1 Introduction

The art of building design and construction has its origins lost in time. It is an ancient field of knowledge that has continuously reached new heights, allowing for always bigger and more extravagant designs. Albeit old, it is an always evolving field fueled by technological advances in many areas of science. A proof of that is the multiplicity of jobs that stem from building design and construction - not very long ago the architect was also the engineer, but with the increase in building’s complexity it is impossible for a single person to possess all the required knowledge.

Users of CAD are generally not very tech-savvy, so most interaction with the system is done through a ‘point and click’ GUI with, perhaps, tiny elements of programming. It is a shame that this is so, because it seems that with today’s technology in User Interfaces and Input/Output devices (typically a mouse, a keyboard, and a graphics tablet) the full computing power available cannot be fully explored. It is hard to create a model in an expressive way using only this kind of input, and so much information ends up lost or is not even entered in the first place.

Some CAD programs are so faithful to the ‘drawing board’ paradigm that several views of the same model (for example, a top and side views) are completely unrelated and share nothing between them. If we change the design of the building, we must change both views separately. This paradigm has started to change with the appearance of Building Information Modeling (BIM) software packages, where all the information is stored in a single model. When information about a specific part of the building needs to be used, it is retrieved from this central model. Regardless of the number of ‘views’ or applications using the model information they are always up to date, and as such there is no duplication.

It seems that next-generation tools can no longer rely solely on the traditional input devices (mouse, keyboard and tablet) but also need to be controlled through programming. If users want to adopt new technologies and use all the power at their fingertips, they will need to adapt themselves or at least have someone on their team who can program their CAD application for them.

Today’s CAD software does not address today’s problems adequately, but we propose the creation of a constraints language that will help solve them. Such a system would greatly simplify making changes, streamlining the creative process. It would allow different people to communicate in a single language, the constraints definition language, where it is easier not only to share drawings with others but also the intent behind them. Finally, it would greatly simplify the breaking down of larger problems (creating the final CAD drawing or model) into smaller problems, as that is not only enabled but also encouraged by a constraints definition language.

There is one architectural element that is widely used and which suffers from these problems in today’s software: the stair. Serving a very simple purpose - allowing people to go from one floor to the next - a stair can be as simple or as complex as the architect desires. Because of this most software on the market requires the user to introduce all stair measurements by hand instead of offering to do so automatically. It is rare for two stairs to be equal, so the architect must perform many calculations and check various regulations for each stair she wishes to place.

From this the usefulness of a system that could automatically generate stairs becomes obvious: even though the underlaying concepts are exactly the same the final result is very different. If those underlaying concepts were captured in some manner and shared across all stair designs we would only need to con-
cern ourselves with the details of the stair - how high should it be, how wide should the steps be, etc.

We hope to prove not only that this problem can solved by a constraints system a constraints system, but also that such a system allows a much higher degree of experimentation while producing only valid solutions.

2 Related Work

In this section we will start by analyzing some commercial CAD software and tools, with a focus on their programmability, constraints system (if any) and KBE capabilities (if any). We then move on to looking at some research work done in the CAD constraints area.

2.1 Autodesk AutoCAD

The first release of AutoCAD was in 1982, so it is a product with an immense history and a very large user base. So much so that its file format, DWG, is supported by most of the CAD products on the market.

Having started in an era where 2D was the norm - and when there was barely enough computational power for it - 3D was added as an after-thought. AutoCAD was thus used as a drafting program rather than a modeling program, a usage which continues up to this day. The user can have several different views of the drawing (top, sides, etc) which are not necessarily interconnected.

Dynamic blocks were added to AutoCAD 2006 to allow the creation of parametric building blocks. One of its main characteristics is that it is 2D only, and as such cannot be used in 3D models.

One of the strongest features of AutoCAD is perhaps its programmability. The addition of AutoLISP[?] to AutoCAD allowed third-party developers to extend the program. Everything from small helper routines to complete systems built on top of AutoCAD could now be done. It was later replaced by VisualLISP in 1997, and included a complete development environment (IDE, debugger, etc.) inside AutoCAD itself. In recent years, AutoLISP and VisualLISP were neglected by Autodesk in favor of VBA, .Net and ObjectARX programming. However, AutoLISP is still the most widely used of the three, as it is significantly easier to use than the alternatives for the average CAD user.

