



DESIGNA - A Shape Grammar Interpreter

Rodrigo Coutinho Correia

Thesis to obtain the Master of Science Degree in
Information Systems and Computer Engineering

Examination Committee

Chairperson: Prof. Dr. Ernesto José Marques Morgado
Supervisor: Prof. Dr. António Paulo Teles de Menezes Correia Leitão
Supervisor: Prof. Dr. José Manuel Pinto Duarte
Members of the Committee: Prof. Dr. Joaquim Armando Pires Jorge

June 2013

Acknowledgments

I would like to express my sincere thanks to my adviser Prof. Dr. António Leitão. For his patience over the years, for all his time, words, work and ideas. For his support and advices. For the journeys in Racket and in Lisp both on Gulbenkian and on computers, new & old. And above all, his sincerity and friendship. All this and more, helped me accomplish this journey.

To Prof. Dr. José Pinto Duarte who introduced me to the shape grammar concept. Also for his patience, serenity, work and conferences, advice and especially new horizons and trips. Also for his place of work while he was out.

And thanks to my family, M, B, Dk, G & D and H, and all that were not mentioned but present.

Lisbon, June 5, 2013

Rodrigo Coutinho Correia

Resumo

Embora a personalização de produtos já seja possível, a personalização em massa da habitação ainda é difícil. Actualmente, quando um Arquitecto desenha um projecto de grandes dimensões, a solução que adopta é desenhar um número limitado de casas, que repete ao longo do projecto. Isto acontece porque, (1) o Arquitecto não consegue desenhar cada casa individualmente respeitando um estilo comum, devido à dimensão do empreendimento, e (2) as técnicas tradicionais de manufactura necessitam de repetição para baixar os custos. De modo a resolver este problema, as gramáticas discursivas foram propostas. Estas combinam uma gramática descritiva e uma gramática da forma (GF) com um conjunto de heurísticas e permitem gerar desenhos sintaticamente e semanticamente correctos. Infelizmente, os interpretadores de GF mais usados apenas trabalham com formas bidimensionais e oferecem pouco ou nenhum suporte para controlar a aplicação das regras. Por este motivo, as abordagens actuais para implementar interpretadores de GF restringem aquilo que pode ser representado e na maioria geram desenhos sem significado. De modo a ultrapassar este problema, esta tese propõe uma arquitectura de software para um interpretador de GF de modo a suportar a maneira de pensar e trabalhar de um Arquitecto, actuando como uma ponte entre o formalismo das GF e as aplicações de CAD. Para validar a proposta, um interpretador de GF, chamado DESIGNA foi implementado, permitindo: (i) criar e manipular formas com rótulos, (ii) representar regras da GF, (iii) controlar a aplicação de regras e (iv) apresentar o desenho gerado nas aplicações de CAD mais usadas.

Abstract

Although customization of products is already possible, mass customization of housing is still difficult. Currently, when a designer conceives a project for a large development, the solution is to design a limited number of house types and then repeat. This happens because, (1) due to the size of the development, the designer is not capable of designing each house in a common style individually, and (2) traditional manufacturing techniques require repetition to lower the costs. To solve this problem, discursive grammars were proposed. Using a combination of a description grammar and a shape grammar (SG), plus a set of heuristics, the discursive grammar is able to generate designs that are both syntactically and semantically correct. Unfortunately, the most used SG interpreters only work with shapes defined in a two dimensional plane, and provide little or no control over the application of rules. Therefore, current approaches to implement a SG interpreter restrict the range of designs that can be expressed and, often, produce meaningless designs. To overcome this problem, this thesis proposes a new software architecture for a SG interpreter that supports the designer's way of thinking and working with shape grammars, by acting as a bridge between shape grammars and a CAD application. To validate the proposal, the SG interpreter DESIGNA is implemented, allowing: (i) the creation and manipulation of labeled shapes, (ii) representation of shape grammar rules, (iii) control of the rule application to shapes, and (iv) presentation of the computed design in the most used CAD tools.

Palavras Chave Keywords

Palavras Chave

Sistema Gerativos

Gramáticas da Forma

Gramática Discursiva

Malagueira

Racket

Keywords

Generative Systems

Shape Grammars

Discursive Grammar

Malagueira

Racket

Contents

1	Introduction	1
1.1	Motivation	1
1.2	Thesis Statement	2
1.3	Contributions	2
1.3.1	Publications	3
1.4	Outline	3
2	Related Work	4
2.1	Generative Systems	4
2.2	Shape Grammars	5
2.2.1	Mathematical Definition	6
2.2.2	Parametric Shape Grammars	7
2.2.3	Emergence	7
2.2.4	Analytical and Original Grammars	9
2.2.5	History	10
2.2.6	Examples	10
2.2.7	Full Example	11
2.3	Shape Grammar Interpreters	11
2.3.1	Visual, Symbolic and Set Grammars	16
2.4	Discursive Grammar	17
2.4.1	Prologue	17
2.4.2	Two Viewpoints	17
2.4.3	Mathematical Definition	18
2.4.4	System Architecture	18

3	Overview	20
3.1	Shapes	21
3.2	Rules	21
3.3	Output	23
3.4	Software Architecture	24
4	Designa	26
4.1	Shapes	26
4.1.1	Shape Representation	26
4.1.2	Shape Generation	30
4.1.3	Labels	32
4.2	Rules	33
4.2.1	Rules Description	34
4.2.2	Rule Application and Control	35
4.3	Output	38
4.4	Designa	38
5	Evaluation	41
5.1	Ice-Ray Shape Grammar	41
5.2	Three-Dimensional Shape Grammar	46
5.3	Malagueira Shape Grammar	49
5.3.1	Prologue	49
5.3.2	Shapes	54
5.3.3	Rules	55
5.3.4	Output	57
5.4	Discussion	57
6	Conclusions	63
6.1	Future Work	64

List of Figures

2.1	A simple shape grammar inscribing squares in squares	6
2.2	Generation of a shape using the SG of figure 2.1	7
2.3	Some shapes in the language defined by the SG of figure 2.1	7
2.4	Simple parametric shape grammar, inscribing quadrilaterals into quadrilaterals	8
2.5	Generation of a shape using the SG of figure 2.4	8
2.6	Some shapes in the language defined by the SG of figure 2.4	8
2.7	Example of a shape grammar	9
2.8	Rule application of shape grammar of figure 2.7	9
2.9	Full example of a shape grammar	12
2.10	Screenshots of shape grammar interpreters	15
2.11	Screenshot of CityEngine	16
2.12	Discursive grammar viewpoints	18
2.13	Discursive grammar system architecture	19
2.14	PAHPA-Malagueira discursive grammar	19
3.1	Example of a tetrahedron shape and its boundary elements	22
3.2	Tetrahedron shown as a graph	22
3.3	Proposed shape grammar osftware architecture	25
4.1	Graph representation of a tetrahedron	28
4.2	Edge representation and edge relations in the winged-edge and halfedge data structures .	29
4.3	Euler operators, <i>split_facet</i> and <i>join_facet</i>	31
4.4	Two other examples of Euler operators	31
4.5	Example of a unit cube creation using Euler operators	32
4.6	Example of a shape grammar rule - rectangle division	35
4.7	Example of an operator - rectangle division	35

4.8	Example of rules/operators - rectangle divisions	36
4.9	Search trees, depth first and breadth first	37
4.10	General overview of the system architecture proposed	39
4.11	DESIGNA showing shape grammar rules and corresponding output	40
5.1	Two example of Chinese ice-ray designs used in ornamental windows	41
5.2	Stiny's original ice-ray shape grammar rules	42
5.3	Result obtained from the implemented ice-ray rule using as the initial shape the shapes represented in the first column	43
5.4	Example of a facet (in gray) that is composed by collinear edges	44
5.5	Derivation example, showed in AutoCAD, using the implemented Ice-Ray shape grammar	44
5.6	Possible designs, showed in AutoCAD	45
5.7	The initial shape used for the results shown in table 5.1, and some designs generated when using the breadth first search without the area restriction	47
5.8	Three-dimensional shape grammar rule - transform a facet into a pyramid	48
5.9	Results of three-dimensional shape grammar example using different search strategies . .	48
5.10	Design solutions from the 3D shape grammar - a tetrahedron as the intial shape	50
5.11	Design solutions from the 3D shape grammar - a cube as the intial shape	51
5.12	Malagueira urban contexts of houses	52
5.13	Simplified Malagueira shape grammar rules and derivation	53
5.14	Example of application of the abstract operators defined	55
5.15	Evolution of design solution using Malagueira grammar	58
5.16	One final design solution of a house with a back yard	58
5.17	Sixteen possible solutions for a backyard house	59
5.18	Malagueira shape grammar rule number 5	62

List of Tables

2.1	Shape grammar implementations	13
5.1	Results of depth and breadth first search for the ice-ray grammar	46

List of Acronyms

2D	Two-Dimensional
3D	Three-Dimensional
CAD	Computer-Aided Design
CGAL	Computational Geometry Algorithms Library
COM	Component Object Model
SA	Software Architecture
SG	Shape Grammar
SGs	Shape Grammars

1 Introduction

1.1 *Motivation*

Mass customization of products is already possible. Examples include the trademarks of Nike and Dell that provide customization for their products. Both provide online stores where the client can personalize the product(s) to his/her needs, desires or tastes. In the Nike store, the client can choose different materials or colours for shoes and other sport goods. In the case of the Dell store, computers can be configured with custom internal specifications, memories, cpu and general aesthetics, ranging from different cases for workstations, to personalized covers for laptops.

In the Architectural field, although there were always the need for personalized features to a given design, these were limited to different materials and colours. However, nowadays, there is a growing and more demanding need to provide even more personalized features. Its area of application ranges from building systems used in construction, personalized designs of a housing, custom furniture, as well as colours and materials used. Unfortunately, it is still very difficult to provide personalized designs. As an example, consider a designer that is faced with a large development. Depending on the project size, it might be impossible to produce a different design for each house while respecting a common style, a limited budget, and tight deadlines.

To overcome these constraints, new processes have been developed. These processes extended computer-aided design (CAD), and computer-aided manufacturing (CAM) with automatic generation of designs, providing mass-produced cost-effective houses with quality usually associated with individually designed houses. This ability to provide many personalized design solutions, even those that were not initially thought by the designer, but still valid for the intended style, is called mass customization of housing, and is the design problem that we will address in this thesis.

To solve this design problem, Duarte (Duarte, 2005b) proposes a system for the generation of designs based on a mathematical model called discursive grammar (DG). A DG can be viewed as a composition of a description grammar, a shape grammar (SG) and a set of heuristics. The description grammar generates a design brief (DB) that describes the design as a set of user requirements and site data. The SG, a generative system based in the application of rules, generates designs in an architectural style, accordingly to a given DB. The set of heuristics is used to guide the generation of design solutions to a desirable design.

While the use of description grammars is already well understood (Duarte & Correia, 2006), most SG interpreters work only with shapes defined in a two dimensional plane and provide little or no support for controlling rule application. Therefore, current approaches to implement a SG interpreter restrict the range of designs that can be expressed, can produce an infinite number of solutions and, often, can produce meaningless designs.

1.2 Thesis Statement

Shape grammars (SGs) can be used to describe existing designs and/or to express new designs, and are considered by many Architects as the best formalism for capturing a design process. Unfortunately, current SG interpreters exhibit several problems, ranging from geometric limitations to the control of rule application.

To overcome some of the problems that affects current SG interpreters, this thesis proposes a new software architecture for a SG interpreter that combines: (1) a rigorous 3D modeler augmented with a labeling mechanism, (2) the description of SG rules as transition operators, (3) search algorithms that rely on transition operators to express the order in which rules are applied, and (4) a fully capable CAD application for further development of the generated designs. This combination allows the implementation of a SG interpreter that: (i) represents, creates and manipulates labeled shapes, (ii) represents SG rules, (iii) applies rules to shapes and controls rule application, and (iv) presents the computed design in AutoCAD or Rhinoceros3D.

1.3 Contributions

The main contribution of this thesis is a SG interpreter that uses the proposed software architecture. The resulting prototype not only allows a generic and flexible way for the definition and generation of labeled shapes, but also provides both manual and automatic modes to control the order of rule application.

A second significant contribution of this thesis is integration between a SG interpreter, a programming environment, and a set of CAD applications. This is a novelty, in the sense that, previous systems which implemented SG interpreters use custom viewers to visualize designs and rely in file exchange to further processing of the design in a CAD application. Consequently, this thesis, also promotes the portability of SG across different CAD applications.

A third contribution is the use of a kernel for exact geometry, where numeric computations can be tuned for performance or precision. The usage of such kernel avoids numeric rounding errors and allows exact queries to be performed on the geometry, e.g. querying if a point lies on a line or not.

A less important but still relevant contribution is the implementation of a computational geometry package for the Racket programming language, allowing the Racket community to easily access a large set of algorithms and data structures designed for the solution of geometric problems.

1.3.1 Publications

During the development of this master thesis two publications were submitted and accepted: (1) *MALAG: a discursive grammar interpreter for the online generation of mass customized housing* (Correia, Duarte, & Leitão, 2010), presented at the workshop on "Shape Grammar Implementation: From Theory to Useable Software", and (2) *DESIGNA: A General 3D Shape Grammar Interpreter Targeting the Mass Customization of Housing* (Correia, Duarte, & Leitão, 2012) presented at eCAADe12.

1.4 Outline

The remaining chapters of this dissertation are organized as follows: Chapter 2 introduces the topics that are relevant for this dissertation. Chapter 3 presents a general overview of the proposed SA. Chapter 4 explains the details of DESIGNA, a SG interpreter which uses the proposed SA. Chapter 5 evaluates DESIGNA with the implementation of three SGs, where we introduce a space of discussion addressing the development of SG interpreters and the results obtained from the implementation of the three SGs. Finally, in chapter 6 we present conclusions and future research paths.

2 Related Work

The scope of this dissertation falls within the area of generative systems. More precisely, this dissertation uses the formalism of shape grammars (SGs) to generate design solutions for the mass customization of housing.

In this chapter we will present the related work and present the relevant concepts for this work. In section 2.1 is presented an overview of generative systems. In sections 2.2 and 2.3 are described the shape grammar (SG) formalism and presented the current SG interpreters. Finally, section 2.4 explains discursive grammars and its composition.

2.1 *Generative Systems*

A *generative system* is the broad term that applies to the process of having a computer do specific calculations based on patterns, relationships and interactions that result in complex output.¹ Initially developed within the area of biology and mathematics, generative systems have also been applied in areas like architecture, music, art,² and computer games.³

Briefly are presented the more significant examples of generative systems, (1) cellular automata, (2) l-systems, (3) voronoi diagrams, (4) fractals, (5) shape grammars, and (6) others:

- *Cellular automata* - developed by Stanislaw Ulam and John Von Neumann in the late 40's. It is a discrete model that simulates the growth of cells in a grid. This growth is described by rules that relates one cell growth to its neighbors. The best known version of this generative system is the game of life by John Horton Conway in the 1970, and first introduced by Martin Gardner in his "Mathematical Games" column in the Scientific American magazine (Gardner, 1970).
- *L-systems* - developed by Aristid Lindenmayer⁴ in the late 60's (Lindenmayer, 1968). It is used to simulate the growth of plants. The growth is simulated by the application of rules to an initial sequence of characters. Each character encodes a given characteristic of the plant and each rule describes how a given character or sequence is replaced by another character.

