ZADÁNÍ BAKALÁŘSKÉ PRÁCE

Název:	Vektorový grafický editor s podporou geometrických pravidel ve Pharo/Bloc
Student:	Vojtěch Balík
Vedoucí:	Ing. Robert Pergl, Ph.D.
Studijní program:	Informatika
Studijní obor:	Teoretická informatika
Katedra:	Katedra teoretické informatiky
Platnost zadání:	Do konce letního semestru 2018/19

Pokyny pro vypracování

1) Seznamte se s vývojovým prostředím Pharo, jazykem Smalltalk a low-level UI frameworkem Bloc.
2) Proveďte konceptuální analýzu vektorového grafického editoru, který bude podporovat zadávání geometrických pravidel a vztahů mezi objekty, inspirujte se v existujících souvisejících řešeních.
3) Implementujte jednoduchý vektorový editor ve Pharo s použitím Bloc podporující základní geometrické útvary.
4) Navrhněte textový jazyk, ve kterém se budou zadávat pravidla pro jednotlivé elementy i diagram celkově.
5) Navrhněte a implementujte řešení pro vyhodnocování a splňování pravidel (solver), komentujte jeho časovou složitost vzhledem k typickému použití. Použijte buď existující vhodnou knihovnu (s případnými úpravami), případně vytvořte algoritmus vlastní.
6) Řešení otestujte a demonstrujte na příkladech pokrývajících možnosti jazyka.
7) Komentujte dosažené výsledky a jejich omezení.

Seznam odborné literatury

Dodá vedoucí práce.

doc. Ing. Jan Janoušek, Ph.D.
vedoucí katedry

doc. RNDr. Ing. Marcel Jiřina, Ph.D.
děkan

V Praze dne 15. ledna 2018

FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Bachelor's thesis

A Vector Graphics Editor Supporting Geometric Rules in Pharo/Bloc

Vojtěch Balík

Department of Theoretical Computer Science
Supervisor: doc. Ing. Robert Pergl, Ph.D.

May 16, 2019

Acknowledgements

I would like to thank my supervisor, doc. Ing. Robert Pergl Ph.D., for the time and energy spent on advising me during the work on this thesis and his positivism throughout. I would also like to thank my parents for the unending patience in their support for me. None of this would be possible without you.

Declaration

In Prague on May 16, 2019 ....

Citation of this thesis

Balík, Vojtěch. *A Vector Graphics Editor Supporting Geometric Rules in Pharo/Bloc.* Bachelor's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2019.

Abstrakt

Při tvorbě strukturovaných kreseb jako jsou diagramy, jsou mezi objekty často zamýšlené geometrické vztahy, nebo-li omezující podmínky, které by měly platit.

V této práci nejdříve shrnuji existující metody řešení omezujících podmínek v interaktivních grafických aplikacích. Posléze implementuji prototyp vektorového editoru ve Pharo Smalltalk, s podporou základních tvarů a lineárních omezujících podmínek, k čemuž využívám algoritmus Cassowary. Dále navrhuji textový jazyk, pomocí kterého se do editoru omezující podmínky zadávají.

Klíčová slova Pharo, Smalltalk, omezující podmínky, interaktivní grafická aplikace, diagram, Cassowary, textový jazyk

Abstract

When creating structured drawings, such as diagrams, it is often desired for certain geometrical relationships, or constraints, to hold among the shapes.

In this thesis, I first review existing approaches to solving constraints in interactive graphical applications. Then I implement a prototype vector editor in Pharo Smalltalk, supporting basic shapes and linear constraints, for which I use the Cassowary solver. I also design a textual language, using which constraints are entered into the editor.

Keywords Pharo, Smalltalk, constraints, interactive graphical application, diagram, Cassowary, textual language

Contents

List of Figures

Introduction

Motivation

When creating structured drawings using vector editors (e. g. UML diagrams), the user often intends for graphical objects to be in a certain geometrical relation with one another. Examples of there relations might be:

- object A to the left of object B,
- A does not overlap B,
- group of objects aligned to a line,
- and so on.

These are referred to as *constraints* in the literature.

Usually, however, there are only limited ways to convey this desire to the editor, and so users have to maintain these relations by themselves manually after a change. This makes it tedious to create visually pleasing diagrams. To ease the process, a system, in which it is possible to declare constraints among objects and which are automatically maintained, is desired.

In this thesis, I explore some of the many approaches developed in the past. After that I describe my prototype implementation of such system in Pharo Smalltalk using the Bloc UI framework, which I call *Aod*.

Goals

1. Familiarize yourself with the Pharo live programming environment, the Smalltalk language and Bloc, a low-level UI framework.
2. Analyze vector graphic editor system supporting constraints among its objects. Take inspiration in existing solutions.

3. Implement a simple vector editor in Pharo using Bloc framework supporting basic geometric shapes.
4. Design a textual language for entering constraints on visual elements into the system.
5. Design and implement a solution to maitain constraints, and comment on it's complexity in typical usage. Either use a suitable existing library, or develop a new algorithm.
6. Test the implementation and provide examples demonstrating its capabilities.
7. Assess the results and discuss their limitations.

Outline

Beginning with the introduction, I briefly summarized the motivation behind this thesis and its goals.

In the Review chapter, I look at the different techniques developed for solving constraints in interactive graphical applications.

Then, in the Analysis chapter, I assess the kinds of constraints that are needed for the subject of this thesis, select a suitable solving technique and discuss how I will use it and its limitations in further detail.

Following that is the Design and Implementation chapter.

In the Validation chapter, I demonstrate the capabilities of the final product on examples.

The Discussion chapter examines the limitations of the prototype implementation and how they can be improved, as well as potential future applications of this thesis' results.

Finally, in the Conclusion chapter, I assess the achieved results.

Chapter 1

Review

Constraints have been used in many graphical applications. In this chapter I will first focus on algorithms, or *constraint solvers*, used in interactive applications. Then I briefly describe technologies I use in my implementation.

Common in vector editors are operations like aligning or equally distributing objects [2], and specialized layouting algorithms in diagramming editors like [3]. However, the assigned relations are not maintained when user directly manipulates objects of the drawing.

Editors with such capabilities exist. At their core, there is an algorithm that enables these capabilities – a constraint solver – which takes in specified constraints and outputs values for constrained variables that satisfy them. Based on those variables the editor then displays visual objects on the screen.

The underlying solver implies many of the editors characteristics, most importantly

- expressiveness, i. e. what kinds of constraints are possible,
- complexity, which is crucial for interactive editing. [4]

In the following sections, I discuss existing approaches to constraint solving in interactive graphical applications.

1.1 Constraint hierarchies

Traditionally, *constraints* describe relations that must be maintained. However, it is often useful to be able to express preferences too and interactive graphical applications are no exception. *Constraint hierarchies* [5] provide a formal framework for this and many solutions adopt it.

Constraint hierarchies enable preferences alongside requirements in constraint systems. Preferred (soft) constraints can be broken, therefore conflicting constraints are possible. There can be an arbitrary number of levels of preferential constraints, where stronger constraints completely dominate weaker ones.

Loosely, a solution to a constraint hierarchy is a valuation for the constrained variables, that satisfies, that

1. all required constraints are satisfied, and
2. there is no *better* solution for non-required constraints.

A *comparator* then defines which solutions are *better*. The definition allows for many different definitions of a comparator, however it requires that the hierarchy is respected – if there is a solution that satisfies all constraints up until some level (going from stronger to weaker ones), then that solution is *better* than solutions that do not.

Comparators differ in two general dimensions;

- local vs. global – whether solutions are compared constraint by constraint or by some global measure, and
- predicate vs. metric – whether the comparator only distinguishes binary satisfied or unsatisfied constraints, or whether it also considers how nearly the constraints are satisfied

The ability to express preferences helps with an issue common to all constraint solvers – *under-constrained* systems of constraints. An under-constrained system is a set of constraints, for which there is no single correct solution – it is ambiguous, which values should be assigned to constrained variables. Consider this one constraint over two variables:

$$X + Y = 0$$

Such system has more than one solution and it is unclear which of the solutions should the solver pick. This is a problem, because as a consequence, unexpected results might be displayed to the user. Constraint-hierarchies-enabled systems add *stay constraints* to each constrained on-screen entity, which, as the name suggests, make variables 'stay at one place'. [4]

A stay constraint is a low priority constraint of the form

$$X = \alpha$$

where α is the value that the constraint solver will assign to X unless a constraint with higher priority forces it otherwise. This disambiguates solutions and ensures that objects do not move unexpectedly.