With AutoCAD 2010 a constraints system is included out of the box. Geometric and dimensional constraints can be created, allowing the user to specify the placement, orientation, or size (among others) of a shape or line. Examples include setting two lines to have the same length, to be collinear or perpendicular, or to ensure they intersect each other. These relationships between objects have to be explicitly created, one point at a time. While an Auto Constrain functionality exists it assumes that the objects on the workspace were implicitly constrained when they were placed. Extreme care is needed when using this functionality, and the automatically placed constraints should always be reviewed for accuracy.

While AutoCAD is evolving into a 3D modeling package this functionality was mostly designed and implemented for usage with 2D drawings. Since this is a fairly new feature - and it is not included in the cheaper version of AutoCAD, AutoCAD LT - it will probably take a few years for it to be picked up by users.

Despite all the shortcomings of AutoCAD, some of which are being actively addressed, it is a software package that will not soon be abandoned. The number of users it still has, combined with their reluctance to learn a new CAD paradigm and new CAD tools, gives AutoCAD a critical mass unrivaled by any other CAD package. More likely than not, it will remain a reference by which all CAD software is measured for years to come.

2.2 Autodesk Revit

Revit is a software package released in 2000 by Revit Technology Corporation (bought in 2002 by Autodesk) which embodies the new CAD paradigm, Building Information Modeling (BIM). It was built from the ground up as a BIM software package which fully supports 3D modeling, so the concept of AutoCAD’s views does not exist in Revit. There is only one model of which there can be many views, and a change in any view is in reality a change to the model itself. Moving a wall in one view moves the wall automatically in all the project’s views.

Revit has a built-in constraints system equivalent to that of IDX Variable Constraint System. Con-
straints have to be defined manually and in a exhaustive fashion, and the user is responsible for manually changing any free values. Users interact through Revit’s graphical user interface, selecting objects and setting their constraints. This system exists mostly as a way of ensuring the correctness of the design and of helping the user not make mistakes.

It is possible to create a master object in Revit where certain constraints are defined (acceptable values or intervals for a property, etc), and when instances of that object are created those constraints are enforced. It is possible to create new object families from existing families, overriding some of the properties and constraints or creating new ones. This concept is akin to that of inheritance in object-oriented programming languages.

Despite these advances over AutoCAD, the constraints system is relatively simple and does not allow for more elaborate results. It is not possible to have Revit present a solution that satisfies a given set of constraints, nor is there a language through which these constraints can be declared.

Revit provides an Application Programming Interface for plugin development based on Microsoft .Net. There is no equivalent of VisualLISP, so programming can’t be done inside of Revit itself. This most likely places the power of programming outside the hands of even the Revit power users, and is used only by third-party developers who wish to sell Revit plugins.

Revit is slowly beginning to take AutoCAD’s place, though it is not expected to fully replace it for years to come.

2.3 Customizing Mass Housing: A Discursive Grammar for Siza’s Malagueira Houses

‘Customizing Mass Housing: A Discursive Grammar for Siza’s Malagueira Houses’[?] is the PhD Thesis dissertation of Prof. José Pinto Duarte. His goal was to create a system that automatically creates designs for mass-produced houses with high quality, satisfying some architectural style and meeting a certain maximum cost. This overcomes the problem of having the same house design repeated over and over again, as the cost of producing new designs by computer is very low. While this is a very over-simplistic view, it is sufficient for the purposes of our work.

Using shape grammars the system starts with a clean design and creates a list of all the possible successors after the application of a simple rule (functional division of the current free space, inclusion of a stair, etc.). This process continues until there is no available space left, and nothing more needs to be added. This was not enough, as the solutions needed to be not only syntactically correct but also semantically correct. Discourse has a similar problem - for humans to communicate they must be both syntactically correct - so the message can be understood by all parties involved - and semantically correct - so it also makes sense in the context of the conversation.

To address this problem the author proposes a Discursive Grammar for Siza’s Malagueira Houses. At first Programming Grammar is used to ensure that the solutions are syntactically correct - that they follow laws and regulations, for example. After that a Design Grammar is created according to some architect’s style - in this case Siza Vieira - and which ensures that only homes that architect would create are presented as solutions. The author also argues that both grammars must be in accordance with each other to ensure that solutions can be created. It does not make sense to have conflicting rules in both grammars, so some care is needed in their construction.