¹From: <http://davidnbrooks.com/journal/generative-systems>

²<http://www.n-e-r-v-o-u-s.com>

³<http://www.spore.com>

⁴<http://algorithmicbotany.org/papers/#abop>

- *Voronoi diagrams* - represents a way to "the partitioning of a plane with points into convex polygons such that each polygon contains exactly one generating point and every point in a given polygon is closer to its generating point than to any other".⁵
- *Fractals* - represents a geometric pattern, which, when divided, reproduce the original pattern but in smaller scale. The term fractal was first used by Benoit Mandelbrot in 1975, where the best known fractal is the Mandelbrot set.
- *Shape grammars*⁶ - developed by George Stiny and James Gips (Stiny & Gips, 1972). Allows the generation of designs by recursively applying rules to an initial shape. Rules describe transformations that happen to a shape that is found on a design. In this group we also include string grammars. Although related to shape grammars, string grammars use sequence of characters instead of shapes to describe designs. The best know example is the system called CityEngine⁷ (Parish & Müller, 2001). Shape grammars are describe in more detail in the following section.
- *Others* - this group includes current applications that accept algorithms described in a visual programming languages. The algorithms are described with boxes, representing black boxes of functionality, and links, that connect two or more boxes, that represent data flow between boxes. The best known examples are (1) Grasshopper3D,⁸ and (2) Generative Components.⁹

2.2 Shape Grammars

Shape grammars are generative systems based on rules that allow capturing, creating, and understanding designs. They are based on the production systems of Emil Post (Post, 1943) and the generative grammars of Noam Chomsky (Chomsky, 1957). Instead of using words to form sentences, shape grammars work directly with shape computations rather than through symbolic computations (Knight, 2000).

Designs are formed by shapes, where a shape is conceived as a finite collection of maximal lines (Stiny, 1980a). Designs are created by recursively applying a set of rules to an initial shape until either a design is completed, or any other condition is meet, or no more rules can be applied. In general, several rules can be applied to any given shape, thus producing many different designs. And by applying rules in a different order, the final result is the generation of designs which belong to the same language or style.

Labels were introduced to help reduce symmetries in shapes, and to control or limit which rules can be applied to a given shape at any given time (Stiny, 1980a). At any stage of the rules application,

⁵from: <http://mathworld.wolfram.com/VoronoiDiagram.html>

⁶<http://www.shapegrammar.org/>

⁷<http://www.esri.com/software/cityengine>

⁸<http://www.grasshopper3d.com/>

⁹<http://www.bentley.com/en-US/Products/GenerativeComponents/>

the labels could be added or removed from shapes. Labels can be letters, symbols, points or something else, and can be associated to any shape.

2.2.1 Mathematical Definition

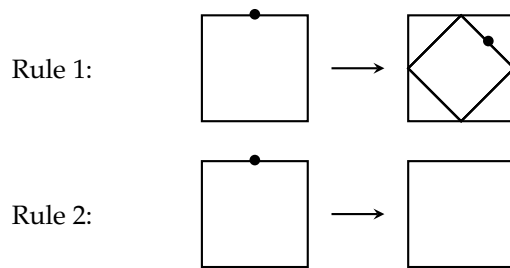
Formally, a shape grammar [SG] is a four-tuple (Stiny, 1980a) namely: (1) a vocabulary of shapes [S], (2) a set of labels [L], (3) an initial shape [I], and (4) a set of rules [R] (see equation 2.1) (Stiny, 1980a):

$$SG = (S, L, I, R) \tag{2.1}$$

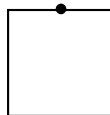
Each rule in a SG is seen as $A \rightarrow B$, where A and B represents a shape. To apply the rule $A \rightarrow B$ to the design C is necessary to know if shape A is a sub-shape of design C. This means that is necessary to find an Euclidean transformation T() (translation, rotation, reflection, scale or a combination of the above) to identify the shape A within sub-shapes of the design C resulting in a new design C' (see equation 2.2):

$$C' = C - T(A) + T(B) \tag{2.2}$$

An example of a simple SG (from (Stiny, 1980a)), is shown in figure 2.1. The symbol • is a label and is located in the midpoint of the given shape edge. This label helps reduce the shape symmetries (expressed in rule 1) and allows to stop a derivation of a design (expressed in rule 2).



(a) Shape grammar rules



(b) Initial shape

Figure 2.1: A simple shape grammar inscribing squares in squares

In figure 2.2 is shown one possible generation of a shape using the SG defined in figure 2.1. And in figure 2.3 is shown some possible final designs that the SG define in figure 2.1 can generate.

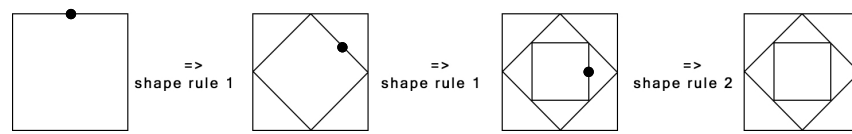


Figure 2.2: Generation of a shape using the SG of figure 2.1



Figure 2.3: Some shapes in the language defined by the SG of figure 2.1

2.2.2 Parametric Shape Grammars

Parametric SGs (Stiny, 1980a), are SGs that allow the shapes to which rules are applied to have parameters. These parameters are generally specified in terms of equations or constraints and whenever an assignment of real values to these parameters are found a parametric shape becomes a shape. Parametric grammars can generate an even greater variety of designs, even though this increases flexibility entails a more complex implementation mainly because the number of design solutions that a system can produce becomes extremely large, if not infinite.

Again, to apply the rule $A \rightarrow B$ to the design C is necessary to know if shape A is a sub-shape of design C . This means that is necessary to find an Euclidean transformation and also the respective parameters to identify the shape A within sub-shapes of the design C resulting in a new design C' (see equation 2.3):

$$C' = C - T(g(A)) + T(g(B)) \quad (2.3)$$

In equation 2.3 functions $T()$ and $g()$ represent, respectively, the Euclidean transformations and parameters which are valid to identify the rule to apply.

In figure 2.4 (from (Stiny, 1980a)), is shown one parametric SG. This SG is a generalization of the SG defined in figure 2.1. In figure 2.5 is shown a generation of a shape using the SG defined in figure 2.4. From this example is clear that by introducing parameters in shapes one can introduce more complex shapes but the SG can also generate more diverse designs as show in figure 2.6.

2.2.3 Emergence

When applying rules to an initial shape, new shapes could emerge or be recognized from the generated design. These new shapes can also be used when applying recursively rules to shapes. Thus, emergence is the ability to recognize and, more importantly, to operate on shapes that are not predefined in a

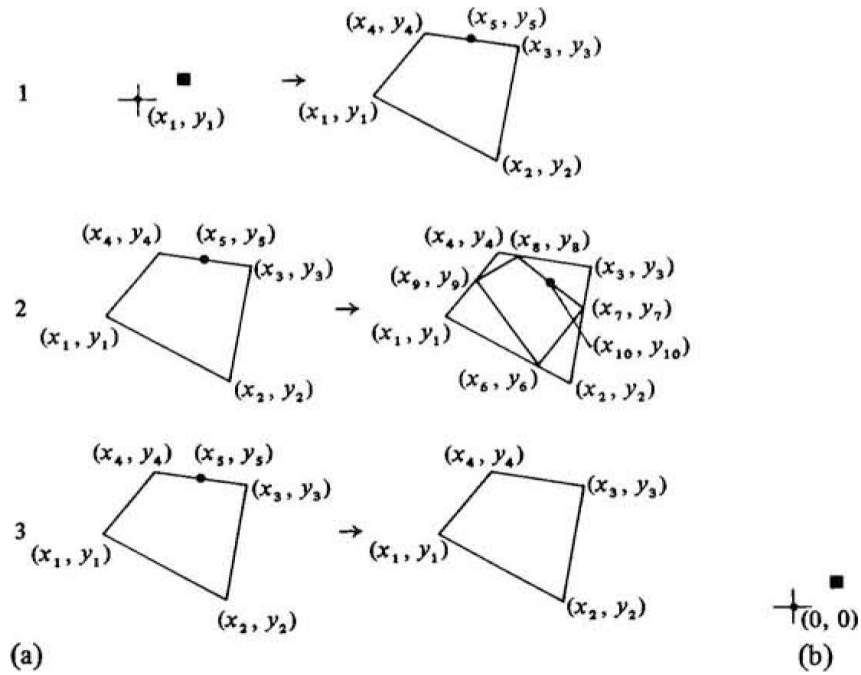


Figure 2.4: Simple parametric shape grammar, inscribing convex quadrilaterals into convex quadrilaterals: (a) Shape rule schemata, (b) initial shape

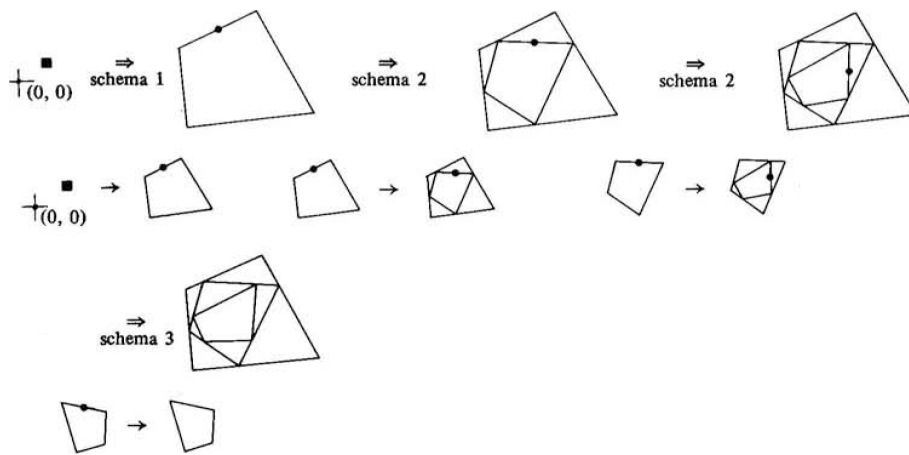


Figure 2.5: Generation of a shape using the SG of figure 2.4

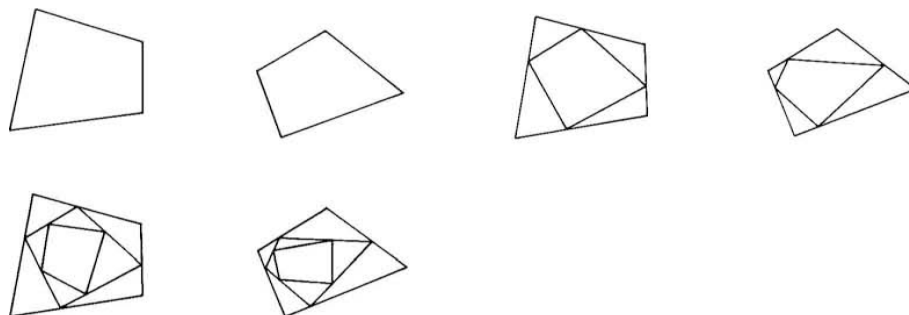


Figure 2.6: Some shapes in the language defined by the SG of figure 2.4

grammar but emerge, or are formed, from any parts of shapes generated through rule applications (Knight, 2000).

As an example, consider the SG rule shown in figure 2.7a (Knight & Stiny, 2001). This rule states that, if a square is found, then delete the square, and add another square with a rotation of 45° . Consider now, a shape that was generated by two equal squares and translated relative to on another. This shape is shown in figure 2.7b. Finally consider the application of the former rule to this shape. While the initial shape is composed by only two squares, there are three possible solutions, shown in figure 2.8. The first, is generated by applying the rule to the upper square (figure 2.8a). The second, is generated by applying the rule to the lower square (figure 2.8b). And the third, is generated by applying the rule to the inner square (figure 2.8c). While this inner square was not defined by the initial shape, it is still a valid square to apply the rule. Thus, is called as emergent.

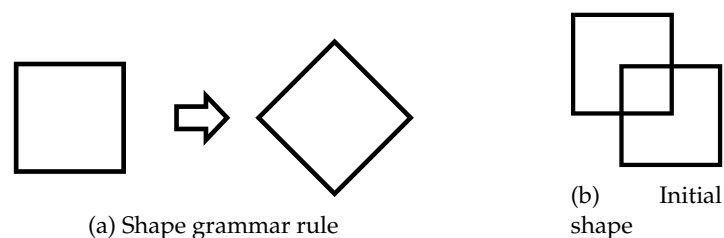


Figure 2.7: Example of a shape grammar

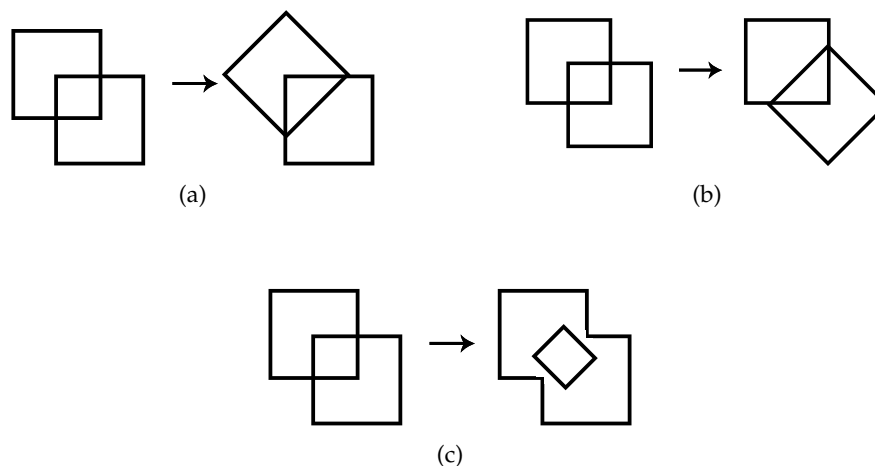


Figure 2.8: Rule application of shape grammar of figure 2.7

2.2.4 Analytical and Original Grammars

To a designer, SGs can provide a system that allows him/her to explore the set of solutions for a given design problem. Further, SGs can allow a designer through analysis of an existing design to encode rules representing it. Thus SGs can be likewise generative and descriptive. They could be used to generate shapes and, at the same time, describe shapes. Generically, SGs can be divided into two categories:

(1) analytical grammars and (2) original grammars. Analytical grammars, more deeply explored than original grammars studies, are developed to analyze and describe existing design styles. On the other hand, original grammars are created to generate new and original designs.

2.2.5 History

Originally developed by Stiny and Gips (Stiny & Gips, 1972) to illustrate the application of SGs for interpreting and evaluating works of art (Knight, 2000), over the years were developed a broader range of SG research studies including:

- Parametric shape grammars, where designs are computed with parametric shapes (Stiny, 1980a)
- Description grammars, that generate descriptions of designs (Stiny, 1981)
- Color grammars, where colours are used to give meaning to shapes (Knight, 1989a)
- Parallel grammars, which simultaneous compute different shapes or symbolic representations of designs (e.g. generate plants, sections (Stiny, 1991)
- Structure grammars, where designs are generated as structures or set of shapes (Carlson, Woodbury, & McKelvey, 1991)
- Grammars with weights, where shapes have an weight (number, material, etc) associated (Stiny, 1992)
- Attributed grammars, that generate shapes with attributes and constraints (Brown, McMahon, & Williams, 1994)

Due to the work developed by Knight (Knight, 2000), SGs are also present in higher education. Examples of such curriculums are the courses by Terry Knight at Massachusetts Institute of Technology, by José Duarte at Faculdade de Arquitectura de Lisboa, and by Gabriela Celani at FEC-UNICAMP.