If there is a stay constraint on Y too however, and their values do not satisfy the equation, one of the stay constraints has to be broken. It is up to the *comparator* to decide which solution to pick.

1.2 Solvers

1.2.1 Local Propagation

Local propagation, thoroughly discussed in [4], is one of the simplest approaches to constraint solving developed. Each constraint is implemented as one or multiple procedures, that are called in order to maintain the constraint. The job of a local propagation algorithm is to figure out which procedures to call for each constraint and in which order to call them. Maintaining the constraints is then only calling the chosen procedures in that order. This makes local propagation solvers very fast.

In its simplest form, only *one-way* constraints are allowed. One-way implies only one procedure per constraint. For example; a constraint $x = y + z$ will only be maintained by setting x in reaction to y and z, i. e. $x \leftarrow y + z$, but not $y \leftarrow x - z$.

With one-way constraints, the preparation phase is equivalent to topologically sorting a directed graph. In this *constraint graph*, vertices represent constrained variables and directed edges represent one-way constraints.

However, this approach cannot handle cycles in the constraint graph, and it is indeed a problem common to all local propagation solvers, as they can only consider one constraint at a time and in isolation. Such cycles arise when multiple constraints interact with the same variables. For example;

$$\begin{gathered} x + y = 100 \\ x = y \end{gathered}$$

To avoid this cycle, these constraints would have to be rewritten to

$$\begin{gathered} x = 100/2 \\ y = 100/2 \end{gathered}$$

This however sacrifices some of the declarative nature of constraints, as the user needs to be wary of creating cycles. Additionally, they cannot always be avoided.

Additionally, there are other limitations to the basic approach.

- Only functional constraints. This means inequality constraints are not possible.
- Only required (hard) constraints, expressing preferential (soft) constraints is not possible.
- Only single-output procedures, i. e. a constraint's procedure can set the value of only one variable.

There are solvers attempting to solve all of these problems.

DeltaBlue [6] supports multi-way constraints (i. e. multiple procedures per constraint) and soft constraints by implementing constraint hierarchies using a locally-predicate-better comparator.

Indigo [7] supports inequalities by propagating intervals instead of values, which are tightened as constraints are applied.

SkyBlue [8] and Ultraviolet [9], are capable of solving some cycles in their constraint graphs. They achieve this by finding subgraphs with cycles and collapsing them into a single constraint. These constraints are then satisfied using a specific solving technique, a sub-solver – for example Gaussian elimination for simultaneous linear equations. Because these subgraphs might relate multiple variables, the main solver has to be able to handle multiple-output procedures for constraints.

To summarize, the advantages of local propagation are

- speed, and
- constraints not limited to numeric domains.

The biggest disadvantage is that it cannot handle cycles without delegating to subsolvers.

1.2.2 Iterative Numeric Solvers

Usually, the constraint satisfaction is transformed into a mathematical optimization problem by expressing the constraints in terms of an error, which the optimization algorithm then minimizes as its objective.

Iterative approaches, like the Newton's method, take an initial guess and try to improve on that to find a solution. These are very general solvers – they allow solving simultaneous non-linear constraints. However, they are relatively inefficient. Another problem is that due to their iterative nature, only local optimums are found (in reasonable times) and the solutions found by the solvers depend on the initial guess. This means that even a small change may lead to an unexpected result.

Iterative numeric solvers are also difficult to implement correctly, due to problems like numerical instability. [4]

The role that these solvers play in the systems using them also differ. Sketchpad [10] uses it as a general fallback in the cases its fast local propagation algorithm fails. Juno [11] allows few simple constraints on points and lines. Using these points and lines, user then specifies the final drawing using *procedural* code, that is executed after constraints have been satisfied.

There have also been attempts to embed constraint hierarchies into solvers using this approach. [12] [13]

1.2.3 Direct numeric solvers

Unlike iterative numeric solvers, direct numeric solvers attempt to find an exact solution instead of approximating. This avoids the difficulties with iterative numeric solvers. However, the trade-off is that less expressive constraints are available. A common restriction is only allowing linear equalities, which can be solved using Gaussian elimination. Simplex algorithm based solutions also allow linear inequalities. [4]

1.2.3.1 Simplex algorithm

Simplex algorithm solves the problem of linear optimization – the task of minimizing a linear *objective function*, subject to a set of linear constraints. The algorithm uses *pivoting* and is split in two phases. Phase I finds an initial *feasible solution* – a valuation of constrained variables such that all constraints are satisfied, phase II finds an *optimal solution* – a feasible solution where the objective function has minimum possible value.

The simplex algorithm, in and on itself, does not allow inequalities. However, each variable in its tableau is guaranteed, from the way the algorithm operates, to be non-negative. Each constraint $x \leq \alpha$, where x is an arbitrary linear expression and α is a constant, can be rewritten as $x + y = \alpha$, where y is a non-negative variable. y is referred to as a *slack variable.*

While the worst case time complexity of the simplex algorithm is exponential, it is polynomial in the average case.

HiRise2 [14], QOCA [15] and Cassowary [16] are all constraint solvers more or less based on the simplex algorithm. [4]

1.2.3.2 Cassowary

Cassowary [16] modifies the simplex algorithm to overcome difficulties which make it unsuitable to use in interactive graphical applications.

First, less crucial issue, is the fact that the simplex algorithm imposes implicit non-negative constraint on all it's variables. Here, Cassowary adopts a solution developed for constraint logic programming languages of using two separate tableaux, one for the simplex variables (e. g. slack and error variables), on which the algorithm operates and the other one for unrestricted variables (i. e. created by user) expressed using the variables in the simplex tableau.

Second issue is how to define the objective function to accommodate constraint hierarchies. Cassowary's objective function is the sum of errors on each non-required constraint. To accommodate constraint hierarchies, e. g. to make errors of stronger constraints have more weight then those of weaker ones, each error (e. g. a value representing how much the current solution deviates from a given non-required constraint) has an associated *weight.* These weights directly corresponds to the strengths associated with constraints and make sure that weaker constraints never out-weight those of stronger constraints, symbolic weights are used (represented as tuples of lexicographically ordered values). This is regarded as the *weighted-sum-better* comparator. In the previously mentioned taxonomy of constraint hierarchy comparators, it is global and metric.

As an example, suppose the following problem, with two levels of non-required constraints: minimize

$$[1,0] \times |x-50| + [0,1] \times |x-30|$$

subject to

$$\begin{aligned} \text{strong } x &= 50 \\ \text{weak } x &= 30. \end{aligned}$$

$[1,0] \times |x-50|$ will always compare larger than $[0,1] \times |x-30|$, with the exception of when $|x-50| = 0$. Hence, in Cassowary, the optimal solution to this problem is $x \leftarrow \pm 50$.

However, using absolute values in the objective function means the simplex cannot be directly applied. Cassowary solves this by introducing *error variables* into each non-required constraint. This is similar to slack variables commonly used in the plain simplex algorithm.

Final issue is efficiency. Similar problems are solved repeatedly, for example when moving an object using a mouse, the constraints stay the same, except the position of the object being moved. Some form of incrementality is needed, so that parts of previous solutions are reused to speed up solving.

First optimization is that Cassowary allows adding and removing constraints while maintaining basic feasible form (i. e. the result of phase I of simplex algorithm).

Another, more essential optimization, is the special handling of edit and stay constraints to edit variables' values. Edit constraints are similar to stay constraints – both are of the form $x = \alpha$, but edit constraints are stronger. To *edit* a variable, Cassowary first adds an edit constraint for that variable. Then, it takes *suggestions* for the new value of this variable and modifies the edit constraint's α and error variables so that it reflects the new value and overall optimal solution is maintained. In the case where feasibility of the solution is broken, dual of the simplex algorithm is used to regain it while maintaining optimality. After each suggestion, Cassowary modifies the stay constraint of that variable to reflect the new value, so that when the edit constraint is removed, the variable does not jump to its old value.

While Cassowary shares the same asymptotic complexity as the simplex algorithm, it manages to cut down a lot of work by being able to work incrementally. While slower than local propagation solvers, numerous practical applications mentioned above have proven it to be sufficiently fast for interactive manipulation.

Cassowary has been used in several different applications:

- GUI layout [1]
- SCWM [17], a window manager
- CSVG [18], an extension to SVG enabling constraints.