Some valuable lessons can be learned from this. The first is that it is useful to separate law and regulation from design constraints. The former apply to every conceivable object created by a constraints system, while the latter do not as they vary from user to user. Not only that but the user should not be allowed to change laws and regulations embedded into the system by mistake. This separation ensures that if they are changed it is because the user expressly wants to, and is well aware of what she is doing. The second lesson is that it is important to ensure that the design constraints follow the laws and regulations. It does not make sense to specify, for example, that a wall of a room must have a length of at most 1 meter if the law determines that the minimum length must be 1.5 meters.

One of the problems the author faced, and which is of interest to us, is that of the gigantic size of the search space. Even the lower bound means an search space in the count of thousands of millions of alter-
natives, which is naturally an impractical situation. This is partly solved by rejecting some of the designs the shape grammar generates. Designs which do not meet a particular requirement are rejected at each step. If symmetric patterns are reached, one of them is eliminated as its inclusion adds no real value. Besides reducing the search space, this strategy guarantees we end up only with valid solutions, and do not waste time pursuing solutions which would never be accepted.

The rules themselves also have some kind of knowledge about the requirements, and enforce them when generating the successors. An example of that is the stair placement rule. When placing a stair in a room, the rule adjusts the area of the room and the placement of the stairwell rectangle so that the ceiling’s pressed concrete beams can run parallel to the stair without interruptions.

Despite these concerns stair creation is not a big focus of this system - only stair placement. The stairs have most of their properties fixed, and might therefore not be suitable for use in other types of buildings. In both the original Shape Grammar and in the final Discursive Grammar stairs always have fourteen steps, and the tread length is fixed at 0.25 meters. Only the riser height changes depending on the height of the floor.

While this work has different goals from ours, there are some shared problems and objectives. Both systems aim to create and validate drawings based on some rules given to the system, and both deal with an immense search space when searching for possible solutions. An automatic stair creation system could be used to enhance this Discursive Grammar by making it more easily applicable to other design styles.

2.4 Design Exploration through Bidirectional Modeling of Constraints

In his PhD Thesis Axel Kilian[?] argues that constraints can be design drivers in design exploration, especially if a generative approach to design is used. But today’s tools are not enough, as they require that the design be at least almost finalized, even if only in an informal fashion, before it can be represented in software. Hierarchical relationships between different parts of the design are pretty much fixed and extremely hard to change.

There are even more limiting factors, such as the way the Graphical User Interface (GUI) is organized or what is made available - and how - through a Application Programming Interface (API). Users interact with the modeling software through its GUI or API, and it is through their usage that users create their mental model of the software and its functionality. It is so much so that the design processes follow the software and it should be the other way around. Software should be created to enable a user’s vision, instead of looking at what is available on the market before deciding how ambitious the project should be.

Constraints are a way of achieving that, and might actually allow the search for new and innovative types of solutions. This is possible because users can experiment at will knowing that when a solution emerges, it follows all the specifications - no manual checking must be performed.

Constraints can also be design drivers in that a new and innovative solution might be the only one that satisfies a particular constraint. For this to be true design drivers constraints must be bidirectional, they might both affect and be affected by any other constraint.

Bidirectionality is also required at other levels. When objects are created through the use of geometrical forms there is a transformation from its ideal representation to a physical representation, and that process is neither lossless nor bidirectional. We can go from the abstract to the concrete representation, but not the other way around. This problem can be solved through the use of programming. Instead of creating the model directly we can program it, never losing information.

Unfortunately programming alone is not enough, as if done poorly it is no better than manually creating everything - only faster. Since designers are accustomed to seeing computer modeling as a process where you extensively create different and isolate objects that together just happen to represent the design you wanted, it is natural that they should follow the same approach when programming.

The conclusions reached support our view that to allow ever bigger and more complex programs users need not only learn how to program, but they need to learn how to program a constraints system. Constraints systems are not impeditive of innovation but on the contrary can be innovation drivers.
We hope to build a system through which the suggestions put forward in this work can be applied in practice.

3 Stairs as a practical example

A constraint system can be useful in many different domains, so we must choose a practical application upon which to build our system. The building of stairs immediately came to mind as an apparently simple problem, but which in reality could greatly benefit from a constraints system.

Stairs are such an integral part of our everyday life that we don’t even give them a moment’s thought. Whether there is the need to climb them or descend them, it is done as naturally as walking in a plain terrain. Some particular stairs might present additional difficulties, such as small steps or a very steep slope, but as soon as the destination is reached the whole ordeal is readily forgotten.