2.2.6 Examples

Initially developed in the area of painting, SGs were increasingly covering another application areas, mainly architecture, engineering and product brand identification. Examples of SGs include: Iceray (Stiny, 1977), The Palladian grammar (Stiny & Mitchell, 1978), Buick vehicles (McCormack, Cagan, & Vogel, 2004), Queen Anne houses (Flemming, 1987), Froebel's building gifts (Stiny, 1980b), Coca-Cola and Head & Shoulders bottles (Chau, Chen, Alison, & Alan, 2004), Hepplewhite-style chair-back designs (Knight, 1980) , Urform grammars (Stiny & Gips, 1972), Coffee machines grammar (Agarwal & Cagan, 1998) , Stijl art grammar (Knight, 1989b), Mughull gardens grammar (Stiny & Mitchell, 1980)

(Knight, 1990), Lloyd Wright's prairie houses grammar (Koning & Eizenberg, 1981), Taiwanese traditional vernacular dwellings grammar (Chiou & Krishnamurti, 1995) (Chiou & Krishnamurti, 1996), The bungalows of Buffalo (Downing & Flemming, 1981), Japanese Tea rooms grammar (Knight, 1981), Queen Anne houses grammar (Flemming, 1987), Wren's language of City church designs (Buelinckx, 1993), Traditional Turkish houses (Cagdas, 1996), Vernacular hayat houses (Colakoglu, 2005), Álvaro Siza Malagueira houses (Duarte, 2001), Casa Giuliani Frigerio (Flemming, 1981), Oscar Niemeyer Architecture (Mayer, 2003), Style of Yingzao fashi houses (Li, 2001), Marrakech Medina urban form (Duarte, Rocha, & Soares, 2007), Patio Houses of the Medina of Marrakech (Duarte & Rocha, 2006), Meander Motif on Greek Geometric (Knight, 1986), Mies van der Rohe's high-rise apartment grammar (Moon, 2007), Vehicles (Orsborn, Cagan, Pawlicki, & Smith, 2006), BMW¹⁰, VW Beetle¹¹, Inner hood panels (McCormack & Cagan, 2002), Buick (McCormack et al., 2004), Harley-Davidson Brand (Pugliese & Cagan, 2002), Classic Ottoman Mosques (Sener & Gorgul, 2009), Geometric Islamic Ornaments (Cenani & Cagdas, 2006), Traditional Malay Long-Roof Type Houses (Said & Embi, 2008).

2.2.7 Full Example

In figure 2.9a is shown a Renaissance church design based on a Greek cross. Based on this design, Terry Knight developed a SG that can generate these and other designs. The full SG is shown in figure 2.9b, along with a possible derivation and few designs that this SG can generate (from (Knight, 1994)).

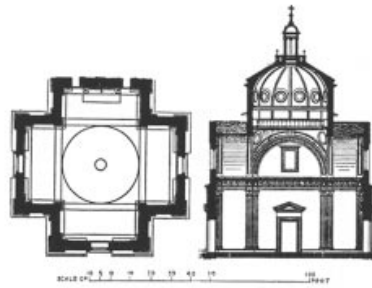
The rule says that if (the leftside of the rule) a two by one rectangle can be found in a design in any orientation or any size, then (the rightside of the rule) another two by one rectangle can be laid on top to form the Greek cross (Knight, 1994).

2.3 Shape Grammar Interpreters

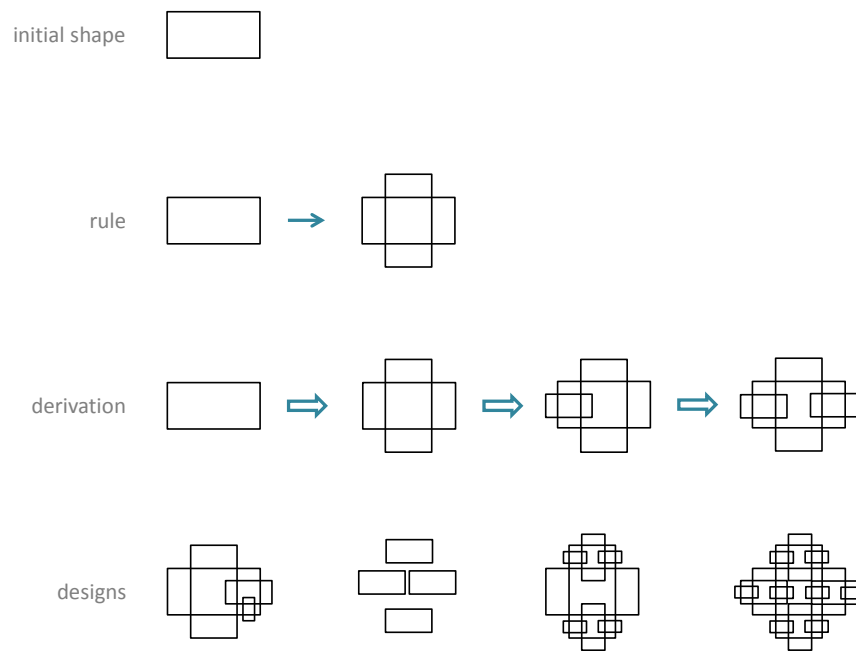
To automate the application of shape rules, researchers have concentrated their efforts on developing SG interpreters. Previous summaries (Gips, 1999), (Chau et al., 2004) show that these researchers have focused on representations of shapes, algorithms for subshape detection and emergence, user interaction, and integration into the design process. The table 2.1 summarizes the list of SG implementations presented in those two summaries. This table was also updated and augmented with more recent references of SG implementations.

¹⁰Ducla, G. D.: BMW Grammar. MIT: 4.201 Computational design final project (2002)

¹¹Arida, S: VW Beetle Grammar. MIT 4.201 ComputationalDesign final project (2002)



(a) Renaissance church design



(b) Shape grammar, with derivation and possible designs

Figure 2.9: Full example of a shape grammar

Table 2.1: Shape grammar implementations

	Name	Reference	Tool(s)	Subshape	2D/3D
1	Simple interpreter	(Gips, 1975)	SAIL	No	2D
2	Shepard-Metzler analysis	(Gips, 1974)	SAIL	No	2D/3D
3	Shape grammar interpreter	(Krishnamurti, 1982)		Yes	2D
4	Shape generation system	(Krishnamurti & Giraud, 1986)	Prolog	No	2D
5	Queen Anne houses	(Flemming, 1987)	Prolog	No	2D
6	Shape grammar system	(Chase, 1989)	Prolog	Yes	2D
7	Genesis (CMU)	(Heisserman, 1992)	C/CLP	No	3D
8	GRAIL	(Krishnamurti & Earl, 1992)		Yes	2D
9	Grammatica	(Carlson, 1993)		No	
10		(Stouffs, 1994)		Yes	2D/3D
11	Genesis (Boeing)	(Heisserman, 1994)	C++/CLP	No	2D/3D
12	GEEdit ¹²	(Tapia, 1996)	Lisp	Yes	2D
13	Shape grammar editor	* <i>Shelden</i> 1996	AutoLisp	Yes	2D
14	Implementation of basic grammar	(Duarte & Simondetti, 1997)	AutoLisp	No	3D
15	Shape grammar interpreter	(Piazzalunga & Fitzhorn, 1998)	ACIS/LISP	No	3D
16	SG-Clips	(Chien, Donia, Snyder, & Tsai, 1998)	CLIPS		2D/3D
17	3D architecture form synthesis	(Wang, 1998)	Java/Open Inventor	No	3D
18	Coffee maker grammar	(Agarwal & Cagan, 1998)	Java		2D/3D
19	MEMS grammar	(Agarwal, Cagan, & Stiny, 2000)	LISP		2D
20	Shaper 2D ¹³	(McGill, 2001)	Java	No	2D
21	U_{13} shape grammar implementation	(Chau, 2002)	Perl	Yes	3D
22	Designing inner hood panels	(McCormack & Cagan, 2002)	Autocad	Yes	2D/3D
23	A simulated shape grammar for generating building sections according to the Yingzao Fashi	(Li, 2001)	Flash	No	2D
24	SGCAD ¹⁴		AutoLisp	No	2D
25	Computational environment for learning	(Wong & Cho, 2004)	Java	No	2D

Continued on next page

¹²available at: <http://www.shapegrammar.org/GEEdit/GEEdit.html>¹³available at: http://designmasala.com/portfolio/ui_shaper2d.html¹⁴available at: <http://web.mit.edu/haldane/www/sgcad/index.html>

Table 2.1 – continued from previous page

	Name	Reference	Tool(s)	Subshape	2D/3D
26	Bracket system	(Wu, 2005)	Actionscript	No	2D
27	Implementation of a description grammar	(Duarte & Correia, 2006)	Java	No	-
28	Qi	(Prats, Earl, Garner, & Jowers, 2006)	Java	Yes	2D
29	Classique Ottoman Mosques	(Sener & Gorgul, 2009)	Delphi	No	2D/3D
30	Design Synthesis and Shape Generation ¹⁵	(McKay, Jowers, Chau, Pennington, & Hogg, 2008)	Java	Yes	2D/3D
31	Shape Designer V.2		Java/Prolog	No	2D/3D
32	Shape grammar interpreter ¹⁶ ¹⁷	(Trescak, Esteva, & Rodriguez, 2010)	Java	Yes	3D

The implementation of Krishnamurti was the first implementation that feature the maximal line representation of straight lines as described in (Stiny & Gips, 1972). It introduce the use of homogeneous coordinates and attempted to overcome the problem of using non exact arithmetic. Flemming, with his set grammar of Queen Anne houses introduces the three dimensional geometry in research and implementations of SGs. Heisserman also introduces a milestone with his Genesis implementation (shown in figure 2.10c), and while the details are not of public domain, it started a commercial interest in SGs. Tapia's GEdit (shown in figure 2.10a) provided basic elements using maximal lines and allowed sub shapes to be identified and therefore enabled emergence. Even that the options were limited, it could only represent two dimensional lines, GEdit provided an interface which allowed a user to select the next rule to be applied. Cagan, with his coffee maker implementation brought the SG formalism to the design of consumer products. The Qi implementation (shown in figure 2.10d) introduced computer vision to the application of SGs, but its poor performance, still limits the design solutions available.

Moreover, current SG interpreters are not yet common and easy available as a general application that a designer can use out of the box. We support this with the results from one of the latest Design Computing Cognition conference in 2010.¹⁸ From more than three decades of its inception, the prototypes of SGs interpreters shown are very specific interpreters in what they can expressed, represent and generate.

Current developments include the combination of more traditional approaches to rectilinear shapes with computer vision to handle as well curvilinear shapes.¹⁹ While other approaches are exploring

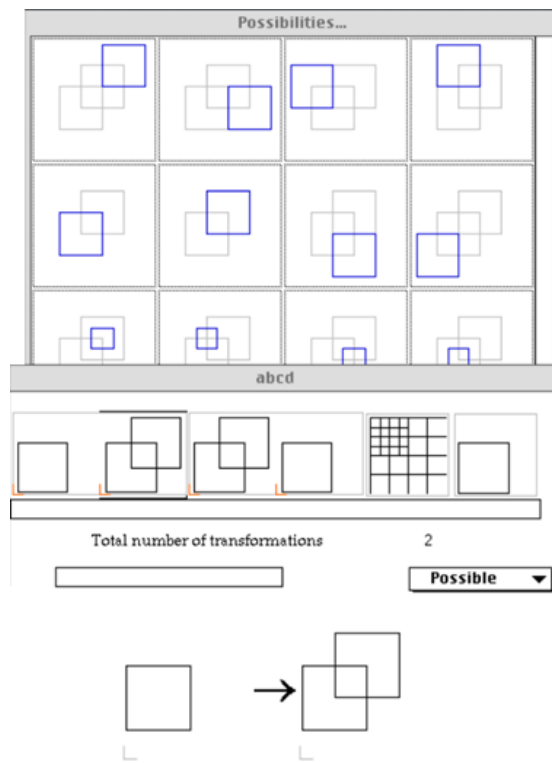
¹⁵available at: <http://www.engineering.leeds.ac.uk/dssg/downloads/requestForm.php>

¹⁶available at: <http://sourceforge.net/projects/sginterpreter>

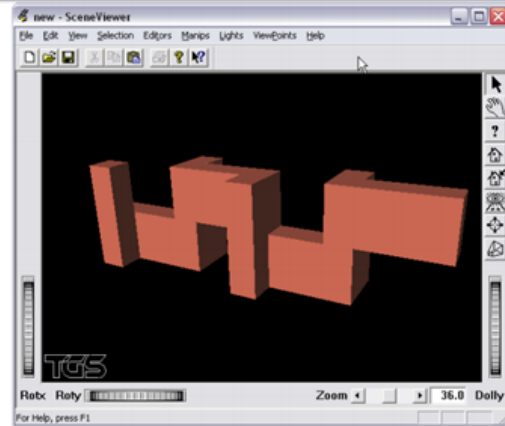
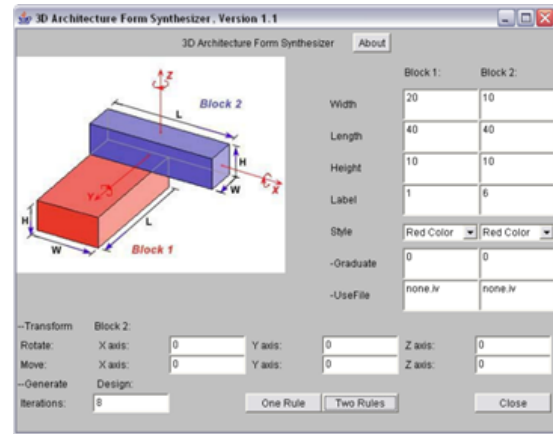
¹⁷more information at: <http://www2.iiia.csic.es/~ttrescak/sgi.html>

¹⁸<http://www2.mech-eng.leeds.ac.uk/users/men6am/WorkshopReport.htm>

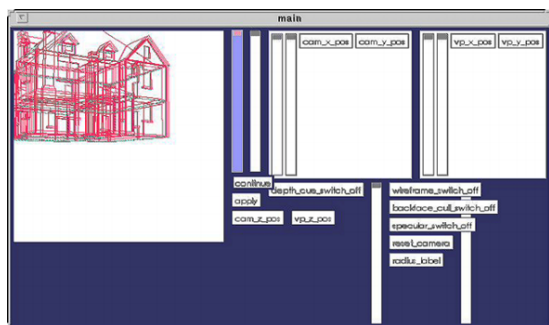
¹⁹<http://tomitrescak.blogspot.pt/2010/07/dcc-2010-conference-and-shape-grammar.html>



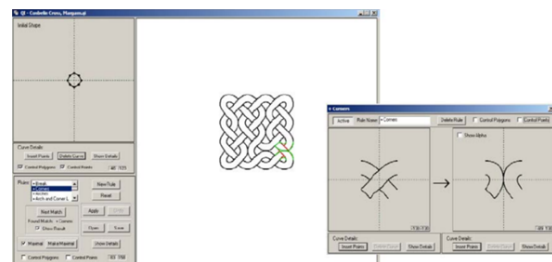
(a) GEdit



(b) 3D shaper



(c) Genesis



(d) Qi

Figure 2.10: Screenshots of shape grammar interpreters

graph grammars in order to combine a graph representation of parametric shapes and the ability to recognize sub-shapes.²⁰

Finally, and perhaps the most popular approach using SGs, is the application CityEngine (shown in figure 2.11),²¹ which allows the generation of detailed city models using a procedural modeling approach. Their approach handles the generation of city buildings using an initial two-dimensional shape, extruding it and applying textures to create façades.