It also has been implemented in several languages, including Smalltalk, Java, C++, etc. [2]

There have been attempts to extend Cassowary with disjunctive constraints. The basic idea is to add and remove constraints on-the-fly, so that constraints comprised of disjunctions (e. g. non-overlapping) are satisfied. [19] [20]

1.2.3.3 QOCA

QOCA [15] is a constraint solving library with several algorithms, one of which is the `QCLinIneqSolver`, briefly described here in comparison to Cassowary.

The main difference between QOCA's simplex based linear inequality solver and Cassowary is that in QOCA, strengths are placed on variables, and not on constraints as in Cassowary. The strengths on variables act as stay constraints. It uses a *least-squares-better* comparator, i. e. it tries to minimize the sum of each variable's stay constraint's error squared. The effect is that when there are conflicting priorities, the error among the stay constraints is balanced, whereas Cassowary picks some to break and some to satisfy completely. [4]

1.3 Technologies

Very briefly, I go through the technologies and frameworks used to implement the prototype application.

1.3.1 Pharo

"*Pharo is a pure object-oriented programming language and a powerful environment, focused on simplicity and immediate feedback (think IDE and OS rolled into one)*" [21].

It's most prominent distinction from other more static languages, is that classes, which describe objects and their methods, are objects like any other.

Pharo is essentially *Smalltalk*, with some extensions. It is a very minimal language. Objects have methods and instance variables to keep state. Objects then communicate by sending each other *messages*. Everything in Smalltalk is a message send, from basic arithmetic operations and control structures

[1] `http://ijzerenhein.github.io/autolayout.js/`
[2] `http://overconstrained.io/`

to high level operations on collections of objects. This is made possible with *blocks*, special objects which wrap code to be executed later, similar to lambdas and closures in other languages.

Pharo does not have types. Compatibility is determined only by whether the receiver of a message responds to it (implements a method with that name).

1.3.2 Bloc

Bloc [22] is a "*Low-level UI infrastructure & framework for Pharo*". It is a new framework under active development, but has been becoming more stable and feature complete during my work on this thesis. There are also plans for it to replace Morphic as the main UI environment for Pharo in the future.

Bloc's basic building block is a `BlElement` object, using which visual presentations are created. Each element uses a `BlLayout` to position it's children, e. g. grid layout. To support interaction, elements send notifications about various events on them, e. g. mouse click, position change, etc. Basic shapes, like rectangles, ellipses, polygons and curves, are supported.

Brick [23] is a widget library built on top of Bloc. It is also under active development, but already provides useful widgets like buttons, and, most importantly, a text editor.

1.3.3 SmaCC

SmaCC (Smalltalk Compiler-Compiler) [24] is a tool, that generates LR parsers. A user describes tokens (lexical analysis), that the language is to recognize, grammar (syntactic analysis) of the language and *actions* to perform when a grammar rule is parsed. Actions are regular Pharo code that can access tokens (terminals) parsed by the corresponding grammar rule and results of actions of its sub-rules (non-terminals).

SmaCC can also generate abstract syntax trees (ASTs). Using a special syntax for actions, SmaCC generates classes and methods for nodes of the AST. Basic visitor mechanism is also generated to work with the AST. A disadvantage of this approach is that for most changes to the parser's source code, the AST classes has to be re-generated. This makes making changes directly to AST classes inconvenient. SmaCC allows specifying additional instance variables for nodes of the AST and their class hierarchy in the parser's source code, but not extra behavior. Extending AST nodes' behavior is best done via custom visitors.

Chapter 2

Analysis

2.1 Choosing constraint solver

The choice of the underlying constraint solving technique will strongly affect the capabilities of the final solution [4].

As the main intended use-case of Aod is creating diagrams, the goal is to allow constraints between *predefined* shapes (e. g. UML class diagram components), rather than to allow defining shapes themselves via constraints.

Looking at figure 2.1 and the shapes in it, constraints used in such diagram might include:

- position, e. g. A to the left of B,
- size, e. g. A is as big as B,

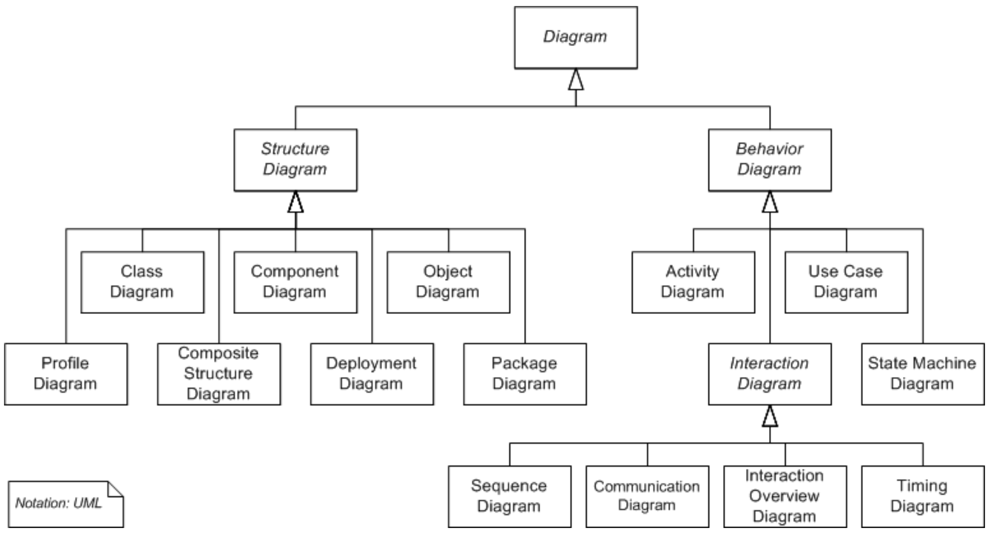


Figure 2.1: UML diagram [1]

- alignment, e. g. a group of shapes placed along a line,
- distribution, e. g. equal spacing between shapes.

Such constraints can be expressed using linear equalities and inequealities (at least considering rectangles, more on this later).

In contrast to these are constraints concerning angles, orthogonality, distance, etc. While these can still be useful, even if not for geometrically constructing shapes, they cannot be expressed using simple linear constraints, and thus powerful general solvers are needed.

Considering the many challenges that come with the iterative numeric solver approach (to briefly reiterate – numerical instability, difficulty of implementation, inaccuracy, slow for interactive manipulation, ...) I find that its benefits do not out-weight the cost and is not worth the effort for Aod's use-case.

Local propagation also allows general constraints, even outside numeric domains. However, their inability to handle cycles is a big issue, as according to [4], cycles seem unavoidable in complex layout problems.

On the other hand, linear constraint solvers do not impose such limitations. If the conjunction of constraints fed into the solver has a solution – no matter *how* they were specified – it will find a solution.

If shapes are limited to rectangles, into which other shapes like polygons or ellipses are inscribed, the system falls into similar domain as GUI layout, where linear constraint solvers, have been used with success.

I decided to base my implementation on Cassowary. It has been successfully used in GUI layout and in a constraint window management system, both mainly dealing with positioning of *rectangular* views. It's Smalltalk implementation has also been ported to Pharo.

2.2 Modelling shapes

Cassowary works with single variables. In this section, I will discuss how shapes will be represented using these variables.

2.2.1 Box

As mentioned above, the main representation of shapes will their bounding rectangle (*bounding box* or just *box* from now on). Looking at it from the other end, the constraints will describe relationships on *boxes*, into which shapes (rectangle, ellipse, polygon, ...) will be *inscribed.*

A box is represented by four variables; *left, top, width, height*, or position and extent.

It is possible to use a different representation for a box – a point representing the top-left corner and another one representing the bottom-right corner.

As far as linear constraints go, these representations are equivalent. One can can get the 'right' variable from the position-extent representation by adding horizontal position and width, and subtract 'left' from 'right' to get the width going the opposite way.

However, there are some practical differences. Stay constraints and edit constraints can only be specified on individual variables.

Changing position of a box with the two-point representation, requires changing both points at the same time. Fortunately, it is possible to edit multiple variables at once and suggest values to both points so that visually, position of the box would change and size would remain the same.

However, there is no way to "insert" a computation like this with stay constraints. With the two-corner representation, there is no way to specify a stay constraint on width or height of a box. One could add a constraint to maintain, for example, width like so

$$\text{Box.bottom\_right.x} - \text{Box.top\_left.x} = \alpha$$

where α is the desired width, and set its strength to that of stay constraints. However, α would not be automatically adjusted after the width has been changed via editing, and so when the editing cycle would end, and the stronger edit constraints removed, the box's width would return to α, disregarding the edit. This would render interactive visual editing of that box impossible. To change the width one would have to first remove the old width constraint and then add a new one with the desired α.