But is it as easy for those who have to create these stairs? What rules are enforced to make the experience so seamless for the average person? What mental process is followed to ensure that a “good” stair is built, and not something which at best is difficult to walk and at worst can lead to accidents with serious injuries?

If we could understand this we could write software to discover solutions automatically. Such software is rare and when it exists it is, as we saw, incomplete. There are so many laws, regulations and conventions that must be met that it can become nearly impossible to follow them all. Not only that, but stair designs can not always be reused. A building designed for one country most likely cannot be built in another without some adaptations required by laws and regulations. Freeing the architects and engineers from the burden of having to know and verify the enforcement of these regulations would be a great step forward.

While some tools do exist that assist architects and engineers in adding stairs to their blueprints, the bulk of the work is still manual. Users must specify all the measurements manually (tread height, riser length, type of stair, etc.) instead of leaving those decisions to the software. This means that all calculations must be made by the user beforehand, and that there is no assistance in checking whether the result complies with applicable laws and regulations. The value of such software comes from the fact that it usually has a series of stair designs among which the user can change easily. An automatic stair creation system would mean cheaper, faster and more correct designs.

As we will see in the following sections, if we wish to write software to create stairs we must first break them down into their core components and understand them well. We must then create some abstract representation with which the computer can work to arrive at results. After that, the stair must be created in some architecture software package.

3.1 Defining a stair mathematically

If we are trying to build a stair we must begin by determining which exactly are its core components. The simplest possible stair has single flight straight with no landings. All steps are equal, which gives us several ways of looking at the stair - for example, it can be seen as a single step repeated numerous times or as several different steps that happen to have the same measurements. Going one step further we can completely eliminate the steps and consider the stair to be formed by several consecutive and interconnected lines. These are just three examples of looking at the problem, but there are many more. Each has its pros and cons and no single one emerges as a clear choice.

The thing to notice is that although the end result is exactly the same the process followed to create the stair in each case is immensely different. A good abstraction is the cornerstone of a good constraints system. Following one path might lead to quick results, while following another might make it impossible to reach a solution. What we are interested in is finding a good compromise between convenience for users and the feasibility of implementing that abstraction as a computer program.

In theory an abstraction should be malleable, modelable, perceptible, simple and abstract enough to encompass most, if not all, possible stair designs.

A seemingly good abstraction fulfilling these requirements is seeing the stair as a set of connected points. It is obviously malleable, modelable, perceptible, simple, and choosing an appropriate granularity it can certainly encompass all possible stair designs.
Unfortunately in practice it does not work. While the idea itself is simple, its translation into a constraints system is extremely complex. The system grows so big that the computational power to solve it is too big - and that is if there are computational methods to solve it at all. Only the smallest of objects could be constrained in such a way, and even then it should only be done if there are no other alternatives.

What we need is a general method that works regardless of the problem at hand. We might need to constrain big and small objects alike, and having different approaches for each should be a last resort.

We have a rather limited number of properties in a stair regardless of how big it actually is. As all the steps are equal we do not need to constrain each one individually when trying to find a suitable stair. This is true not only of steps, but also of landings - although landings differ from the regular steps, all landings of one stair are usually equal to each other.

Applying this reasoning to the entire stair we came up with the basic properties which fully define a stair: riser height, tread width, landing width, number of steps between landings, number of landings, number of flights, and the width of the landing between flights.

These findings suggest that we can constrain an object's properties instead of constraining the entire object explicitly. In the case of the stairs this means that instead of constraining each step individually we constrain only the list properties. The fact that each step does not have to be calculated individually, and aggregated calculations are performed instead, greatly reduces the equation system.

For example, if we constrained each step we would have to specify that the width of each step's tread was equal to that of its successor. This had to be done for each step individually with the exception of the last one, resulting in \( n-1 \) different equations. Now all that is said is that the sum of all treads has to be equal to the total width, and that all the tread widths are equal. In other words, we went from \( n-1 \) equations to one without any loss of information. This greatly reduces the computational load as there is a much smaller number of constraints to satisfy which, in turn, allows us to constrain much bigger objects. We gain both in terms of scalability and response time.