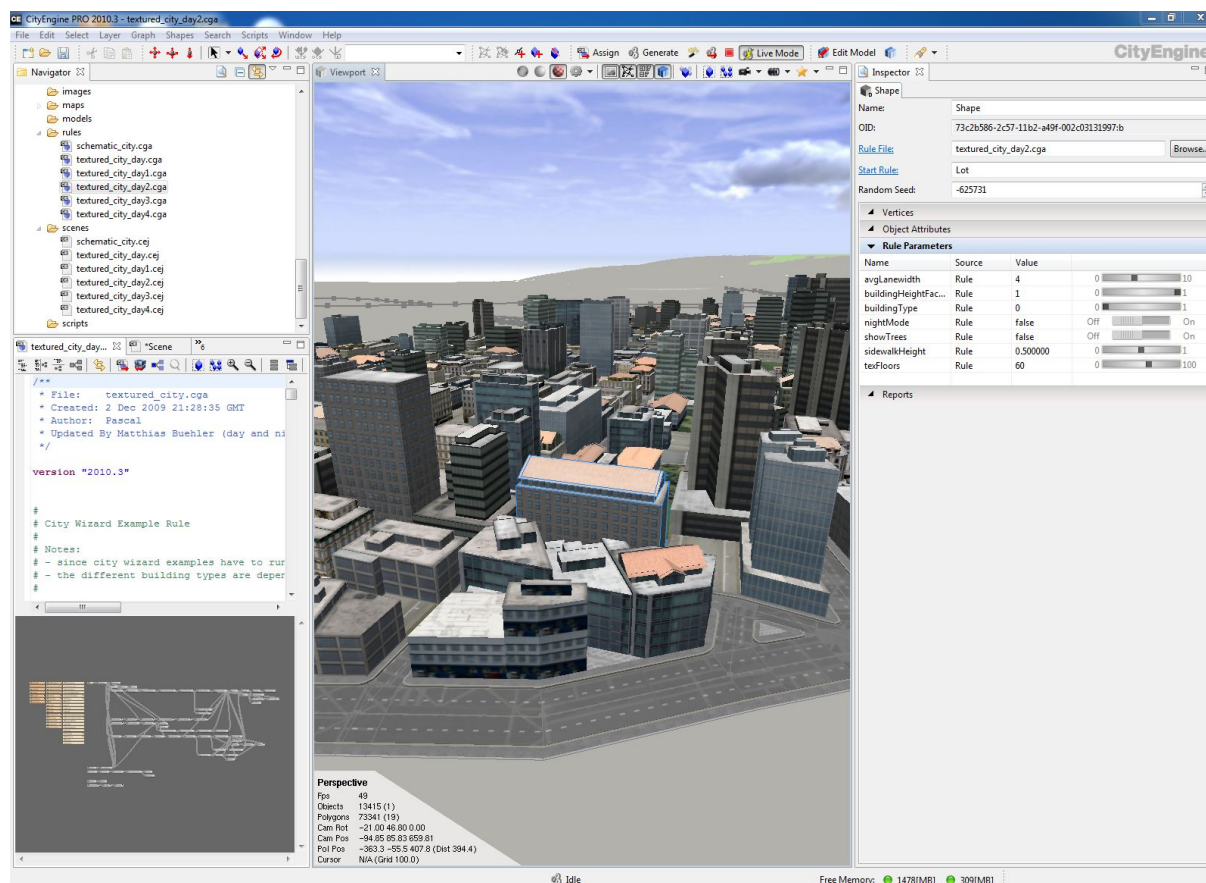


Figure 2.11: Screenshot of CityEngine

2.3.1 Visual, Symbolic and Set Grammars

Implementations of SGs can be divide in three main groups: (1) visual, (2) symbolic, and (3) set grammars. Visual implementations are characterized by identifying shapes directly in the geometry. This feature approximates these implementations to the mathematical bases of SG defined by Stiny (Stiny, 1980a). Examples in this group are the implementations by Krishnamurti, (Krishnamurti, 1982) and (Krishnamurti & Earl, 1992). Unfortunately they only deal with 2D shapes, as 3D shapes poses technical

²⁰<http://plus.swap-zt.com/projekt/grape/>, web-based: <http://grape.swap-zt.com/>

²¹<http://www.esri.com/software/cityengine>

and mathematical problems still to be solved. The third group is characterized by representing shapes as atomic entities. An example contained in this group is the SG implementation by McGill (McGill, 2001).

Because current computer systems where these SG are being implemented are symbolic by nature, the separation of the former two groups is very thin and in fact, this separation can be viewed as an abstraction.

2.4 *Discursive Grammar*

Given that the main focus of the evaluation of the SG interpreter developed in this dissertation is the implementation of a SG for the mass customization of housing, and because this SG is part of a discursive grammar (DG), this section introduces the concept of a DG and a brief overview of its original system architecture.

Note that, in this section, DESIGNA is the second module of the DG, but in the scope of this dissertation, DESIGNA is also the name of the SG interpreter developed. The same nomenclature is followed for PROGRAMA, which is both the name of the first module of the DG and the name of its implementation (already discussed in (Duarte & Correia, 2006)).

2.4.1 Prologue

To solve the problem faced by a designer has when he conceives a project for a large development, Duarte (Duarte, 2005b) proposes the use of a DG, a rigorous mathematical model for the generation of designs according to a housing brief.

A DG is a grammar that is capable of generating both syntactically and semantically correct designs. In other terms, it deals with both form and meaning so that it finds a design within the language that matches a given criteria (Duarte, 2001).

2.4.2 Two Viewpoints

A DG is defined by two viewpoints: (1) a technical, and (2) an operational. In figure 2.12 is shown the relation between the two.

From a technical point of view, a DG is composed of a description grammar, a SG, and a set of heuristics. For each rule in the SG, there is a corresponding rule in the descriptive grammar, so that as the shape evolves by rule application, its design description is also continuously updated. Heuristics are used to guide the application of rules, for example, to select or limit the rules that can be applied at each step of the generation or to evaluate and select the entities that are closer to some pre-established

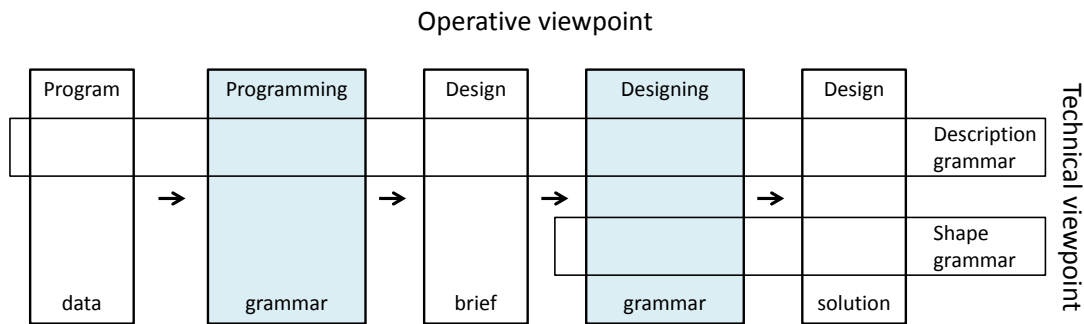


Figure 2.12: Discursive grammar viewpoints

goal. This allows discursive grammars to generate designs that are not only formally valid but are also semantically correct.

From the operational point of view, the DG is formed by two independent grammars, linked in sequence. The former is a programming grammar that is encoded as a description grammar, formulating a housing brief constrained by user input and a set of regulations. This programming grammar was already discussed in (Duarte & Correia, 2006). The result of the programming grammar, a housing brief, is then used as input to the designing grammar which is encoded as a description grammar and a SG tied together. These grammars compute a house design whose description matches the housing brief.

2.4.3 Mathematical Definition

Formally, a DG is a ten-tuple (Duarte, 2001), namely: (1) a set of description rules [D], (2) an initial description [U], (3) a goal description [G], (4) a set of heuristics used to decide which rule to fire at each stage and guide the solution [H], (5) a set of SG rules [S], (6) a set of labels used to control computations [L], (7) a set of weights associated with shapes [W], (8) a set of similarity transformations [T], (9) a set of functions that assigns values to parameters in rules [F], and (10) the initial shape [I] (see equation 2.4). Note that, heuristics are to description rules as labels are to shape rules; they constrain the way in which rules are triggered and fired (Duarte, 2001).

$$DG = (D, U, G, H, S, L, W, T, F, I) \tag{2.4}$$

2.4.4 System Architecture

In figure 2.13 is shown the generic system architecture for the DG. The system is composed of two modules: (1) PROGRAMA and (2) DESIGNA. PROGRAMA encodes the rule set of a given country regulation in a programming grammar. PROGRAMA, based on user and site data, generates the design brief (DB), that is, a symbolic description of an adequate house design that satisfies the encoded

regulations. DESIGNA encodes the architectural styles of a given Architect in a designing grammar. DESIGNA accepts as input a DB encoding user and site data corresponding to all the previous goals and constraints. With this data, DESIGNA computes a set of designs according to the architectural style encoded in its designing grammar.

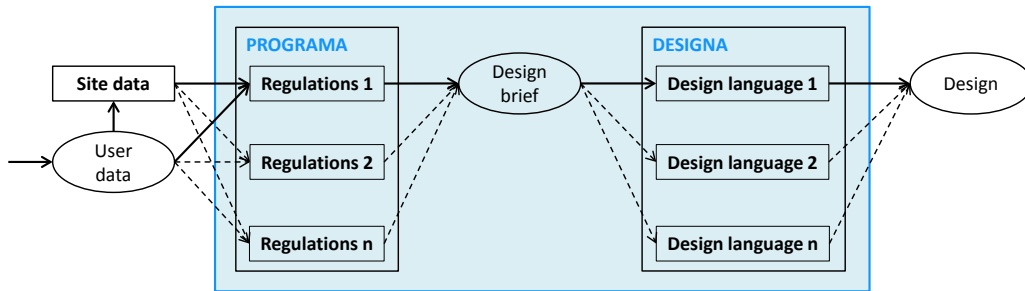


Figure 2.13: Discursive grammar system architecture

In theory, different programming grammars can be combined with different designing grammars to form various discursive grammars (Duarte, 2001). For instance, using the Swedish regulations to encode rules in a programming grammar, and using the style of Frank Lloyd Wright's to generate designs encoded in a designing grammar.

In practice, the independence of programming grammars from designing grammars is limited (Duarte, 2001).

For experimentation and evaluation purposes, these grammars use realistic grammars (as proposed in (Duarte, 2001)): the programming grammar follows the rules of PAHP - the Portuguese housing program and evaluation system, while the designing grammar encodes the rules laid out by the Architect Álvaro Siza for the design of the Malagueira houses, an award-winning project that is under construction since 1977.

In figure 2.14 is shown the proposed PAHPA-Malagueira grammar (Duarte, 2001).

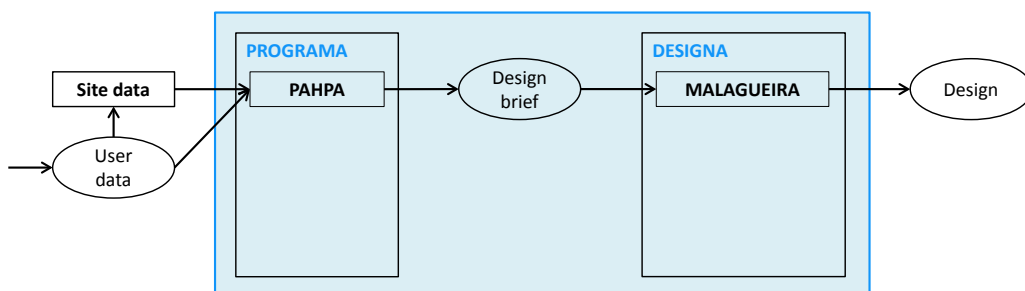


Figure 2.14: PAHPA-Malagueira discursive grammar

3 Overview

Despite all the advances in computer-aided design (CAD) applications, it is still difficult to develop large scale projects and provide mass customization of housing, that is, provide many different and personalized design solutions and still satisfy user needs and preferences.

Shape grammars (SGs) allow the generation of designs by recursively applying rules to an initial shape. Thus, a SG is a rule-based generative system, which allows the capture, creation, and understanding of designs. While research has been focused in new ways to express and capture designs, the application of rules, or the generation of design solutions, are still predominantly manual processes (as with the GEdit implementation presented previously (Tapia, 1996), among others). Consequently, many design solution paths are not explored. Although this is a limitation, it also enables the use of the formalism to converge to a solution based on the user's common sense. In order to fully exploit the SG formalism, SG interpreters have been developed. These allow the automatic application of rules to shapes, and therefore generate significantly more solutions than a manually approach could have accomplished. Unfortunately, current SG interpreters still lack some basic features for a generic use of the SG formalism, such as graphical user interface (GUI), and automatic shape recognition. Moreover, these SG interpreters usually suffer from (1) limited shape representation, (2) poor control of rule application, and (3) in their majority, do not support directly the designers standard CAD tool to present or further work the solutions, but instead provide their own visualizer, introducing one more step in the designers workflow.

This thesis proposes a new software architecture (SA) for a SG interpreter that overcomes some of the limitations of current SG interpreters. Focus is given to the basic features that a SG interpreter needs to support, namely: (1) labeled shapes, (2) rules, and (3) smooth integration in the workflow of the designer.

These features and how they work together are explained in the next sections, starting with the proposed shape and rule representation and description, followed by how the solutions are presented, and finally, the overall look of the SA proposed.

3.1 Shapes

Because current computer systems are implicitly symbolic, a SG interpreter that is implemented in these systems needs to represent a shape symbolically. While there are many ways of describing and representing shapes, see (Grasl & Economou, 2011), we explore the work of Heisserman (Heisserman, 1992) regarding logical reasoning about solids using first-order logic.

Like in Heisserman work, we also describe a shape by its boundary using a graph representation. This representation allows us to describe shapes in terms of facets, edges and vertices, and their relations between each other, where geometric information, or points, is assigned to vertices. Therefore, this representation allows a clear separation between topology and geometry. The first is concerned with relations among the shape elements, such as connectedness and adjacencies of facets. And, the second is concerned with geometric properties of the shape, such as lengths, angles, or areas of such elements. A shape is created and modified using Euler operators. These are topological operators that only modify the incident relationship between shape elements. Lastly, labels, representing a number, a string or any other type of non-geometric information can be assigned to any of the shape elements.

This more complex representation also brings some advantages. Shapes can be mapped directly into a graph, and within this graph, computations can be done. Shapes can be modified, topologically or geometrically, and independently. Also, queries can be computationally made, in a quick and inexpensive way. Finally, with graphs, we can easily translate one representation into other, e.g. for visualization purposes.