I use the position-extent representation. I also insert additional constraints – *invariants* – to ensure width or height cannot go negative.

2.2.2 Line

One shape where modelling it with a box would be too awkward is a line. Those are modelled them using two points – *start* and *end*. No invariants are needed for lines.

2.2.3 Other

Other models are of course possible, but it might be difficult to express useful constraints using them.

For example, a polygon could be modelled by a point for each of it's vertices, instead of being inscribed into a box. However, with linear constraints, there are limited capabilities to express which shape should these points form. To express more complex shapes (like a pentagon), the relative positions of these points would need to be "hard-coded" via constraints (i. e. to change the shape, these constraints would need to change). If the model were to be compatible with boxes, the left-, top-, right-, bottom-most point would need to stay the same, as constraints cannot be expressed on the minimum or the maximum of a set of variables.

With these limitations, there is not much advantage over inscribing into a box.

2.2.4 Constraints using bounding boxes

Looking back to the diagram in figure 2.1 and the potential constraints, here I will show how some of those constraints can be defined with what I have established so far.

Relative positioning of shapes in terms of a horizontal or vertical direction. E. g. keeping box A above box B is achieved using the following constraint:

$$A.\text{top} + A.\text{height} \leq B.\text{top}$$

.

The position of a box (or any point) can be aligned to an arbitrary line by relating it's horizontal and vertical positions, e. g. to align to a diagonal:

$$A.\text{top} = A.\text{left}$$

.

More often, it is desired to align multiple shapes to same horizontal or vertical lines. This can be done by equating their *left* or *top* anchors, and the boxes can then be moved with this alignment maintained.

A slightly more complex constraint is distributing variables evenly. Consider four boxes, A, B, C, D, and two constants, α and β. To place, for example, their *left* variables evenly, starting from α and ending at β, the following constraints would be used:

$$\begin{aligned}
A.\text{left} &= \alpha \\
B.\text{left} &= A.\text{left} + \frac{\beta - \alpha}{3} \\
C.\text{left} &= B.\text{left} + \frac{\beta - \alpha}{3} \\
D.\text{left} &= \beta
\end{aligned}$$

2.3 Limitations of linear constraints and Cassowary

2.3.1 Expressiveness

Using Cassowary means that only linear equalities and inequalities are available, so for example a general distance constraint is not possible, as it is quadratic. It is however possible to define distances between along some *line*, therefore alignment constraints with spacing are feasible. Relating sizes of objects is also possible, as long as these relations are linear.

Important to note is also the fact that only conjunctions of constraints are possible (though this is not limited to Cassowary). This means that, as is, expressing non-overlap constraints is not possible. More generally, the area in which a point can be constrained to lie is a *convex polygon*.

To backtrack a little, while local propagation solvers cannot handle disjunctions either, non-overlapping between *two* shapes could be implemented by a single constraint, as the procedures enforcing constraints can contain arbitrary code. However, having a group of more than two shapes not overlap, which would likely be the desired use of non-overlapping, would necessarily lead to cycles in the constraint graph and hence end up not solvable by local propagation either.

2.3.2 Constraint conflicts

Conflicting constraints are a set of constraints, where there is no solution that satisfies all of them. If constraints of required strengths are in conflict, Cassowary fails. If they are of mixed strengths, Cassowary chooses to satisfy to stronger ones. If they are of the same non-required strength, Cassowary chooses basing on it's internal state, which, from the outside, is effectively at random. (There are ways to influence this, but results are not guaranteed.)

Consider two boxes A and B, with stay constraints of the same strength on each of their variables. Furthermore, there is a required constraint that A is left of B:

$$\text{A.left} + \text{A.width} \leq \text{B.left}$$

What happens in the solver when B is moved left far enough so that the shapes collide? The value of one of the other two variables needs to change so that the constraint is satisfied. B.left is being edited and so is *strongly* constrained to the suggested value. A.left and A.width are kept in place only by weak constraints. To satisfy the constraint, Cassowary will either move A to the left, or shrink its width. (With QOCA, the change would be split between the two variables.) To make Cassowary choose one over another, one solution would need to be *better* than the other, i. e. one of the variables could have a stronger stay constraint.

Nevertheless, there cannot *practically* be an individual strength level for each constraint, mainly for performance reasons. (Each additional level means one more additional comparison to compare symbolic weights.) When creating a drawing, such conflicts happen often, and the more variables are interconnected, the more possibilities of constraints, which the solver can break in the face of a *single* edit, there are. A possible remedy is for the user to add a required constraint, to "lock" some property of the drawing they are satisfied with at an absolute value. In the example from previous paragraph, this could be: A.width = 50.

2.4 Textual language

The textual language has to be able to talk about shapes' models and the variables from which they're composed.

Since Cassowary will be the solver used, the basis of the language will be the following six operations that form linear constraints.

$$+, -, *, /, =, \leq$$

The basic data types will be constrainable variables and numeric constants. To be able to use variables, there must be a way to access models of shapes and access the variables of which those models are composed.

With this, it is possible to express any linear constraint. However, the language should also provide more data types, like points, and descriptive ways to specify common constraints on those types, like the previously mentioned distribution, to ease the process of specifying constraints.

Furthermore, the DSL should also provide a programmable API, so that other ways of specifying constraints, other than textual, are possible in the future (e. g. via snap-dragging).

Chapter 3

Implementation

3.1 Overview

The implementation is split into four packages:

- `Aod`, the main package with most of the logic,
- `Aod-Compiler`, that contains code related to parsing the textual language,
- `Aod-UI`, that contains classes concerning GUI,
- and `Aod-Tests`. [3]

3.2 Main classes

At the heart of Aod is the `Drawing` class [4]. A drawing holds shapes that the user sees, and a `Program` – a description of the relationships between the drawing's shapes that should hold, a collection of constraints. Its main job is to maintain the consistency among shapes and the program. Invariants that must hold within a drawing are:

- shapes' IDs and names must be unique,
- program must be *valid* (more on that below)

[3]Additionally, there is also the Cassowary package in the project. It is the Pharo port of the original Smalltalk implementation, which can be found at `https://github.com/Ducasse/Cassowary`, with some minor fixes I added.

[4]Pharo does not have namespaces, so classes from different projects are prefixed to avoid name collisions and distinguish them from the rest of the system. For example `Bl` for Bloc's classes and `Aod` for Aod's classes. When I talk about Aod's classes, I will omit the prefix.

`Drawing` also uses a `ConstraintSolver` to maintain the program's constraints.

A `Shape` represents visual objects, such as rectangles, ellipses, lines. A shape also knows about its various properties, such as ID, name, color. Most importantly, it also has a `Vessel`, which determines its spatial properties (i. e. position and size).

A `Vessel` is the constrainable model (e. g. box or a line) of a shape I described in the section 2.2. [5]

A `Program` is a collection of constraints. It's validity in and on itself corresponds to the validity of its individual constraints. In relation to a drawing, shapes referenced in the programs constraints must exist in the drawing. The drawing's constraint solver also must be able to maintain the program's constraints, i. e. they must be linear and satiable (no conflicting constraints).

`ConstraintSolver` object's responsibility in a `Drawing` is to take constraints from both vessels (stay constraints and invariants) and the program, and maintain them. It is the single authority on the values of anchors in vessels, i. e. all changes to shapes' spatial properties in a drawing must be performed through its constraint solver.

A `Constraint` is the intermediate representation. The textual language described later is the front-end and Cassowary-specific constraints (and its simplex tableau) are the back-end. It holds the description of the relationship to maintain, described using *actors*, and a strength. Currently, three levels of strengths (in addition to the *required* strength) are supported, mapping directly to Cassowary: *weak*, *medium* and *strong*.

3.3 Types

Types represent objects of the language. To represent them, I use Pharo classes, and therefore, apart from a few exceptions, instances of those classes represent their data.

The only method common for all types is a method to check type compatibility, which is mapped directly to the class hierarchy of the types. This enables simple subtyping in the language. A type (i. e. a class) is compatible with another type if it is the same type or it is a subtype (i. e. a subclass) of said type.

The most basic type is `Anchor`, from which other types are composed. Those types are:

- `Point`,
- `Span`,

[5]"Model" and "variable" are quite broad as names for classes, and so in the implementation I use the names "vessel" and "anchor", respectively. Vessel "carries" a shape around and its movement can be limited with anchors.