But there are shortcomings, the most immediate of which is that more exotic or artistic stair designs cannot be produced with such a system. As we are only concerned with the functional aspects of the stair this is not a problem for us. Not only that, but constraining an entire object at once creates even bigger problems. It is bound to produce errors, especially if the object is very large, and greatly reduces the ability to experiment as the system becomes too complex very early on. Because of this constraining only a few key properties and not the entire object is a valid, even desirable, approach for our goals.

### 3.1.1 Straight flights

The simplest possible stair is a straight flight - a stair which might have some landings, but whose direction never changes. Because they are so simple and easy to understand they serve as an excellent starting point.

For a simple stair with one flight we need constrain only three properties - riser height, tread width and landing width. More specifically, we must ensure that the sum of all treads and landings equals the total stair width, and that the sum of all risers equals the total stair height. Little else is need to be able to create a basic stair.

As the stairs become more complex and several flights and landings are introduced the constraints system grows, but the underlying principles remain unchanged.

### 3.1.2 High stairs

Unfortunately stairs are rarely as simple as the one just described.

Most are comprised of several flights (an uninterrupted series of steps between landings) and straight flights (a series of flights going in the same direction until an intermediate landing is reached before the stair changes directions). Our stairs can currently have several flights, but only one straight flight.

This means that high stairs cannot be created, as in these usually there is not enough room for a single straight flight. It is often necessary to create stairs composed of various connected straight flights in a U pattern. Two problems arise from this: how do we detect that we need a U-shaped stairway, and what do we need to change in our system to produce this type of stairway?
The first problem is solved by the heuristics we introduced, which will be discussed in more detail later. Briefly, what happens is that some simple calculations are performed and we see if we have enough space for a stair with a given number of straight flights. If we do not, we try to figure out if it is because the height to cover is too big. If this is the case, we place two more straight flights and start the process again.

As we had a simple and working system we wanted to make the least amount of changes possible, and did not want to include any extra complexity in the system. Following a close study of these stairs we were convinced that few changes were required. A high stair can be seen as the union of the straight flights which our system already produced, so expanding on what we already had was sufficient. Furthermore, each of the united straight flights is equal to all the others. This means that instead of having to perform calculations for the entire height of the stair we need only create the first straight flight, and then duplicate it as many times as we need.

This simplification allows our system to continue producing results for the vast majority of its uses, and nothing more is required to be able to produce both long stairs and high stairs. Our system of equations is able to accommodate them all, and the adequate solution emerges automatically without any user intervention or previous indication.

### 3.1.3 Spiral stairs

Unfortunately sometimes there is not enough room for a straight stair, but a solution is needed nonetheless. In those cases, it is not unusual to build spiral stairs. Revolving around a central axis, they occupy less space while still being able to cover adequate vertical distances.

Before finding the adequate inequations we need to first discover what the free variables should be. The height, width and depth of the steps are obviously dependent on the space available. We also need to know the number of steps per revolution of the stair, the number of revolutions until we reach the desired height, and the angle between two steps. Given these variables, we actually do not need many constraints.

The stair must fit in the available width and depth. We specify both separately, as the available step does not necessarily have to be a square. Since the stair is symmetrical, we need only perform these calculations for half the stair.

For simplicity we model steps as simple boxes instead of the triangular prism-like shape they would in reality take. Even so, all the steps must have their inner edges connected. For this to happen, the distance to the central axis must be aligned with the edge width. A simple trigonometric equation enforces the usage of the correct measurements, meaning that all the steps are aligned.

The placement of each step is at a fixed distance from the center, with only the rotation degree changing. This makes polar coordinates a better alternative than Cartesian coordinates. The code becomes much simpler, and the amount of calculations that need to be performed is greatly reduced. Using polar coordinates also has another added benefit: it becomes trivial to specify the rotation of the steps as a function of the number of steps per revolution. A full rotation is represented by an angle of $2\pi$. If we have $n$ steps, we need only divide $2\pi$ by $n$ to determine the angle of the rotation of each individual step.

The other big constraint remaining is establishing that the stair height is equal to the sum of the height of all the steps.

A detailed analysis of spiral stairs reveals that we do not need as many variables as we had initially listed, as some of them can be obtained implicitly from the values of other variables. For example, knowing the step’s width and the number of steps for revolution we can obtain the step rotation angle.

In the case of a simple spiral stair such as ours, with only one revolution and no landings, only four variables are needed. These are riser height, tread width, step depth, and the number of steps per revolution.