As an example to illustrate this representation, consider a tetrahedron (see figure 3.1a). A tetrahedron can be described as having four triangular facets (see figure 3.1b), six edges, four vertices (see figure 3.1c), and four points (not showed in the figure). Using the graph representation (see figure 3.2), each facet is connected with three edges and three vertices, and shares all the edges and all the vertices with the other three facets, where each vertex has an assigned point (not showed in the figure).

The more complex nature of this representation allows us to answer questions like, to what facets is an edge connected. Or even, to navigate through all shape elements.

3.2 Rules

As SGs are rule based systems, the application of rules becomes central to the generation of solutions. Unfortunately, current implementations provide little support to express the order in which the rules are to be applied, forcing the user to rely on mechanisms that were not developed for that end, namely: (1) prioritizing rules, using the salience property associated with each rule, and (2) using SG labels. These mechanisms are discouraged, first, because trying to force rules to fire in a particular order is considered bad style in rule-based programming and will have a negative impact on performance of rule engines,

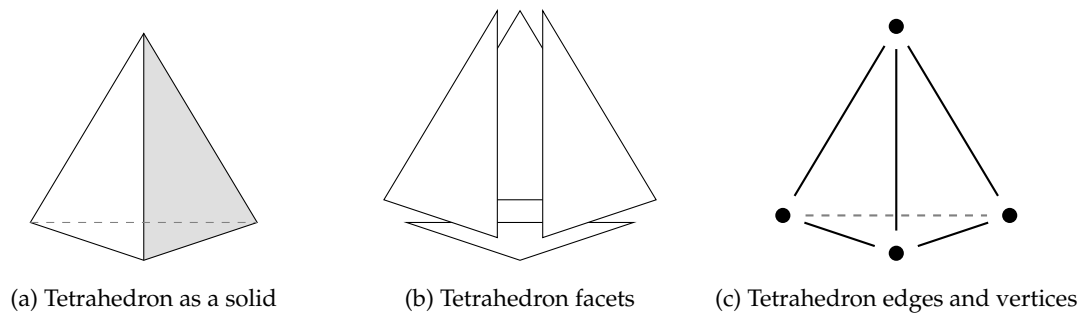


Figure 3.1: Example of a tetrahedron shape and its boundary elements (the dashed style represent hidden elements)

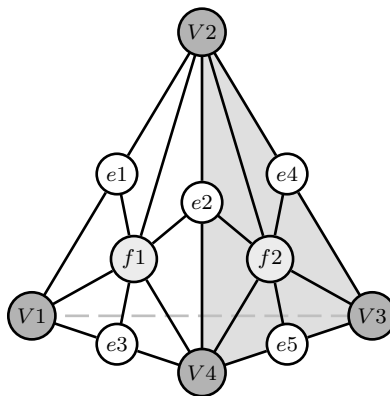


Figure 3.2: Tetrahedron shown as a graph (not all elements are represented) - the nodes are topological elements and the arcs represent relations

and secondly, because SG labels were not created to allow the specification of the order in which rules are to be applied, but instead, to annotate shapes, which can restrict the rule application because of its presence. Obviously, there is a difference between restricting the application of a rule and specifying its order of application.

Our proposal to overcome these limitations is to describe SG rules as transition operators, which operate on states, encoding a particular design, and using strategies to express the order in which these operators are applied. Therefore the generation of designs is the result of applying a set of operators to states, moving the states towards a solution. With this approach we detach the description of rules from the order in which they are applied.

Moreover, SG rules usually condense too much information in one rule. For a human reader, this is acceptable, as his knowledge and common sense allow him to understand these rules. Unfortunately, current computer systems do not have the same capabilities and cannot understand typical SG rules as easily as humans can.

Lastly, while the exponential grow of possible solutions that can be generated using SGs can not be solved easily, our approach to this issue is to generate solutions incrementally and provide the user with an extensible set of strategies that the user can choose in order to control the application of rules and generate solutions.

3.3 Output

The way the design solution is presented to the user is also important, because it will dictate if the solution can be simply visualized or if it can also be modified in the user standard CAD tool. Unfortunately, the majority of SG interpreters provide their own visualizer, which has the advantage that the interpreter developer does not depend on third party tools, and directly controls all available options. However, there are also some serious disadvantages. First, developing another tool to visualize geometry, ultimately develop another modest CAD tool, takes time and resources, which would be better invested in the SG interpreter development. Second, the user of a SG interpreter also knows how to use a standard CAD tool. With this approach the user needs to learn and yet use another tool to do what he already does with his CAD tool. And, third, the user of such interpreters needs one more step, if available, to further work on the design solution, in a standard CAD tool.

Another approach is to implement the SG interpreter directly in the language provided by the standard CAD tool. While this has the advantage that the generated solutions can be directly manipulated in the CAD tool, it has one major disadvantage: the programming languages and frameworks provided by current CAD tools are not usually expressive enough to be usable for the development of complex programs. Instead, they were designed to provide simple and small customizations of the tool, easing its use. Thus, a SG interpreter implemented in these languages will be difficult to develop and will be

restricted to the functionality provided by its development language and the restrictive functionality provided by the CAD tool.

Again, we take another approach to the way a design solution is presented to the user. Instead of developing our own visualizer, or rely on restrictive programming languages that work directly in the CAD tool, our approach is to develop a SG interpreter in a capable programming environment, providing a modern and capable programming language, which can also abstract the CAD tool, providing a way of communication between the two.

3.4 Software Architecture

Independently of what designs a SG capture, a SG interpreter needs to handle shapes, rules and somehow, output the generated design solutions. To support this approach, the proposed SA has two core components and one loosely coupled component. The core components support the basic functionality for shapes and rules. The loosely coupled third component is responsible for the solution's output.

The core components are the *modeler*, and the *interpreter*. The first one is responsible for the abstraction and representation of the labeled shapes, and it is in this component where all shapes are created, modified and live. The second module is responsible for handling the rules, and is where rules are described, applied and controlled. It is this component responsibility for calling the functionality provided by the other components. The final component, the *exporter*, allows the generated solutions to be presented to the user by providing a communication with the CAD tool.

While changes in the first two modules change the functionality of the SG interpreter, e.g., changing the nature of its shape and rule representation, changes in the third component are merely local, yielding a interchangeable CAD tool to the SG interpreter. This happens because this component supports several CAD tools by the way of a common language, which abstracts the CAD tool and provides basic and common functionality of CAD tools. The SA described is illustrated in the figure 3.3.

We now describe how a user would use a SG interpreter application that implements this SA. The user starts by describing a set of rules and choosing a strategy that dictates how the rules are applied. These rules reside and are applied in the *interpreter*. Here, they are continuously applied to the current shapes using the chosen strategy. When rules are applied to shapes, they modify or create additional shapes. Shapes are created and modified by rules using the functionality of the *modeler*. This happens until a solution is found, or no more rules can be applied, representing a failure. When a solution is found, the *interpreter* instructs the *modeler* to begin the output of the solution. This is done using the *exporter*, with the *interpreter* invoking both its functionality to communicate the shapes directly to the CAD tool.

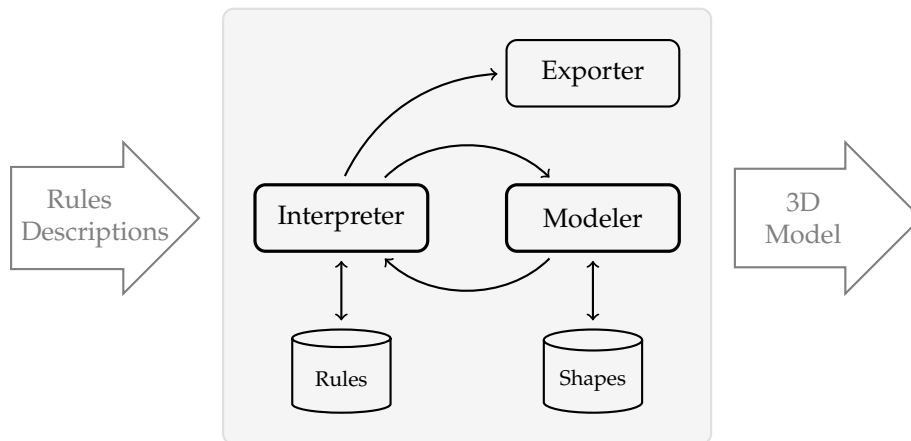


Figure 3.3: The proposed shape grammar software architecture with two core components, interpreter and modeler, and one loosely coupled component, exporter

4 Designa

I have not failed. I've just found 10,000 ways that won't work.

– Thomas A. Edison

Up until now, we explored key factors that a SG interpreter should provide for a successful application of the SG formalism. We also described some limitations, of which we identified three issues that are addressed as basic features that a SG interpreter should provide. These features are related with (1) shapes, (2) rules, and (3) a link to a CAD tool, and they express (i) how to represent, create and modified shapes, (ii) how to describe, apply and control rules, and, (iii) how to present a design solution.

Our approach to these issues was a proposal for a SG interpreter using a different SA, which combines: (1) a 3D modeler with a labeling mechanism, (2) describing rules as transition operators, (3) a search algorithm, and (4) a CAD tool for visualization and further manipulation of the generated shapes.

While the previous chapter outlined our proposal for a SG interpreter, in the following sections we describe in detail how we implement it. We name it DESIGNA.

First, we detail how labeled shapes are implemented. Then, we explain, how rules are described, applied and controlled, and how the solution designs are presented. Finally, we explain how all these components work together.

4.1 Shapes

Our approach to represent a labeled shape is based in a graph data structures representing the shape's boundary. In the following sections we describe in detail this approach to represent, create and modify labeled shapes.

4.1.1 Shape Representation

As stated in the previous chapter, our approach to represent shapes is based on the work of Heisserman (Heisserman, 1992). We represent shapes by its boundary using a graph data structure. This describes shapes in terms of facets, edges and vertices, and their relations. However, contrary to Heisserman's work, instead of developing our own 3D modeler, which takes time and resources, we use a third party library called CGAL. CGAL stands for Computational Geometry Algorithms Library and it is a library

for efficient and reliable computational geometry algorithms and data structures, that is being extensively developed for years in several universities and it is used in academia and in the industry. It also has an active and helpful community. Therefore, we use one of CGAL data structure for our representation of shapes, and CGAL functionality of that data structure to support the creation and modification of shapes.

Similarly to Heisserman, shapes are represented as a set of vertices, edges, facets, and incident relations between them. This means that, in both data structures, topology is represented as a graph where the nodes are topological elements and the arcs represent the adjacency relations between elements.

Geometry, described by three dimensional points, is associated to vertices. While there is a connection from a vertex to a point, the inverse does not exist.

To illustrate this graph representation, consider a tetrahedron. Its graph is presented in figure 4.1, where squares represent vertices, without the geometrical information associated, and, edges and facets are circles and diamonds, respectively. The dashed nodes and arcs represent the hidden elements of the tetrahedron, like if we saw the tetrahedron in perspective, as shown in figure 3.1a.

Heisserman developed the split-edge data structure to support his graph-based boundary representation of solids. This representation allowed Heisserman the specification of clauses for matching on conditions of solid models and the generation of modifications to those solids. Heisserman's data structure is a variation of the winged-edge data structure to represent polyhedrons, developed by Baumgart (Baumgart, 1972). With winged-edge structures, each edge has associated both vertices, facets and preceding and subsequent edges for both adjacent facets. This data structure allowed Baumgart to easily and efficiently answer many topological queries. For example, finding all the incident edges (or facets) of a vertex (Neperud, Lowther, & Shene, 2007).

This contrasts with traditional data structures, where vertices, edges and facets information are kept in tables or linked-lists, making the cost to answer topological inquiries too big, because they required too many comparisons and complete scans in the data structures. To avoid these limitations, Baumgart developed the winged-edge data structure, and Heisserman implemented the split-edge data structure to represent solids.




Our approach is similar to Heisserman's but uses the halfedge data structure provided by the CGAL library. This data structure represents how the vertices, edges and facets are composed and organized to represent polyhedral surfaces.¹

The halfedge data structure also represents shapes with a graph of its boundary representation, but represents a shape edge with two opposing halfedges, where each halfedge is associated with its incident vertex and facet, and also with its next, previous, and opposite halfedges.

For a better understanding how the winged-edge and the halfedge data structures related with each

¹In CGAL nomenclature a polyhedron represents a solid described by its boundary surface.

Legend:

-  Vertices
-  Edges
-  Facets

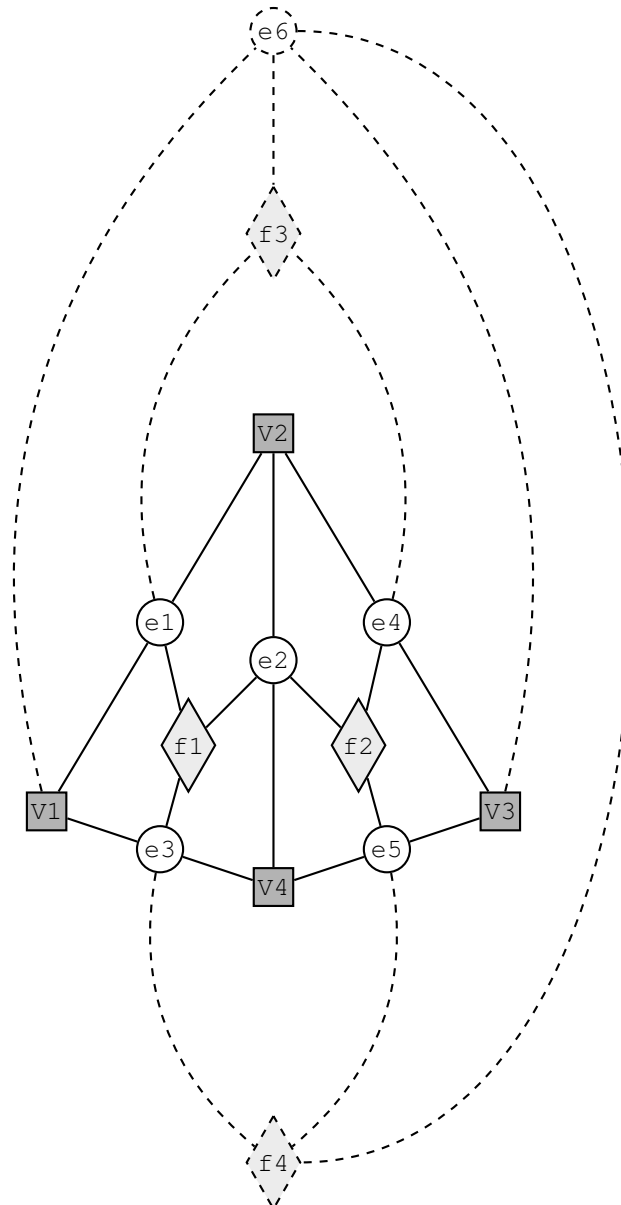
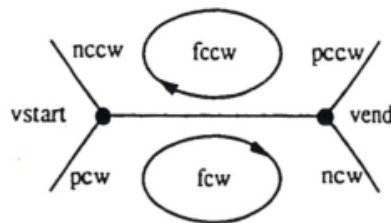
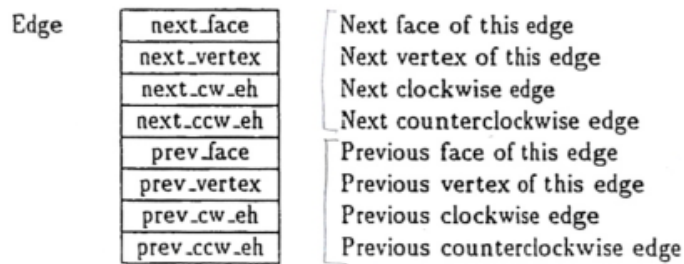
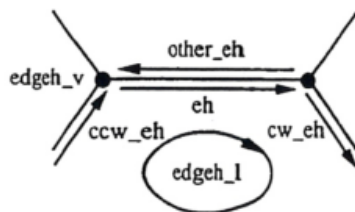
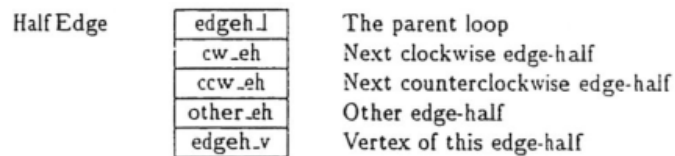


Figure 4.1: Graph representation of a tetrahedron - nodes represent topological elements, and arcs represent relations between elements (the dashed style represent hidden elements of the shape)

other, in figures 4.2a and 4.2b are illustrated the edge relations to the other elements of a shape, and its respective edge representation. In this figure are represented, for both data structures, the required fields to implement such data structure and it is schematic the relation of an edge with the other elements of a shape.