- and `Vessel`, of which there are currently
    - `BoxVessel`,
    - and `LineVessel`.

A `Span` represents horizontal or vertical line segments. For example, the left and width anchor together form a horizontal span of said box. They are useful in many cases, for example to talk about spacings between boxes, or to specify actors on horizontal and vertical properties in a generic way.

For batch declaration of constraints, there is a `List` type. In addition to these, `Constraint` and `Actor` can act as types too.

3.4 Actors

3.4.1 Overview

Actors can be thought of as macros. They define an operation that takes one or more arguments and returns a result. Generally, the operation is expressed as a composition of other actors. They are represented as classes. Instances of actors represent nodes of a tree, where the children of a node represent arguments for the actor. This tree is then used to describe the relation of `Constraint` objects.

There are four main categories of actors:

- transformers, used for destructuring objects in the language, e. g. taking the x-coordinate of a point or the top anchor of a box,
- operators, which take two or more objects and return a new one, e. g. addition or multiplication,
- relators, which form the relation of a constraint, e. g. equality, object A is left of B,
- and constructors, to create new points, spans and lists.

However, note, that this is just a naming convention used to sort-out the many actors that are in the system. Same rules apply to all actors.

3.4.2 Data bindings

A special case of actors are data bindings, which bridge the gap between regular actors and the data they operate on. They take 0 arguments and are always at the bottom (leaves) of the actor tree.

For the `Drawing` to be able to properly maintain consistency between it's shapes and program, the data bindings in the constraints of a *program* need to bind shapes, not just their vessels or their components. For example, a shape cannot be removed if there are currently constraints in the program referencing it.

3.4.3 Expansions

The operation of an actor is defined by specifying an actor tree that represents the same operation. I call the transformation of an actor into this tree an *expansion*.

Of course, expansions have to stop somewhere, some actors must actually perform some operation. *Atomic actors* are actors that cannot be further expanded. They are the basic anchor operations; $+, -, *, /, \leq, =$, data bindings, and transformers, that directly destructure vessels, e. g. `Box.position` is atomic, but `Box.left` is not (or rather needn't be), as it is equivalent to `Box.position.x`.

3.4.4 Defining a new actor

An actor declares the number of arguments it takes, it's argument types and it's return type in `#argumentTypes`, `#argumentCount` and `#returnType` class methods respectively. The essential part of an actor is the `#expandOn:` class method. Finally, `#identifier` declares the string that will be used to reference this actor from text.

The expansion, is defined as a stack operation. This means that the actor tree, into which an actor expands, is described in post-fix form.

Generally, an `#expandOn:` method

- first pops its arguments off the stack,
- and then pushes the arguments and other actors on the stack.

. There are three ways to push to the generator stack:

- `#push:`, which expects an actor and sends it the `#expandOn:` message,
- `#raw:` places the argument provided on top of the stack as is,
- and `#length:`, which is used specifically when constructing the variably long *lists*.

As an example, 1 shows definition of these methods for an actor representing point-wise addition. It will get the X part of a point and sum it with the X part of the other point. Then it will do the same for the Y part. Finally, it will construct a new point.

```
AodPointAddOperator class>>#argumentCount
    ^ 2

AodPointAddOperator class>>#argumentTypes
    ^ { AodType point . AodType point }

AodPointAddOperator class>>#returnType
    ^ AodType point

AodPointAddOperator class>>#expandOn: aGenerator
    | left right |
    "first, pop arguments of stack"
    right := aGenerator pop.
    left := aGenerator pop.

    "second, push to stack accordingly"
    aGenerator
                raw: left;
            push: AodPointXTransformer;
                raw: right;
            push: AodPointXTransformer;
        push: AodAnchorAddOperator;
                raw: left;
            push: AodPointYTransformer;
                raw: right;
            push: AodPointYTransformer;
        push: AodAnchorAddOperator;
    push: AodPointConstructor

AodPointAddOperator class>>#identifier
    ^ #pointAdd
```

Listing 1: Expansion method for point-wise addition

3.4.5 Generating constraints

When a constraint is added to a solver, it needs to be transformed, or generated, into Cassowary's own format. This is done by pushing the nodes of the constraint's actor tree onto the generator's stack accordingly (the same way as in `#expandOn:` methods). After all actors have been expanded and atomic actors performed, the stack will contain one or more items, each of which is a constraint in Cassowary's own format, ready to be added.

As I mentioned, the result of an expansion can be more than one back-end specific constraint. This is because some relators, e. g. point-wise equality, must be expressed using multiple atomic constraints.

3.5 Language grammar and syntax

The textual definition of a program and constraints is closely related to the internal `Program` and `Constraint` objects. So much in fact, that the textual version of the constraints can be reconstructed from their `Constraint` object counterparts.

This was intentional, so that in the future, if constraints are added using other means (e. g. the previously mentioned snap-dragging), they can be displayed to the user in text and treated the same way as other constraints of the program.

The starting points of each constraint are the shapes, whose *anchors* are being constrained. To access these from the program, *shape references* are used. A '`#`' character followed by an integer represents an ID reference, e. g. `#33`. It's return value is the shape's *vessel*. Another way to access shapes of a drawing from the program is to reference its name. A '`@`' is used to denote a name reference – e. g. `@circle2`.

From the shapes' vessels, actors are used to build up constraints. Consider the following constraint:

```
isLeftOf(#1, #2) : required;
```

An identifier `isLeftOf` with arguments inside brackets after it represents an actor invocation. Any *actor* in the system can be invoked from the text this way. However, for convenience, there are other ways.

```
#1.left + #1.width <= #2.left;
```

This example shows, that anchor operators can also be invoked using their respective symbols in infix notation. Single argument actors (usually transformers) can be expressed by appending '`.`' to its argument followed by the actor's identifier.

Note that in the second constraint, strength is left unspecified. In that case, the constraint will have a *required* strength.

There is also special syntax to construct points and spans. For points,

```
{#1.left, #1.top}
```

is equivalent to

```
#1.position
```

For spans,

```
#1.left -> #1.right
```

is equivalent to

```
#1.horizontal
```

They can also be constructed by using the general actor invocation syntax, i. e. `point(#1.left, #1.top)` and `span(#1.left, #1.width)`.[6] There is also syntax for constructing lists: `[#1, #2, #3]`. List's constructor actor cannot be called directly from the text, as it needs special treatment to handle its variable number of arguments.

All operators in the language are left-associative, except for `<=`, `>=` and `=`, which can only appear once in a constraint. Their precedence is as follows in ascending order:

```
=, <=, >=
->
+, -
*, /
.
```

I use SmaCC to implement, or rather generate, parser for the textual language. Not many other options exist in Pharo. A notable one is PetitParser [25], a framework for building parsers, however it is more low-level and not as complete solution as SmaCC.

3.6 Stay constraints

There are implicit stay constraints for every vessel in the drawing. Without that, shapes would be placed more or less unpredictably and editing with Cassowary would be impossible (there would be nothing holding the shape in the position it was moved to).

This can be manually overridden from the program for individual shapes (or rather vessels) and their components, like so

```
stay(@circle1.position) : medium;
```

The overridden stay constraint does not get removed – if the new one is stronger, it overpowers the old one and the effect is the same. This is to easily ensure that a stay constraint is present on all anchors (i. e. Cassowary variables) at all times.

All stay constraints use have the *weak* strength, except for box's extent, where I opted to use stronger stay constraints (by one level, i. e. *medium*) by

[6]A span is internally represented by a start anchor and a length anchor, but it is usually more convenient constructing it with a start and an end anchor, so the `->` operator does just that.

default. During my experimentation with the system, it always seemed more intuitive for the size of a box to stay the same.

3.7 GUI

The GUI has three main parts:

- the canvas, where shapes are displayed,
- the program editor, where constraints are entered,
- and property list, where properties of the selected shape are shown.

The Bloc low-level UI framework is used for all Aod's visual elements.

GUI elements communicate changes to the model, i. e. the `Drawing` object, using commands, e. g. `SetProgramCommand`. The `Drawing` object sends announcements about the changes in its state and GUI components listen to these announcements and change what they display accordingly.

To position elements representing Aod's shapes accordingly, I use a custom layout strategy, that simply positions and resizes the elements according to the information in the vessels of their respective Aod shapes (to which each such element holds a reference). Bloc's layout strategies work with bounds (i. e. encompassing rectangle, bounding box) of an element, hence incompatible vessels like the `LineVessel` need first be converted.