Since the distance to the central axis must be calculated and used when creating the stair, in practice it might be more convenient to include it in the stair tuple. In our implementation we do precisely this, and add two extra bits of information after the tuple is created: the angle between steps and the total height of the stairs. Since this information is needed by the spiral stair creation procedure, it is calculated beforehand for convenience.

If after the constraints system is solved we have a
solution, the stair is created. If not, our system currently does not support any other stair type and, as such, no stair can be created in the available space.

3.2 The implementation

So far we have shown that it is possible to completely define a stair mathematically. We need only break down the stair into its core components and figure out the configuration those core components should have.

But how can we find solutions to the constraints? And after having a solution how can we make the stair appear in the user’s software of choice?

We will address these matters in the following sections.

3.2.1 Finding a solution

Having input the inequation system into the system we need to find its solution. This task is performed by the Mathematica software package.

The inequation system is transformed into the format that Mathematica understands, and a solution is requested.

After Mathematica tries to solve the inequation system, there are three possible results. Either a solution is found, or no solution exists, or no solution could be found but it cannot be proven that no solution exists.

In the first two cases we have a clear and expected result - either a stair can be created under those constraints, or it cannot. If it can, it is passed on to the function responsible for creating the actual stair based on the solution tuple. If it cannot, we try to place a spiral stair (described in more detail in 3.1.3) as that type of stairs usually requires less space and, as such, might be a viable alternative. If spiral stairs cannot be used either then our system is not able to propose other alternatives. The constraints must be alleviated, or the space reserved for the stair changed.

The third case, when a solution cannot be found nor proved to be inexistent, is a bit more subtle. We cannot know if a solution exists or not, as Mathematica does not have the tools required to answer us. There are many possible causes for this. The inequation system might be too big or too complex, the number of free variables might be too great, or there might not be enough computational resources available for Mathematica to use. There is nothing we can presently do to alleviate this problem except trying to make the inequation system smaller.

If no solution is returned the system stops its execution at this point. Otherwise the stair is created following the algorithm described in the next section.

3.2.2 Creation of the actual stair

So far we have only dealing with an abstract stair that had no real existence. We have the information required to create the stair, but have not yet done so. That creation only happens at the final step of the stair creation algorithm.

In the cases where a solution tuple was found it is passed on to a function that knows how to interpret it and create the stair, placing the steps and landings on their correct locations. How this is done depends wholly on the user’s desires, as this function has a deep connection to the stair restrictions which are in place.

The solution to the constraints system must have a full semantic match to the stair generating function. Each tuple’s element has a very precise meaning, and a wrong interpretation would almost certainly lead to an incorrect stair. For example, if a riser height in the solution tuple is interpreted as a landing width in the stair creation function, the resulting stair will have completely wrong measurements.

There might also be other things to have in consideration. For example, a 2D stair might be drawn differently from a 3D stair, and different software packages might require different approaches to the drawing or modeling of the stair.

We have implemented two stair creation procedures. The first creates a 2D stair which is plotted by Mathematica, and the second creates a 3D stair in AutoCAD. Both produce only straight stairs with equal tread widths and riser heights for all steps.

Both create steps in an iterative fashion. Knowing the basic measurements for all the steps they add one step after another until the desired number of steps is reached. If there are landings the system knows how many steps there should be between landings, thus knowing when to place a landing or a step. The system also knows how many flights each straight flight has, so it can keep track of where it is in the stair
creation and change directions accordingly.

4 Work validation

No system is useful if it is not correct, and in our case it is even more so. Deviations from the specification, even if small, can lead to construction problems or even loss of human lives. As such it is paramount that such deviations never take place.

Not only that different locations have different regulations, no two designs - even if from the same architect - are equal, and different architects have different styles. Because of this it is important for our system to be able to easily accommodate different regulations and design styles.

The Portuguese building regulations, Regulamento Geral de Edificações Urbanas[?], specify that the minimum tread width is 25cm and the maximum riser height is 19.3cm meters. Inputing those as regulation constraints any design constraints we add must respect those measurements. If we add as a design constraint that a riser should be higher than 20cm or that the tread width should be shorter than 15cm we get, as expected, no solution.

But constraints can be subtler. Blondel, for example, established[?] that the sum of the tread width with twice the riser’s height should be equal to a stride - which he calculated to be about 64.77cm.