(a) Winged-edge data structure and respective relations



(b) Halfedge data structure and respective relations

Figure 4.2: Edge representation and edge relations in the winged-edge and halfedge data structures

With the halfedge data structure we solve three problems (Ghali, 2008): (1) each element of a shape has constant space requirements, (2) the list of adjacencies for each element can be reconstructed efficiently, and (3) traversing the neighborhood of an edge occurs always in the same orientation.

This results in similar features provided by the winged-edge data structure without sacrificing storage space and answering topological enquiries more efficiently, and allowing, also efficiently, to (Ghali, 2008): (1) traverse the list of vertices or the edges of a facet, (2) traverse the edges outgoing from or incoming to a vertex as well as the adjacent facets, and (3) traverse the faces adjacent to a given facet.

Unlike Heisserman, we use multiple representations for numbers and geometric calculations, allowing users to choose between different degrees of precision and speed. With the highest numerical precision, we avoid common problems associated with rounding errors. However, we always use exact queries, meaning that geometric tests are always correct. For example, checking if a point lays on one side or the other of a plane will not be affected by numerical imprecision. This means that numbers are tailored for speed, but all geometric queries sacrifice execution time and storage space over exactness.

4.1.2 Shape Generation

Similarly to Heisserman's work, we also create and modify shapes using operators that locally modify the boundary of the shape but preserving the Euler's equation 4.1, where V is the number of vertices, E is the number of edges, and F is the number of facets of a solid.

$$V - E + F = 2 \quad (4.1)$$

Therefore, the Euler operators are functions that operate on the halfedge data structure, altering it by adding or removing vertices, halfedges, and facets so that the Euler's equation 4.1 remains valid on the solid that is modified after the execution completes (Ghali, 2008).

These Euler operators were first introduced by Baumgart (Baumgart, 1975) for use on his winged-edge data structure. They manipulate the graph representation of the topological elements and adjacencies of a shape, providing an interface that ensures the construction of a valid topological representation of a solid, and at the same time avoiding invalid topologies, e.g., a facet with a hanging edge. Moreover, each of these euler operators can be executed efficiently (Ghali, 2008).

While Euler operators use the acronyms, m , k , v , e , f , standing for make, kill, vertex, edge and facet, respectively, we use the CGAL Euler operators, which provide the same effect but with a different interface. For example, the two operations *split_edge* and *join_edge* have the same effect as *mev* and *kev* operators, adding and removing one vertex and one edge, respectively.

Thus, using CGAL Euler operators, we can create or modify shapes, where the provided set of operators include, *split_edge*, *join_edge*, *split_facet*, *join_facet*, *create_center_vertex* and *erase_center_vertex*.

To illustrate what is happening in the halfedge data structure, we take two operators *split_facet* and *join_facet* as an example. The first takes two halfedges with the same incident facet, h and g , and splits

its incident facet in two facets, with a new edge between its incident vertices. The former, takes one halfedge h and removes its opposite halfedge facet. Visually this is illustrated in figure 4.3.

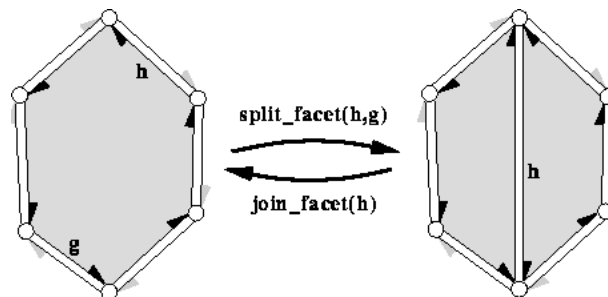
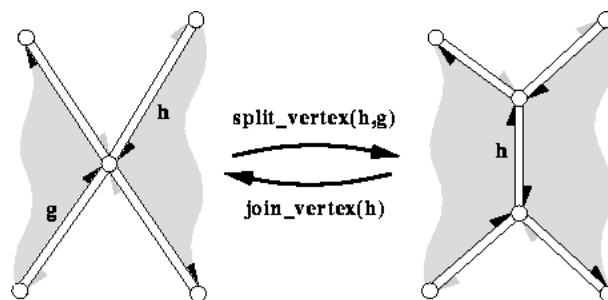
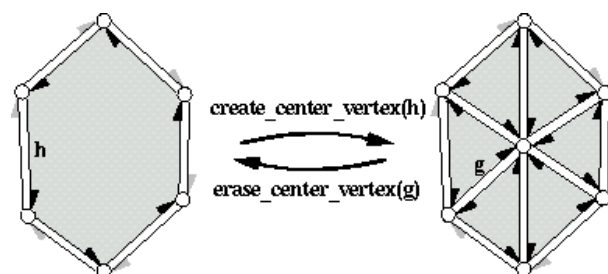


Figure 4.3: Euler operators, *split_facet* and *join_facet*, which splits a facet in two or joins two facets in one, respectively

In the figures 4.4a and 4.4b are illustrated two other sets of operators, *split_edge/join_edge*, and *create_center_vertex/erase_center_vertex*. The operator *split_edge* splits the vertex incident of the halfedges h and g into two vertices, connecting both with a new edge. The operator *join_edge* joins the incident vertex of the halfedge h and its opposite halfedge incident vertex, removing this last vertex. The operator *create_center_vertex* performs the barycentric triangulation of the facet incident of the halfedge h , creating as many new edges as the number of vertices of the facet. Finally, the operator *erase_center_vertex* reverses the previous operator, by removing the incident vertex of the halfedge g , removing all the edges and merging all incident facets.



(a) Euler operator to split and join a vertex



(b) Euler operator to create and erase a center vertex in a facet

Figure 4.4: Two other examples of Euler operators

To illustrate the different steps involved, and how the Euler operators are used to generate a shape, let us consider a unit cube as an example. Instead of starting with a single set of points and calling the standard Euler operators to make vertices, edges and facets, we use CGAL functionality to create incrementally the cube. Roughly, we start by (a) creating a tetrahedron, representing the smallest representable closed surface, and then we (b) split the necessary edges, creating three more vertices, (c) associate geometry to these new vertices, (d) create a new facet, (e) split another edge, creating the final vertex and associate the respective geometry, and finally (f) create the last facet, closing the surface into a cube. In figure 4.5 and algorithm 1 are shown the steps just described.

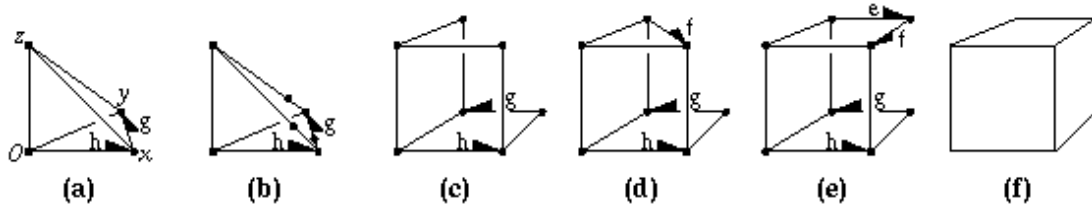


Figure 4.5: Example of a unit cube creation using Euler operators

Algorithm 1 Create a unit cube using CGAL tetrahedron and operators

```

P ← new_polyhedron()
h ← P.make_tetrahedron(new_point(1 0 0), new_point(0 0 1), new_point(0 0 0), new_point(0 1 0))
g ← h.next().opposite().next() ▷ (a)
P.split_edge(h.next())
P.split_edge(g.next())
P.split_edge(g) ▷ (b)
h.next().vertex().point() ← new_point(1, 0, 1)
g.next().vertex().point() ← new_point(0, 1, 1)
g.opposite().vertex().point() ← new_point(1, 1, 0) ▷ (c)
f ← P.split_facet(g.next(), g.next().next().next()) ▷ (d)
e ← P.split_edge(f)
e.vertex().point() ← new_point(1, 1, 1) ▷ (e)
P.split_facet(e, f.next().next()) ▷ (f)

```

4.1.3 Labels

In general, labels are used to distinguish elements of a shape by associating non-geometric data with any topological elements of shapes. Labels can also be used to restrict rule applications by imposing specific conditions on the shape generation or modification. Take note that this is different from defining how to control the application of rules or even defining its order.

Labels are implemented using a numerical identifier for each topological element of a shape, along with a hashtable, creating the labelling mechanism as a generic named property, where the name rep-

resents the label and the value contains all the shape elements that were associated with the label. The functionality provided includes functions to compare, assign, query and remove labels. These are automatically generated for each label required, and the user only needs to invoke the function *new-prop*.

As an example, consider that a user wants a generic *id* label for labelling any element of a shape with an integer. To accomplish this, the user only needs to invoke *new-prop(id)* and the interpreter would generate *id-eq?*, *id!*, *has-id?*, and *remove-id!* functions, which, respectively, compares two *id* labels, sets an *id* to any of the shape elements, inquiries if two *id* labels are the same, queries the shape elements if they contain the *id* label, and removes the *id* label from any shape element.

4.2 Rules

With limited options available to apply rules, current SG interpreters restrict the use of the SG formalism. This happens because they only provide users the functionality to apply rules manually, or usually rely in the user to dictate the order in which rules are individually applied. With such limitations, the user ends up using mechanisms that were not created for that purpose, e.g. the salience property of rule engines or even SG labels. Moreover, SG rules usually condense too much information in one rule. For the human reader this is an advantage, but for a SG interpreter, that do not have the common sense of humans, this way of describing rules do not work well. Another problem noticed, in SG in general, is the explosion of design solutions that can be generated once the process is started. With parametric SGs the problem gets worst, because if parameters are allowed to have a large variation, like with real numbers, the number of solutions generated grows to infinity.

To overcome these limitations, we take a different approach. We see the language generated by the SG as a state-space. That is, each state encodes a particular design, and each SG rule is encoded as a transition operator, moving from one state to another. Thus, the application of the SG to a given design can then be seen as a search, in the state-space, for a path that connects the initial state to some goal state. This path encodes the sequence of SG rules that, starting from that initial design, leads to the final design. In general, the specification of the initial state is direct, as there is usually an initial design available. However, the specification of the goal state is much more abstract as, in general, we know some of the properties that it must possess, but not the actual shape. This means that, in practice, the goal state is described by a predicate that is true only when all the properties of the intended final design are satisfied. In this case it is said that a solution was found.

There are several different strategies to find a path in the state-space which connects the initial and final states. In order to understand these strategies, it is important to realize that, in most cases, it is impossible to actually generate the entire state-space because the application of SG rules is a combinatorial problem with an enormous number of possibilities. In fact, for the majority of problems, the number of states grows exponentially with the number of transitions and, due to memory limitations, this means

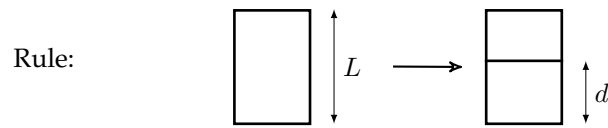
that searching the space-state must be done incrementally, by generating the space-state as the search proceeds. In order to do this, the transition operator becomes a generator: its application to a given state generates the “next” state, that is, the state that represents the design which results from the application of the corresponding SG rule. The application of all possible transition operators to a given state is then called the expansion of that state. In general, all search strategies are based on the recursive expansion of states, from the initial to the goal state, and the order in which the states are expanded determines the search strategy. However, when the goal state is reached, it might be necessary to know which path was followed. To this end, each state is enriched with additional information, namely, which state and which transition operator were used to generate it. This enriched state is called a node and the exploration of the state-space entails the corresponding enlargement of the graph connecting these nodes. In this graph, the edges correspond to the transition operators (i.e., SG rules) that were applied to a node to produce its descendants. Expanding a node entails expanding the corresponding state followed by the creation of a node for each generated state.

4.2.1 Rules Description

For this approach to work, as previous presented, a SG rule is encoded as a transition operator. As in the SG rule, the corresponded operator also has two parts, known as antecedent and consequent. The antecedent describes the design to which the operator applies. The consequent describes the design that results from the application of the operator. In practice, to minimize the number of operators that must be written, each operator uses, instead of a consequent, a set of consequents, i. e., a set of possible designs. Given that this set is computed during the search of the state-space, it is possible for this set to be empty, meaning that the specific operator could not make the transition from a given state. Other operators, however, might be able to compute such transitions.

To better illustrate our approach for describing rules, let us consider first a simple SG rule. The rule states that, if in a given design there is a shape that resembles a rectangle, then this shape can be transformed into a different design where the rectangle is divided in two parts. The division occurs parallel to the lower edge at values of one third, one half, or two-thirds of the original length of the rectangle. This means that the antecedent of the rule describes how to identify a shape in a design, a shape of four vertices and four edges, which, in sets of two are opposite, parallel and congruent. The consequent describes how the identified shape is to be modified. In this case, it describes the introduction of two new vertices on the longest edges, connected by new edge that is parallel to the lower edge. Visually, the rule, is presented in figure 4.6.

For a human reader, this rule is well described. Mainly because it condenses three possible outcomes of the rule application in just one. In other words, when a human reader sees this rule, he immediately understands that if a rectangle is found in a design, it will generate another design with one



where d can be : $\frac{1}{3}L$ or $\frac{1}{2}L$ or $\frac{2}{3}L$

Figure 4.6: Example of a shape grammar rule - if there is a rectangle in a design, then divide it in two by d which can be one third, one half, or two-thirds of the original length.

of the three possible possible outcomes. But for a SG interpreter implemented in current computer systems, this way of describing rules is ambiguous and needs a better and more meaningful way to describe rules.

Therefore we describe such rule as an operator, where the antecedent of the operator also recognizes a rectangle as the antecedent of the SG rule does, and the consequent returns a set of three rectangles. Visually, the operator, is shown in figure 4.7.