3.7.1 Handles and editing shapes

Editing a shape must mimic the procedure used in Cassowary to edit variables. First, a shape to be edited must be declared, so that edit constraints can be added to Cassowary. Then, suggestions can be made via *handles*. Finally, the editing must be explicitly ended, so that the no longer needed edit constraints are removed from Cassowary.

A `Handle` object facilitates editing of a vessel in some way, e. g. a handle to edit the size of a box. When a shape is edited, all of it's vessel's anchors are edited at the same time, disregarding which part of it is going to change. The job of a handle is to translate a point input, usually coming from mouse pointer position, into suggestions to the individual anchors, so that the desired effect is achieved. If some anchors are not meant to change, their current values must be suggested back for them.

Editing also the anchors of the vessels, that are not meant to change, has the side effect of giving them temporarily, for the duration of the edit, stronger pseudo stay constraint. This is useful as the shape being edited will stay the same in favor of other shapes when constraints conflict.

Currently, there are only handles for points – position and extent for boxes, and start and end for lines. Visually, they are represented as small circles on

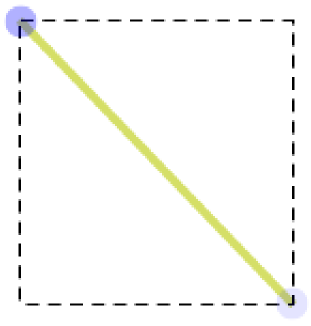


Figure 3.1: Handles of a selected line

the currently selected shape and the whole editing process is performed by clicking on and dragging the visual handles. Figure 3.1 shows a selected line with it’s two handles.

Chapter 4

Validation

4.1 Testing

All tests can be found in the `Aod-Tests` package. I used the Smalltalk's classic SUnit unit-testing framework [26] and focused on the fundamental parts of the implementation, namely constistency of a `Drawing` and correct translation of constraints.

There are also examples of usage of UI components in the `Aod-UI` package, some of which also perform tests.

4.2 Example of creating a binary tree

In this section, I will demonstrate working with the application by walking through the creation of a binary tree diagram. Code equivalent to this example can be found in the `AodElementExample` class. Figure 4.2 shows the screenshot of the final result.

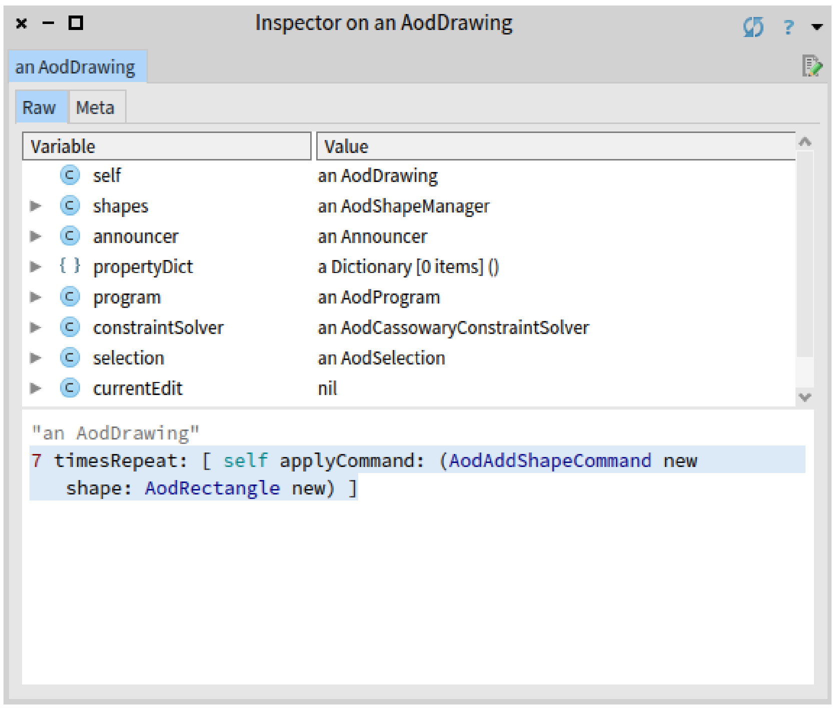
Start with opening an empty drawing in the GUI

```
AodElement openOn: AodDrawing new
```

First, shapes need to be added. The tree will have three levels, so seven nodes. Aod does not yet have a way to add shapes from the GUI, so they have to be added manually. Open inspector on the drawing object being edited by clicking on the "drawing" button and type the following code into it's evaluation window, highlight the code and evaluate it. Figure 4.1 shows screenshot of the inspector with the code highlighted.

```
7 timesRepeat: [ self applyCommand: (AodAddShapeCommand new
    shape: AodRectangle new) ]
```

This creates seven randomly colored rectangles with IDs 1–7 (assuming a fresh instance of a `Drawing`), all with the same position and size. Shapes can



Variable	Value
self	an AodDrawing
shapes	an AodShapeManager
announcer	an Announcer
propertyDict	a Dictionary [0 items] ()
program	an AodProgram
constraintSolver	an AodCassowaryConstraintSolver
selection	an AodSelection
currentEdit	nil

```
"an AodDrawing"
7 timesRepeat: [ self applyCommand: (AodAddShapeCommand new
   shape: AodRectangle new) ]
```

Figure 4.1: Inspector on a drawing with code to add shapes

be (re)named and referenced from program by name. Let's do that for the root node. Again, in the drawing inspector's, enter and evaluate the following.

```
self applyCommand: (AodRenameShapeCommand new
    "select the shape to rename by ID"
    reference: #1 asAodReference;
    name: 'root')
```

With shapes in place, we can go back to the editor window and start adding constraints. First, align shapes in each level to have the same vertical position.

```
// middle level
equal_anchors(
    map(
        [#2, #3],
        top));
// bottom level
equal_anchors(
    map(
        [#4, #5, #6, #7],
        top));
```

Next, it define the order of shapes in a level.

```
sequenced(
    map(
        [#2, #3],
        horizontal));

sequenced(
    map(
        [#4, #5, #6, #7],
        horizontal));
```

Levels should also be ordered. Since the top anchors are aligned in levels, this can be somewhat achieved by specifying the order on individual shapes in the levels.

```
sequenced(
    map(
        [#1, #2, #4],
        vertical));
```

There is one deficiency. If the height of node `#3`, is changed to collide with shapes of the level below, their positions will not get adjusted. To fix this, one more constraint is needed.

```
isAfter(#3.vertical, #4.vertical);
```

A nice property of a tree to have, is that the center of a parent and the centers of it's children nodes are aligned. [7]

```
#1.h.middle = (#2.left -> #3.right).middle;
#2.h.middle = (#4.left -> #5.right).middle;
#3.h.middle = (#6.left -> #7.right).middle;
```

Another beautification might be that the spacing between the nodes in the lowest level is equal in all cases.

```
equal_anchors(
    map(
        spacings(
            map(
                [#4, #5, #6, #7],
                horizontal)),
            length));
```

[7] `h` and `v` are shortcuts to `horizontal` and `vertical` respectively

At this point, the user can notice that it is quite impossible to predict how exactly will shapes move in response to editing one of them. As I mentioned in 2.3.2, this happens many variables are interconnected. To help with this, set the spacing to a constant value. Since they are all constrained to be equal, only one needs to be constrained.

```
spacing(#4.h, #5.h).length = 40;
```

Finally, add lines that will connect the nodes, by again, opening the drawing inspector and applying commands. Here, it is also shown, how shapes can have their properties set before they are added.

```
1 to: 6 do: [ :index |
    self applyCommand: (AodAddShapeCommand new
        shape: (AodLine new
            propertyAt: #color put: Color black;
            propertyAt: #name put: 'line', index asString)) ]
```

Then add constraints, to connect their ends to shapes. [8]

