, 80);
```

Listing 12. OpenSCAD code for generating a 3D model where the ball has a specific size.

Listing 13 illustrates how the same can be achieved in PrintTalk, without requiring any explicit calculations. We instantiate the scene (line 3) using a constraint variable `tabDia` (line 1) that represents the diameter of the table. We can retrieve the `ballDia` constraint variable from this gadget, and constrain it to be equal to the desired diameter (line 5).

6 Limitations and Future Work

We envision several avenues for future research:

Maturity of the Prototype. Since Z3 is not a geometric constraint solver, geometric constraints must be converted into a combination of constraints that are supported by Z3.

```
1  (gadget: scene2 (x y z ballDia height) (tabDia)
2   (script:
3    (named: scene (scene x y z tabDia height)))
4   (constraints:
5    (constraint:=! (scene@ballDia) ballDia)))
6  (print: (scene 0 0 0 100 80) "table.stl")
```

Listing 13. PrintTalk code for generating a 3D model where the ball has a specific size.

In future work, PrintTalk may be extended with support for a geometric constraint solver that natively supports these constraints. The implementation can also be improved by making PrintTalk generate STL files directly instead of relying on 3rd party software such as OpenSCAD for this. Finally, topologies can only be added by modifying the underlying Racket code, and not from within PrintTalk itself. In future work, concepts that enable programmers to define their own topologies from within PrintTalk may be introduced.

Error Handling. In the current implementation, all errors are handled at the level of the tool where the error manifests itself. Since PrintTalk is built atop the Racket platform, it would be interesting to investigate how Racket's advanced contract system [6] can be employed to track errors and 'assign blame' to the right code. This is especially relevant in the face of constraint programming because the program location where an error occurs can be very different from the location that is actually responsible for the error. This is an interesting avenue for future research.

Static Analysis. Some of the errors are not really software errors but stem from the fact that the resulting 3D shape does not make much sense in the real world. E.g., a 3D printer automatically inserts supporting legs when printing objects of which the centre of gravity lies too high. It would be nice to preclude such objects from being printed in the first place. Hence, another avenue for future research is to apply static analysis to 'check the laws of physics and stability' on a program before printing happens. Additionally, static analysis could be applied to analyse the cost and time associated with printing the 3D model.

Validation and Evaluation. We have only used PrintTalk for simple designs. Much more experience is needed to test how well it scales to larger and more complex shapes. We also want to perform usability evaluations with end-users to assess the strengths and weaknesses of PrintTalk, and to learn what users think about the language.

7 Conclusion

In this paper, we present PrintTalk, a DSL for programmatic 3D modelling. PrintTalk allows programmers to construct 3D models using the CSG modelling technique. In PrintTalk, CSG components are represented by gadgets. Next to ordinary parameters, gadgets contain constraint variables, to which values can be assigned by the underlying Z3 solver. PrintTalk sits on a sweet spot for combining constraints with imperative programming for designing 3D models. We achieved this by integrating constraint variables and constraints (which can be user-defined) directly into an imperative host language. To evaluate PrintTalk, we compared it with OpenSCAD, one of the most widely used programmatic CAD languages. This comparison demonstrated the advantages of PrintTalk's constraint-imperative approach over purely imperative programming languages.

References

[1] Greg J. Badros, Alan Borning, and Peter J. Stuckey. 2001. The Cassowary Linear Arithmetic Constraint Solving Algorithm. *ACM Trans. Comput.-Hum. Interact.* 8, 4 (Dec. 2001), 267–306. https://doi.org/10.1145/504704.504705

[2] Leonardo de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, C. R. Ramakrishnan and Jakob Rehof (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 337–340. https://doi.org/10.1007/978-3-540-78800-3_24

[3] Tim Felgentreff, Alan Borning, and Robert Hirschfeld. 2014. *Babelsberg: Specifying and solving constraints on object behavior.* Vol. 81. Universitätsverlag Potsdam.

[4] Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, Shriram Krishnamurthi, Eli Barzilay, Jay McCarthy, and Sam Tobin-Hochstadt. 2018. A Programmable Programming Language. *Commun. ACM* 61, 3 (Feb. 2018), 62–71. https://doi.org/10.1145/3127323

[5] Robert Bruce Findler, John Clements, Cormac Flanagan, Matthew Flatt, Shriram Krishnamurthi, Paul Steckler, and Matthias Felleisen. 2002. DrScheme: a programming environment for Scheme. *Journal of Functional Programming* 12, 2 (2002), 159–182. https://doi.org/10.1017/S0956796801004208

[6] Robert Bruce Findler and Matthias Felleisen. 2002. Contracts for Higher-Order Functions. *SIGPLAN Not.* 37, 9 (Sept. 2002), 48–59. https://doi.org/10.1145/583852.581484

[7] Jessie Frazelle. 2021. A New Era for Mechanical CAD: Time to Move Forward from Decades-Old Design. *Queue* 19, 2 (April 2021), 5–16. https://doi.org/10.1145/3466132.3469844

[8] Martin Grabmüller and Petra Hofstedt. 2004. Turtle: A Constraint Imperative Programming Language. In *Research and Development in Intelligent Systems XX*, Frans Coenen, Alun Preece, and Ann Macintosh (Eds.). Springer London, London, 185–198. https://doi.org/10.1007/978-0-85729-412-8_14

[9] Christoph M. Hoffmann. 1989. *Geometric and Solid Modeling: An Introduction.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[10] Gus Lopez, Bjorn Freeman-Benson, and Alan Borning. 1994. Kaleidoscope: A Constraint Imperative Programming Language. In *Constraint Programming*, Brian Mayoh, Enn Tyugu, and Jaan Penjam (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 313–329. https://doi.org/10.1007/978-3-642-85983-0_12

[11] K. Rajaguru, T. Karthikeyan, and V. Vijayan. 2020. Additive manufacturing – State of art. *Materials Today: Proceedings* 21 (2020), 628–633. https://doi.org/10.1016/j.matpr.2019.06.728

[12] Francesca Rossi, Peter van Beek, and Toby Walsh. 2006. *Handbook of Constraint Programming.* Elsevier Science Inc., USA.