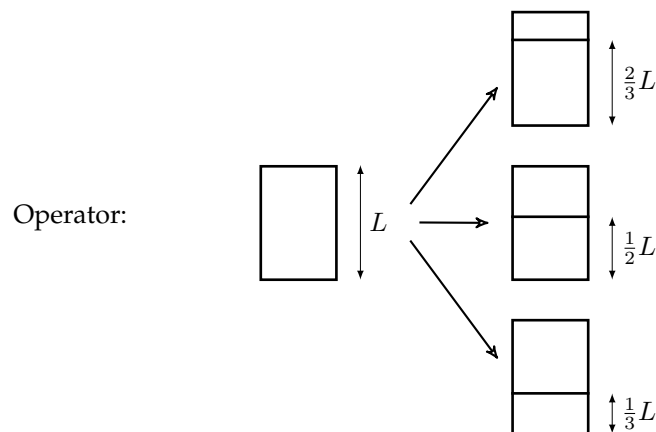


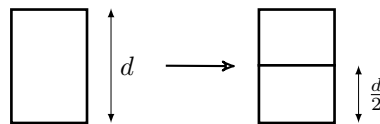
Figure 4.7: Example of an operator, based in the shape grammar rule of figure 4.6 - if there is a rectangle in a design, then generate three possible outcomes that divide the original rectangle in two by one third, one half, and two-thirds of the original length.

4.2.2 Rule Application and Control

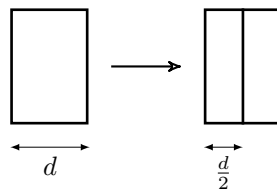
Has previously presented, the continuous application of rules, described as transitional operators, to shapes, encoded as a state, generate a state-space where each state is connected to another. This can be represented as a tree, where the nodes represent states (which encode a design) and the arcs represent operators (which encode a rule). Because there is a combinatorial explosion of states, this usually means that we cannot keep in memory all the states. Thus, a strategy is needed to generate the states incrementally and to generate as few as possible.

Given that different search strategies can be used and that their success (or failure) depends on the specific problem one is trying to solve, our approach does not force a particular search strategy and, instead, it provides three different ones: (1) depth-first, (2) breadth-first, and (3) A*. (1) Depth-first search explores a complete path of the state-space graph before exploring another path. (2) Breadth-first search explores all paths with length n before advancing to length $n + 1$. And, (3) A* search ranks paths according to the estimated cost from the initial state to the final state and explores the one with lowest cost.

To illustrate and allow a better understanding of how the depth and breadth first works, let us consider a SG as example. The initial shape is a rectangle and the only two rules state that, if a rectangle is found in a design, divide it horizontally and vertically. These rules are shown in figures 4.8a and 4.8b, respectively, and curiously, in this case, both the SG rules and the operators can be described in the same manner.



(a) Rule 1: If a rectangle is found, divide it horizontally at $\frac{d}{2}$

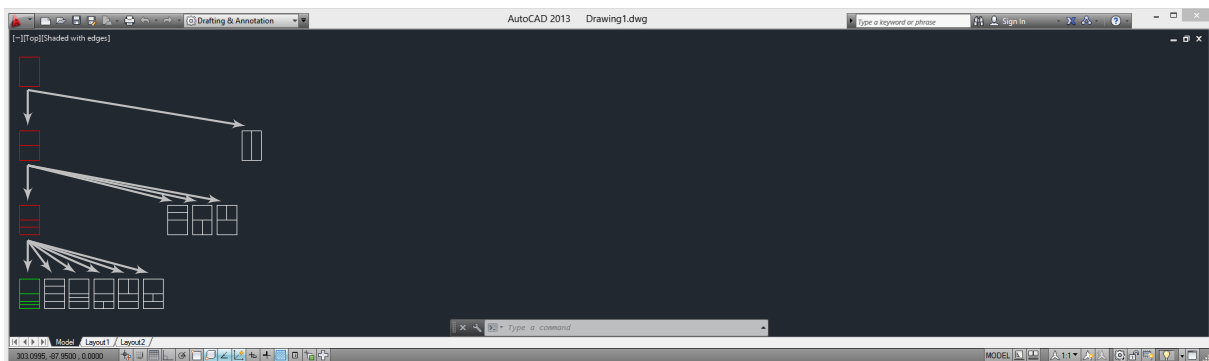


(b) Rule 2: If a rectangle is found, divide it vertically at $\frac{d}{2}$

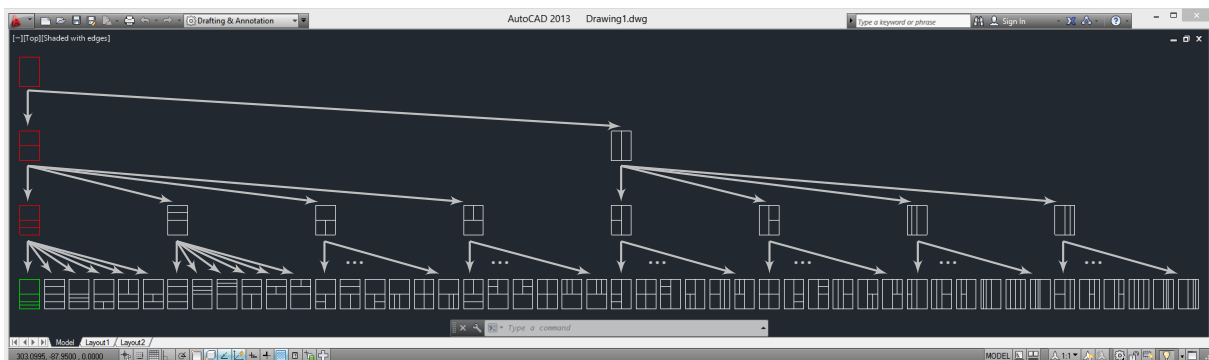
Figure 4.8: Example of rules/operators, in which a rectangle is divide in half, horizontally or vertically.

The search for a solution starts by applying both the operators to the initial rectangle, generating two possible designs (two rectangles each divided horizontally and vertically). Hereafter the operators are again applied to the two generated rectangles, generating yet more designs, and so on. Depending on the strategy used, and the problem to be solved, the resulting tree can vary greatly but for illustrative purposes, and using the SG just described and the depth and breadth first strategies, the resulting trees are shown in figures 4.9a and 4.9b, respectively. The red designs represent the path from the intial shape to the solution, in green.

For any of the strategies to work, and because many designs can be considered simultaneously, the strategies need to copy the current state so that the consequent of a rule can modify it safely, and thus making progress to a solution. To make this copy of a state, which encodes a design that is composed of shapes, we save all the function invocations along with their arguments that generated and labeled the shape until that time. This information is kept in the special label *history* in each shape, and when



(a) Results of depth first strategy using rules of figure 4.8



(b) Results of breadth first strategy using rules of figure 4.8

Figure 4.9: Search trees, depth first and breadth first - red designs represent the path to the solution, in green

the strategy needs to create a copy of a shape, instead of traversing all the graphs shape and copy it, it invokes all the functions saved in this *history* label. The main advantage of this approach is that is very easy to implement and fast, compared to the copy of the graph of a shape. The only, minor disadvantage is that it requires one additional line of code in each function that creates and manipulates labeled shapes, to save the current function invocation.

4.3 Output

Currently, SG interpreters adopt one of two possible ways to present the generated designs. One is to complement the SG interpreter with a dedicated visualizer, which typically is an extremely limited version of the CAD tools used by the Architects. The other is to present the design in some widely used CAD tool by implementing the entire SG interpreter inside that CAD tool. Unfortunately, in this last case, the SG interpreter tends to be as limited as the programming languages and frameworks provided by that CAD tool. Another consequence of such approaches is that the SG interpreter cannot easily be adapted to work with a different visualizer or with a different CAD tool.

In order to overcome these problems, we decided to use Rosetta (Lopes & Leitão, 2011), a tool for generative design that combines expressive programming languages and frameworks with the most used CAD tools. One major advantage over all other approaches is that we make the use of SGs portable across as many CAD tools as the Rosetta environment supports. Another advantage is that the design solution is able to be worked in the CAD tool or even to be saved into any other format the CAD tool provides.

Therefore, for our system, Rosetta provides an abstraction layer which allows us to communicate directly with the CAD tool. While this abstraction layer supports common CAD functionality like layers, points, lines and surfaces, it does not directly provides the functionality to support SG shapes. To overcome this limitation we implemented the traverse of the graph representation of shapes and by invoking the Rosetta functionality we draw shapes as two-dimensional polygons or as three-dimensional surfaces.

4.4 Designa

To validate our thesis statement, DESIGNA, a SG interpreter, was implemented. Using the proposed SA, DESIGNA overcomes some limitations of current SG interpreters focusing on labeled shapes, rules and an output of the solution designs directly to a CAD tool.

In figure 4.10 is shown the overall architecture of DESIGNA.

The architecture interconnects three main components: CGAL, the library that supports the halfedge

representation (written in the C++ programming language), Rosetta, where our SG interpreter is implemented (written in the Racket programming language), and finally, the CAD applications.

To make the C++ halfedge data structure available in DrRacket, the programming environment where Rosetta is implemented, a wrapper was developed. This wrapper, defined as foreign functions in the DrRacket, provides all the functionality to create and manipulate the halfedge representation from Rosetta.

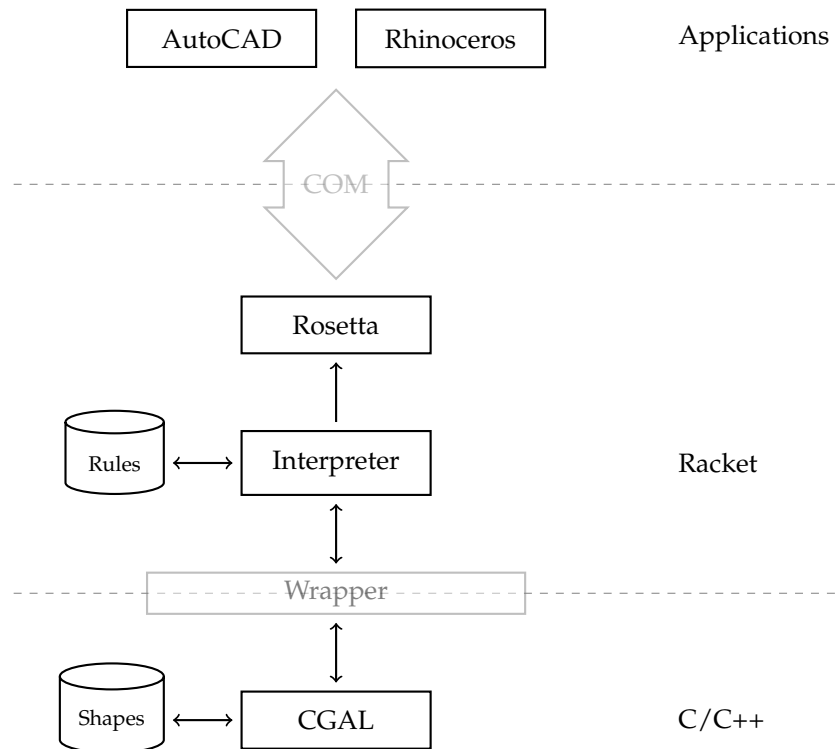
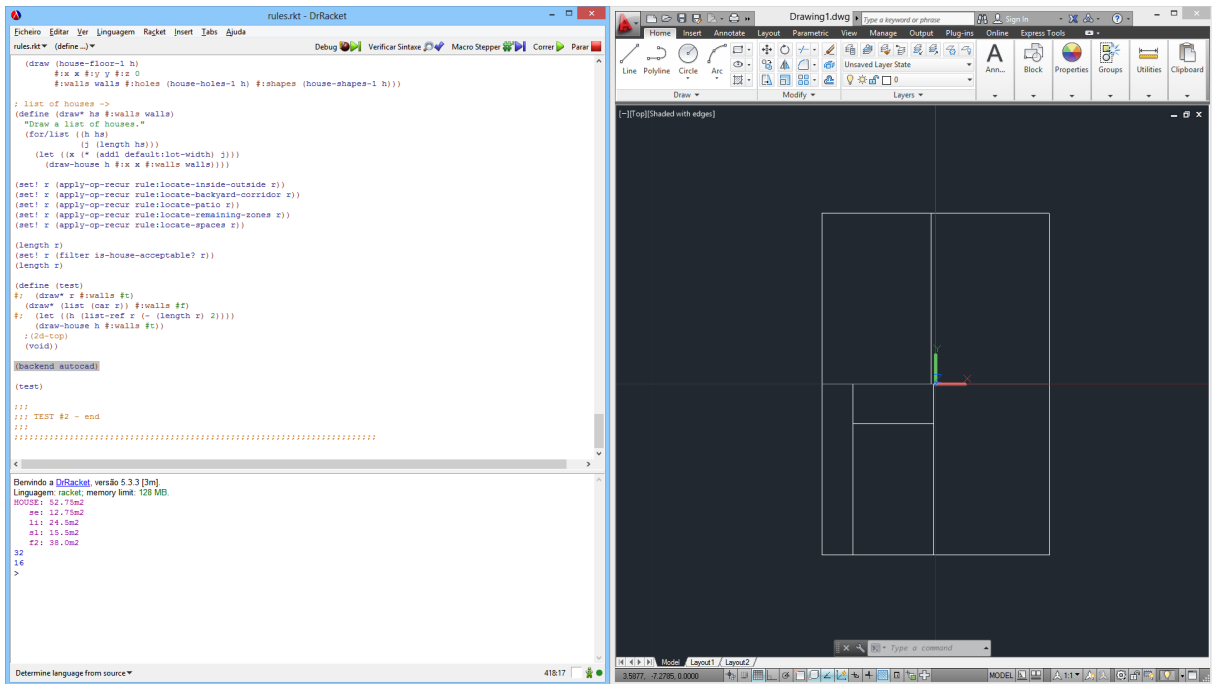


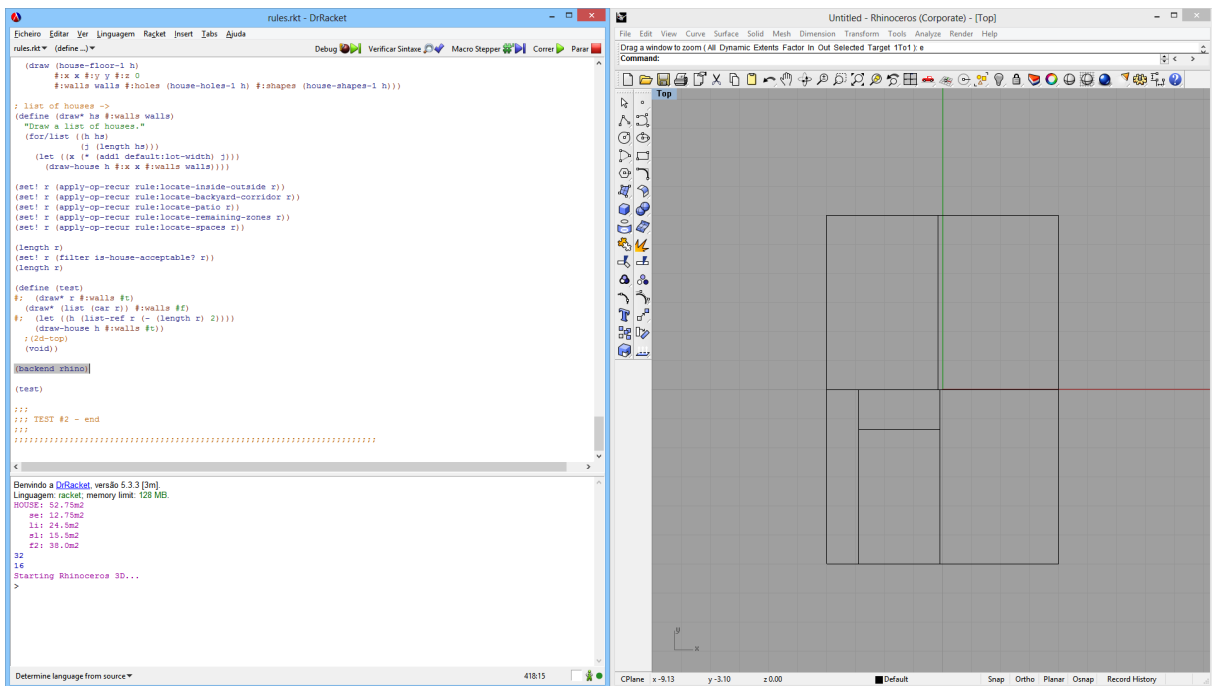
Figure 4.10: General overview of the system architecture proposed

As stated previously, DESIGNNA is a SG interpreter. This means that the rules, are continuously applied to shapes until a solution is found. The rules encode a design brief (DB), and shapes created represent a house. A DB represents all the user preferences along with site data, and describes a housing in terms of required and optional spaces. It also details the spaces in terms of areas (expressed in square meters) and the relations among each other (e.g. the pantry should be included in the kitchen). The rules also reflect the regulations and restrictions representing the style captured by the SG implemented.