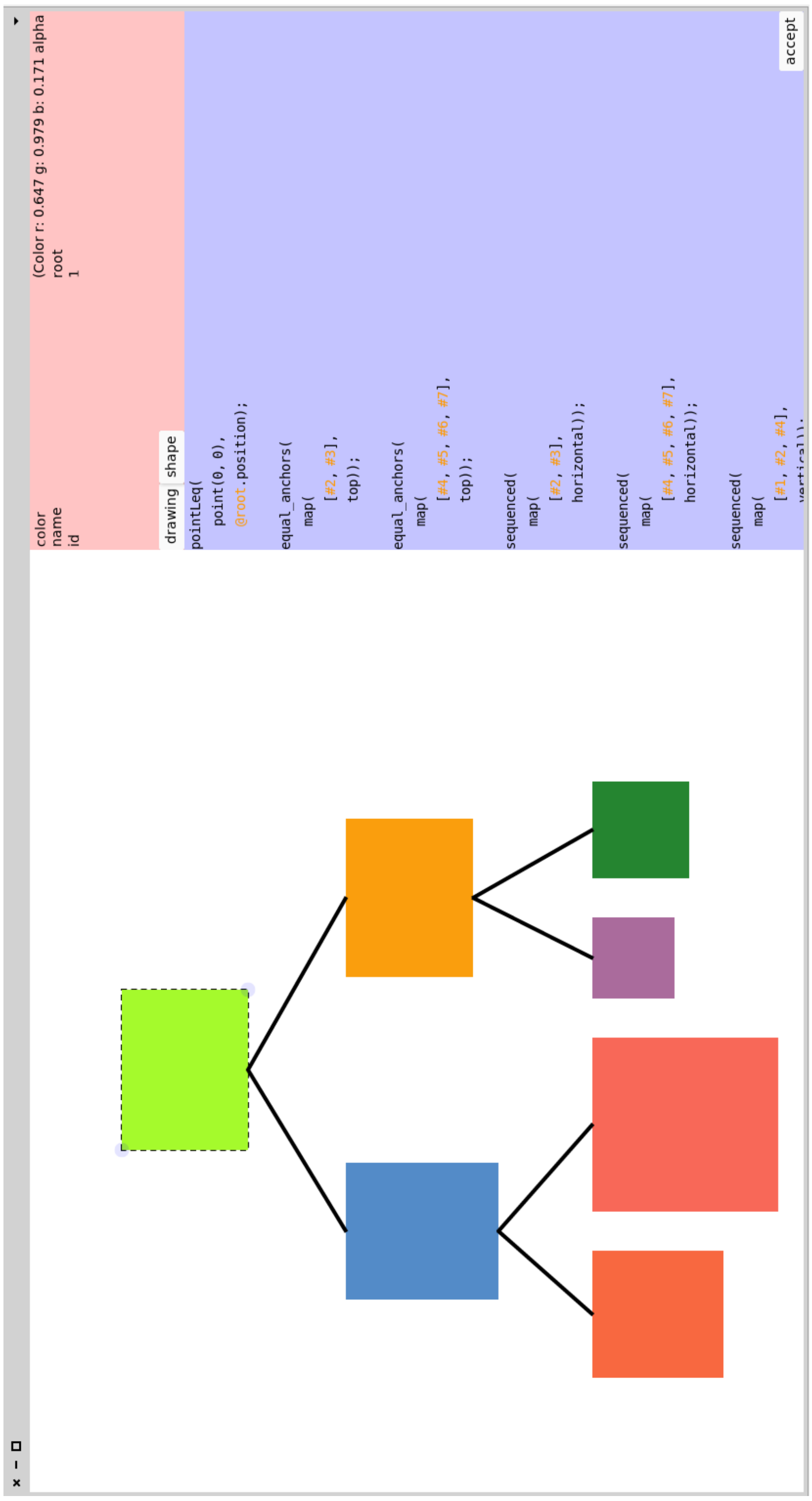
```
connect_points(
        @line1,
        point(
                @root.h.middle,
                @root.v.end),
        point(
                #2.h.middle,
                #2.v.start));

connect_points(
        @line2,
        point(
                @root.h.middle,
                @root.v.end),
        point(
                #3.h.middle,
                #3.v.start));

...
```

Figure 4.2 shows the screenshot of the final result.

[8]I experienced problems when displaying "{" and "}" characters with Brick's editor, which are the special syntax for creating points. To get around this, I refer to the point constructor actor directly.



```
color    (Color r: 0.647 g: 0.979 b: 0.171 alpha
name     root
id       1
```

```
pointLeq(
    point(0, 0),
    @root.position);

equal_anchors(
    map(
        [#2, #3],
        top));

equal_anchors(
    map(
        [#4, #5, #6, #7],
        top));

sequenced(
    map(
        [#2, #3],
        horizontal));

sequenced(
    map(
        [#4, #5, #6, #7],
        horizontal));

sequenced(
    map(
        [#1, #2, #4],
        vertical));
```

Figure 4.2: Final result of binary tree example

Chapter 5

Discussion

5.1 Limitations and Future Work

The effect constraints have on the drawing and its shapes can be confusing. In the end, every constraint is composed of Cassowary variables, and as long as the constraints are satisfiable, Cassowary will assign some values to those variables. It is however difficult to tell, especially when there are many constraints, if a change in a program had the desired effect.

Therefore, the main priority for Aod in the future is to attempt to alleviate this difficulty and make working with constraints easier.

5.1.1 Visualizations

A large topic to explore is how to visualize constraints, or perhaps even edit and create them visually.

While displaying arbitrary linear constraints meaningfully would be difficult, if they are expressed using the language's higher level constructs, these visualizations can be much more specific. For example, span as an arrow between its two anchors.

As I described in section 2.3.2, it is important to add required constraints on properties the user is satisfied with. Currently, however, there are no tools to help with this. User has to manually enter the required constraints and change the values by hand if needed.

Though editing required constraints is not possible with directly Cassowary, [20] shows that it is possible to simulate that by adding and removing constraints on-the-fly and maintain interactive speeds (if not overused). This combined with constraint visualizations could provide a way to visually edit (or rather *change*, to avoid confusion with edit constraints used to change the values of variables) constraints and their parameters.

5.1.2 Textual language limitations

Representing types as Pharo classes is convenient, however it is also limiting. For example, parametric types (e. g. a list of boxes) cannot be easily implemented. (Each instantiation would need its own class to represent the type.)

Some form of generics can also be useful. For example, it is errorneous to create a span with one anchor representing a horizontal dimension and the other a vertical one. However, it cannot be ensured in the current implementation without duplicating code concerning spans for the vertical and horizontal case.

Allowing overloading of actors' identifiers is another useful potential addition. Sometimes an identifier makes sense for multiple different actors, like `start` and `end` for line and span. Overloading would also allow operators to be easily reused, e. g.

```
#1.size = #2.size
```

instead of

```
pointEq(#1.size, #2.size).
```

5.1.3 Constraints

In terms of constraint expressiveness, combining Cassowary with local propagation à la SkyBlue or UltraViolet is a possibility. One example of where that might be useful are lines that connect shapes, e. g. a constraint that connects the closest points of two ellipses. This is not possible with plain linear constraints, but is no problem for local propagation's arbitrary methods. Such constraints do not usually create cycles in the constraints, as shapes' properties do not need to depend on the properties of lines connecting them. This could also increase performance. Solving via local propagation is very fast and the number of variables and constraints in Cassowary's tableau would be reduced.

5.2 GUI editor

An idea I would like to explore in the future, is making Aod into a tool, where the user designs the layout of their GUI in the same way as they would use Aod now, and then generate a custom Bloc layout strategy (`BlLayout`) that maintains specified constraints the same way.

Conclusion

The goal of this thesis was to analyze and implement a basic vector editor with support for constraints in Pharo using the Bloc UI framework, where constraints were to be entered via a textual language.

I started by reviewing existing approaches to solving constraints in interactive graphical applications.

The prototype editor, which I call Aod, is based on the Cassowary linear constraint solver. It supports basic shapes, whose spatial properties are modelled using constrainable variables. These models and their variables can then be accessed and constrained from the textual language. Currently implemented models are *box*, into which rectangles and ellipses are inscribed, and *line*.

Apart from variables and basic operations on them (i. e. $+, -, *, /, \leq, =$), constraints can also be specified on types composed of variables, like points and spans (e. g. the horizontal or vertical dimensions of a box). For batch declarations of constraints, a list type is also supported. Custom operations for the language can be easily defined, albeit only outside the textual language, by composing existing operations in the system.

Fundamental parts of the implementation were tested and I demonstrated the usage of the editor on a binary tree example.

Finally, in the chapter 5, I described Aod's limitations and presented possible ways of alleviating them. I also described a possible alternative use-case for Aod – GUI layout editor.