Experimentation showed that results fully followed these constraints, producing only correct results. But being correct is not enough. Given the nature of our system it must also be flexible - both because users want to experiment and because different projects have different requirements. It thus becomes necessary to check how easy it is to introduce changes to the system.

Adding Blondel’s rule to the design was one way of customizing the design, albeit a rather simple one. We want our system to be as flexible as possible, so we will now create an exotic stair to show what our system can do.

The stairs we have created so far all had steps with the same tread length, but a user might want, for example, a stair where each step has a tread 10% bigger than its predecessor. With a small amount of work our system also allows for such designs.

The first thing to notice is that the tread width is now specified by a geometric progression and that the total width of the stair is the sum of that progression.

We need only change two parts of our system: the equation that calculates the stair width and the stair creation procedure. Altering the equation is trivial, and modifying the stair creation procedure is easy too.

Instead of creating steps with a uniform tread width we keep track of the current step’s tread, and increment it by 10% with each step we create. Landings are of course placed normally, as they are not affected by our calculations.

If heuristics are used they must be updated too, if the user wishes to keep them. As was mentioned in Section 3.2.2 there must be a full semantic match between the heuristics and the stairs the system is prepared to create. As such when the type of stairs that are created changes, the heuristics must be changed accordingly.

These two examples show that the system is correct, creating only valid stairs, and that it can be changed with ease. We had already introduced a completely new stair type, spiral stairs, in Section 3.1.3 but have now shown that existing stair types can be altered with ease as well. All that is required is a mathematical model of the desired type of stair.

5 Conclusion

As we saw Computer Aided Design is an integral part of the workflow of today’s architects and engineers. Unfortunately many of today’s CAD vendors still follow the drawing board paradigm too closely and have only recently started to move to completely different ways of working. We believe the next step is a constraints system that can automatically propose solutions to the user.

The most obvious approach to this, constraining each and every point of the object at hand, proved to be ineffective for all but the smallest of objects. For all others the constraint system grows so big that the computational resources available are not enough to solve it. This poses a problem, as big and complex objects are precisely the ones that benefit the most from constraining - all the others can be created directly must faster.

A good solution comes from the fact that most ob-
jects, if not all, can be recreated without loss of detail by knowing just a few key details. Constraining only those core properties makes the constraints system much smaller and easier to understand - we only need a way of interpreting those properties and reconstructing the desired object from them.

This is important, as it can define the success of the system. There must be a full semantic match between the constrained properties and their interpretation when the object is constructed. Failure to do so might lead to catastrophic results - one such example would be interpreting a circle’s radius as the length of the side of a square. This imposes a small maintenance overhead, but the gains in user friendliness and processing time more than make up for that nuisance. Systems can be solved must faster, and it is easier for humans to understand and introduce changes to the system.

We showed an example of that when we altered our stair creation system to create stairs whose step’s treads follow a geometric progression. A few inequations and the stair creation procedure had to be tuned, but those were simple changes with no implications for any other part of the system.

Producing several output functions makes it easy to produce any type of output from any constraints system with ease, such that when changes are made all the outputs can be reprocessed again automatically and without further human intervention.

While it is true that not every conceivable design can be captured by our system enabling that was not our main objective. We were mostly focused on the functional aspect of the stair by choosing the adequate number of steps and landings as well as their measurements. This is enough for architects and engineers to take it from there and insert all the desired artistic elements. It is even conceivable that another constraint system could be placed on top of ours, adding those artistic elements itself.

Our ultimate goal is to enable architects and engineers to think in terms of building blocks, mixing and matching them as desired to create a building. Our own system is constructed of four different building blocks for greater flexibility:

- The inequation system representing the constraints
- The mathematical software package that solves the inequation system
- The functions that translates the solution to the inequation system into a stair
- The modeling software package where the stair is actually created

All these modules are well separated and can be changed independently of all others. This gives users the greatest flexibility possible in terms of which software packages to use, and makes it easier to introduce changes to the system. Such an architecture allows us to ensure correct results and introduce changes to the system much more easily.

We believe that we have proven our initial thesis to be correct - that a constraints system is a good choice for enabling both experimentation and code reuse, and that if a certain care is taken it produces only valid solutions.

We hope that our contribution, albeit small, serves as an eye-opener. The lack of constraints systems in the market might make it seem like users do not want to learn how to program, but the past has shown us that with a simple enough language and a powerful system behind it many users are willing to put the effort into mastering CAD programming.