The code of the prototype implementation can be found at and installed from the faculty's gitlab instance. [9]

[9]https://gitlab.fit.cvut.cz/balikvo1/aod

Bibliography

[1] Merson, P. UML diagram. [Online], 2011, [accessed 2019-05-15]. Available from: `https://commons.wikimedia.org/w/index.php?title=File:Uml_diagram2.png&oldid=321701704`

[2] The Inkscape Team. Inkscape. [Online], 2004, [accessed 2019-05-16]. Available from: `https://inkscape.org/`

[3] yWorks. yEd. [Online], [accessed 2019-05-16]. Available from: `https://www.yworks.com/products/yed`

[4] Badros, G. J. *Extending Interactive Graphical Applications with Constraints*. Dissertation thesis, University of Washington, 2000. Available from: `http://www.badros.com/greg/papers/gjbadros-dissertation.pdf`

[5] Borning, A.; Duisberg, R.; et al. Constraint Hierarchies. *SIGPLAN Not.*, volume 22, no. 12, Dec. 1987: pp. 48–60, ISSN 0362-1340, doi:10.1145/38807.38812.

[6] Freeman-Benson, B. N.; Maloney, J. The DeltaBlue algorithm: An incremental constraint hierarchy solver. In *Computers and Communications, 1989. Conference Proceedings., Eighth Annual International Phoenix Conference on*, IEEE, 1989, pp. 538–542.

[7] Borning, A.; Anderson, R.; et al. Indigo: A Local Propagation Algorithm for Inequality Constraints. In *Proceedings of the 9th Annual ACM Symposium on User Interface Software and Technology*, UIST '96, New York, NY, USA: ACM, 1996, ISBN 0-89791-798-7, pp. 129–136, doi:10.1145/237091.237110.

[8] Sannella, M. Skyblue: a multi-way local propagation constraint solver for user interface construction. In *Proceedings of the 7th annual ACM*

symposium on User interface software and technology, ACM, 1994, pp. 137–146.

[9] Borning, A.; Freeman-Benson, B. Ultraviolet: A Constraint Satisfaction Algorithm for Interactive Graphics. *Constraints*, volume 3, no. 1, Apr 1998: pp. 9–32, ISSN 1572-9354, doi:10.1023/A:1009704614502.

[10] Sutherland, I. E. Sketchpad: A man-machine graphical communication system. Technical report UCAM-CL-TR-574, University of Cambridge, Computer Laboratory, Sept. 2003.

[11] Nelson, G. Juno, a constraint-based graphics system. In *ACM SIGGRAPH Computer Graphics*, volume 19, ACM, 1985, pp. 235–243.

[12] Hosobe, H. A Modular Geometric Constraint Solver for User Interface Applications. In *Proceedings of the 14th Annual ACM Symposium on User Interface Software and Technology*, UIST '01, New York, NY, USA: ACM, 2001, ISBN 1-58113-438-X, pp. 91–100, doi:10.1145/502348.502362.

[13] Hosobe, H. A Hierarchical Method for Solving Soft Nonlinear Constraints. *Procedia Computer Science*, volume 62, 2015: pp. 378 – 384, ISSN 1877-0509, proceedings of the 2015 International Conference on Soft Computing and Software Engineering (SCSE'15).

[14] Hosobe, H. A simplex-based scalable linear constraint solver for user interface applications. In *2011 IEEE 23rd International Conference on Tools with Artificial Intelligence*, IEEE, 2011, pp. 793–798.

[15] Marriott, K.; Sen Chok, S. QOCA: A Constraint Solving Toolkit for Interactive Graphical Applications. *Constraints*, volume 7, no. 3, Jul 2002: pp. 229–254, ISSN 1572-9354, doi:10.1023/A:1020513316058.

[16] Badros, G. J.; Borning, A.; et al. The Cassowary linear arithmetic constraint solving algorithm. *ACM Transactions on Computer-Human Interaction (TOCHI)*, volume 8, no. 4, 2001: pp. 267–306.

[17] Badros, G. J.; Nichols, J.; et al. SCWM: An intelligent constraint-enabled window manager. In *Proceedings of the AAAI Spring Symposium on Smart Graphics*, 2000, pp. 76–83.

[18] Badros, G. J.; Tirtowidjojo, J. J.; et al. A constraint extension to scalable vector graphics. In *Proceedings of the 10th international conference on World Wide Web*, ACM, 2001, pp. 489–498.

[19] Marriott, K.; Moulder, P.; et al. Solving disjunctive constraints for interactive graphical applications. In *International Conference on Principles and Practice of Constraint Programming*, Springer, 2001, pp. 361–376.

[20] Hurst, N.; Marriott, K.; et al. Dynamic approximation of complex graphical constraints by linear constraints. In *Proceedings of the 15th annual ACM symposium on User interface software and technology*, ACM, 2002, pp. 191–200.

[21] Pharo community. Pharo. [Online], 2008, [accessed 2019-05-16]. Available from: `http://pharo.org/`

[22] Cavarlé, G.; Syrel, A. Bloc. [Online], [accessed 2019-05-16]. Available from: `https://github.com/pharo-graphics/Bloc`

[23] Cavarlé, G.; Syrel, A. Brick. [Online], [accessed 2019-05-16]. Available from: `https://github.com/pharo-graphics/Brick`

[24] Brant, J. SmaCC. [Online], [accessed 2019-05-16]. Available from: `http://www.refactoryworkers.com/SmaCC.html`

[25] Moosetechnology.org. PetitParser. [Online], [accessed 2019-05-16]. Available from: `https://github.com/moosetechnology/PetitParser`

[26] Beck, K. SUnit. [Online], [accessed 2019-05-16]. Available from: `http://sunit.sourceforge.net/`

Appendix A

Acronyms

GUI	Graphical user interface
AST	Abstract syntax tree
DSL	Domain specific language
UML	Unified modeling language
API	Application programming interface

Appendix B

Contents of enclosed CD

```
readme.txt ........................ description of the contents of the CD
src .......................................... directory of source codes
├── thesis .................. directory of LaTeX source codes of the thesis
└── impl ................. directory of source codes of the implementation
text ................................... directory with text of the thesis
└── thesis.pdf ................................... thesis in PDF format
```

Appendix C

DSL Grammar

What follows is the input source for SmaCC, from which it generates the lexer, parser and AST. I omitted AST related code from the version presented here for brevity. Full version can be viewed by opening SmaCC's editor from Pharo's world menu and selecting `AodProgramParser`.

First, the tokens for the lexer. There are only few, as literal ones can be expressed directly in the expansion rules.

```
<integer>       :   0 | ([1-9] [0-9]*)  ;
<whitespace>    :   \s+                 ;
<identifier>    :   \w+                 ;
<comment>       :   \/\/ [^\r\n]*       ;
```

Next, SmaCC allows to specify priority and associativity of operators directly, so that it does not have to be encoded in the grammar's rules. Priorities are specified in ascending order.

```
%nonassoc "=" "<=" ">="                 ;
%left     "->"                          ;
%left     "+" "-"                       ;
%left     "*" "/"                       ;
%left     "."                           ;
```

Finally, the grammar's expansion rules. Character sequences wrapped in quotation marks is a token of the lexer. SmaCC also supports using operators like `*`, `+` and `?`, with their usual meanings, to define rules.

```
Program
  :  ( Constraint ";" (  Constraint ";" ) * ) ?
  ;

Constraint
  :  Value ":" Strength
  |  Value
  ;

Strength
  :  Identifier
  ;

Value
  :  Reference
  |  ActorExpression
  |  DotExpression
  |  Integer
  |  LinearExpression
  |  Point
  |  List
  |  Span
  |  "(" Value ")"
  ;

Point
  :  "{" Value "," Value "}"
  ;

Span
  :  Value "->" Value
  ;

List
  :  "[" ArgumentList "]"
  ;

ActorExpression
  :  BasicActorReference "(" ArgumentList ")"
  ;

ArgumentList
  :  ( Value ( "," Value ) * ) ?
  ;
```

```
Identifier
  :  <identifier>
  ;

Integer
  :  IntegerLiteral
  ;

IntegerLiteral
  :  <integer>
  ;

Reference
  :  ActorReference
  |  ShapeReference
  ;

ActorReference
  :  BasicActorReference
  ;

BasicActorReference
  :  Identifier
  ;

ShapeReference
  :  "@" Identifier
  |  "#" IntegerLiteral
  ;

DotExpression
  :  Value "." BasicActorReference
  ;

LinearExpression
  :  Value "+" Value
  |  Value "-" Value
  |  Value "*" Value
  |  Value "/" Value
  |  Value "=" Value
  |  Value "<=" Value
  |  Value ">=" Value
  ;
```