In the figure 4.11 are shown the overall aspect of the DESIGNNA interpreter, using AutoCAD (figure 4.11a) and Rhinoceros (figure 4.11b) as the CAD tools.



(a) DESIGNA with output in Autocad



(b) DESIGNA with output in Rhinoceros.

Figure 4.11: DESIGNA showing shape grammar rules and corresponding output in AutoCAD and Rhinoceros

5 Evaluation

There are no facts, only interpretations..

– Friedrich Nietzsche

In this chapter we present the results obtained with the implementation of three SGs using the system developed within this work. The main focus of the evaluation of the system is the implementation of a SG for the mass customization of housing, the Malagueira SG. However, we will start by describing the implementation of a much simpler grammar that generates ice-rays, as a demonstration of the features of DESIGNA, and presenting a three-dimensional SG, to show, through 3D shapes, the results of the search strategies implemented.

In all these implementations four questions always arise: (1) would we describe a shape in two or three dimensions?, (2) how do we describe the rules, (3) how do we apply rules, using a custom strategy or use one provided with the interpreter?, and finally (4) do we present the design solutions with solids, lines or polygons? Our answers to these questions for the three SGs are presented next.

5.1 *Ice-Ray Shape Grammar*

Ice-ray grammars were first formalized by Stiny (Stiny, 1977), using a parametric SG as a means to describe the design of Chinese lattices. These designs were used mainly as ornamental windows in China between 1000 BC and 1900 AD. Two examples of such designs are shown in figure 5.1.

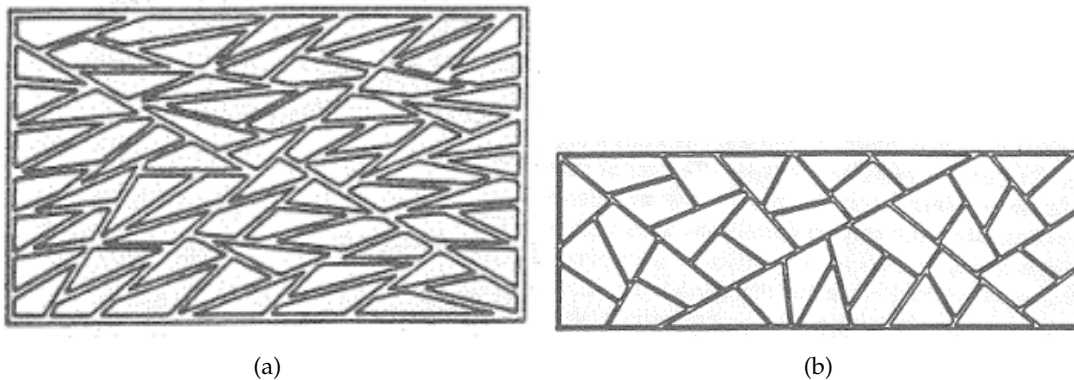


Figure 5.1: Two example of Chinese ice-ray designs used in ornamental windows

Stiny’s ice-ray SG defined five parametric rules and a set of conditions, which restrict the application of the rules. For a better understanding of what the rules describe, they are quoted next (Stiny, 1977).

“The first shape rule states that any triangle with area greater than some given constant may be augmented once by placing a line between any two of its edges to form another triangle and a quadrilateral with approximately equal areas. (...) The second and third shape rules state that any convex quadrilateral with area greater than some given constant can be augmented once by (a) placing a line between any two of its adjacent edges to form a triangle and a convex pentagon with approximately equal areas or (b) placing a line between any two of its nonadjacent edges to form two additional convex quadrilaterals with approximately equal areas. The fourth shape rule states that any convex pentagon with area greater than some given constant can be augmented once by placing a line between any two of its nonadjacent edges to form a convex quadrilateral and another convex pentagon with approximately equal areas. The fifth shape rule in this SG allows for the symbol \bullet to be erased, and hence for the termination of the shape generation process. (...)”

In other words, the first four rules describe the subdivision of shapes that resemble (1) triangles, (2) quadrilaterals, and (3) pentagonals, and the last rule define the condition to stop the generation of designs with the removal of the label \bullet of a shape. The original five rules, as defined in (Stiny, 1977), are shown for illustrative purposes in figure 5.2.

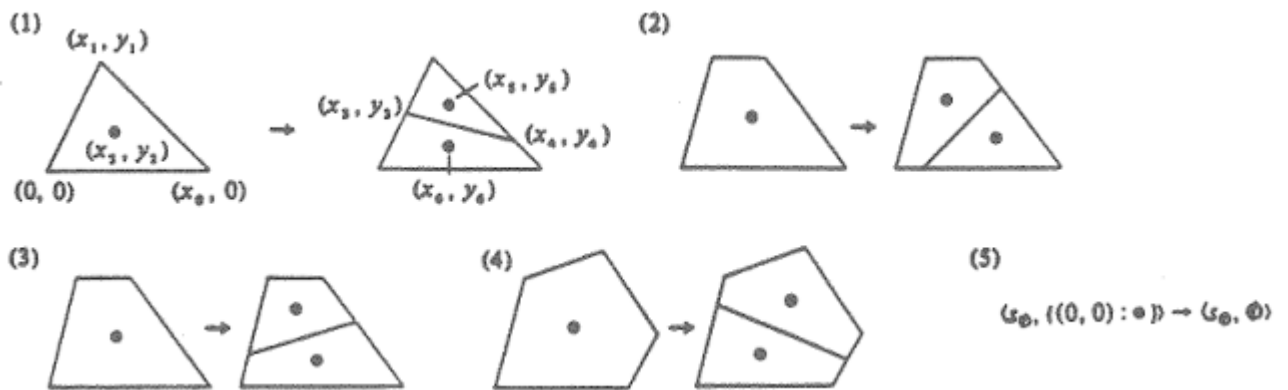


Figure 5.2: Stiny’s original ice-ray shape grammar rules

To implement this SG, we represent a design as one 2D polyhedron, where shapes are described by the polyhedron facets, representing triangles, quadrilaterals or pentagonal shapes. Therefore, and considering the context to which these rules apply is limited to triangles and convex quadrilaterals and pentagonals, the antecedent of the rules only have to identify facets in a polyhedron with the given required number of vertices, three for triangles, four for quadrilaterals and five for pentagonals shapes. And where the consequent of the rules only have to describe a set of all the possible placement of new lines, which is done by describing the Euler operator for splitting a facet in two.

This approach simplifies the required transformations needed to apply rules, as initially state by Stiny when he described the SG formalism. This happens because our transformations to shapes, the application of the Euler operators, are made locally.

Given that the initial shape for this SG is always a rectangle, which restricts the application of the SG to rectangular windows, we go further and describe only one rule which could be applied to any shape. Therefore and using the representation of shapes just described, the antecedent of our rule only has to identify a facet (a shape) in a polyhedron (a design), and the consequent returns a set of polyhedrons, in which the facet identified is divided in two at a prescribed value (recall figure 4.7 where it was explained how the rules described as operators which return a set of possible designs). This becomes even more flexible because we write the consequent of the rule to split the facet in any permutation of its edges. To help understand what this can generate consider a triangle, a cube and two other shapes, one with four vertices and the other with five vertices. These initial shapes are presented in the left column of the figure 5.3. The rest of the columns in the figure represent the set that results when the rule is applied to the shape of the first column. Note that, in the figure is only shown one possible value for the parameter of the rule, representing where in an pair of edges the division will occur. Therefore we can use the permutation described with any set of values where the split should occur and therefore create an infinite number of possible solutions.

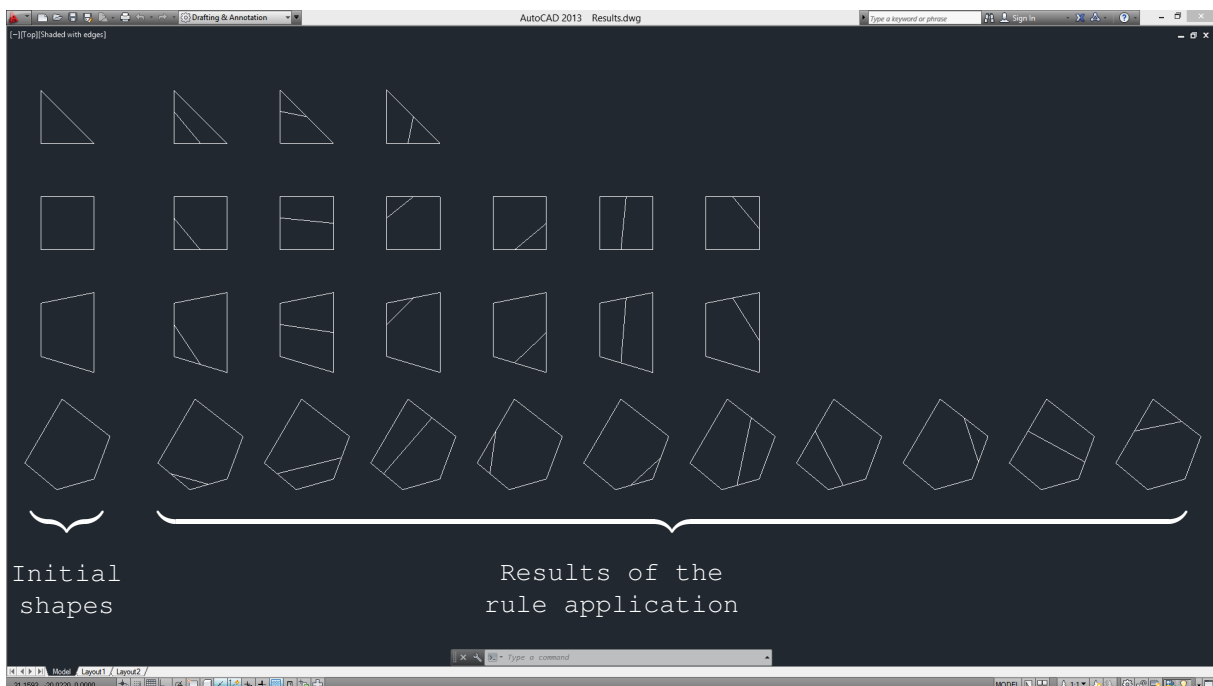


Figure 5.3: Result obtained from the implemented ice-ray rule using as the initial shape the shapes represented in the first column

Unfortunately, this representation poses a problem. Most of the times, a facet will be composed by two or more collinear edges. This actually results from successive splits of the facets in two. Take for

example the case illustrated in figure 5.4. The design illustrated results from the splitting of an initial rectangle, first, from A to C and then, from B to D . Here, even if the human reader sees the edge that goes from vertex A to C as the right edge of the gray facet, in fact with our approach to shape representation, the system will see two halfedges, one from A to B and the other from B to C . This means that for the system to split the facet in gray along the edge AC , it needs to take into account that this right edge of the facet is composed by two halfedges, and not just one. We deal with such cases with the creation of an abstracted operator that splits a facet by taking into account its collinear edges. This was implemented with the *split_edge* Euler operator that takes into account the collinear edges, which are identified by navigating through all halfedges of a facet and checking if they are collinear and split them accordingly.

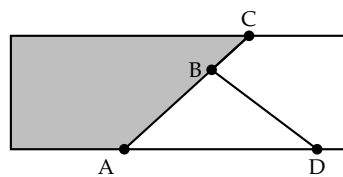


Figure 5.4: Example of a facet (in gray) that is composed by collinear edges

To illustrate a derivation of a design of an ice-ray using the SG implemented, in the figure 5.5 is presented the initial steps to generate a design. In this example, the rule is applied manually and only using specific values to where new lines are introduced. This replicates what a user of the SG would do if he/she was applying the rule by hand. The result mimics the design of an ice-ray very similar to Stiny's original generated design.

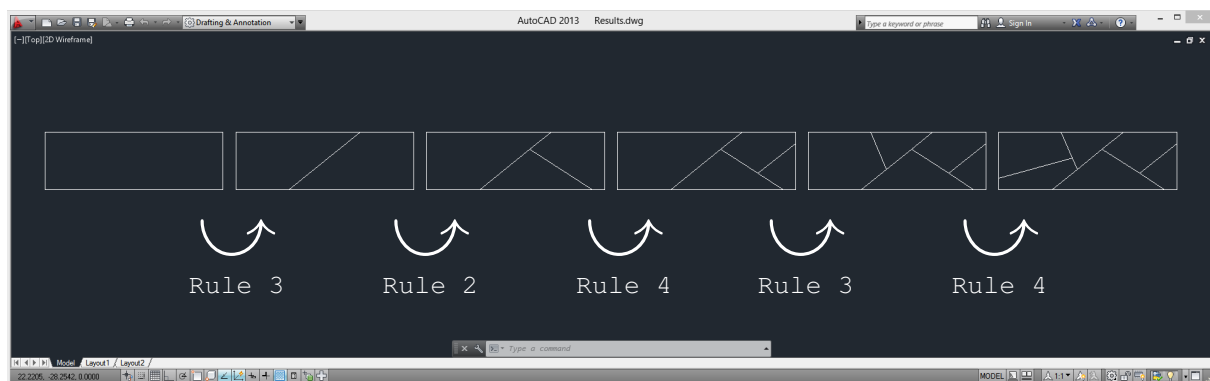


Figure 5.5: Derivation example, showed in AutoCAD, using the implemented Ice-Ray shape grammar

To show the exponential explosion of solutions that this simple SG can generate, in figure 5.6 are shown all the possible design solutions generated by the consecutive application of the rule three times to the initial shape, a rectangle. For this example the area of the facet was not taking in account, and as the figure illustrates, there are many solutions, and, of these 258 possible designs only a handful of them can be considered to be a good solution.

Figure 5.7 shows the initial shape and some results of designs that were generated using the ice-ray

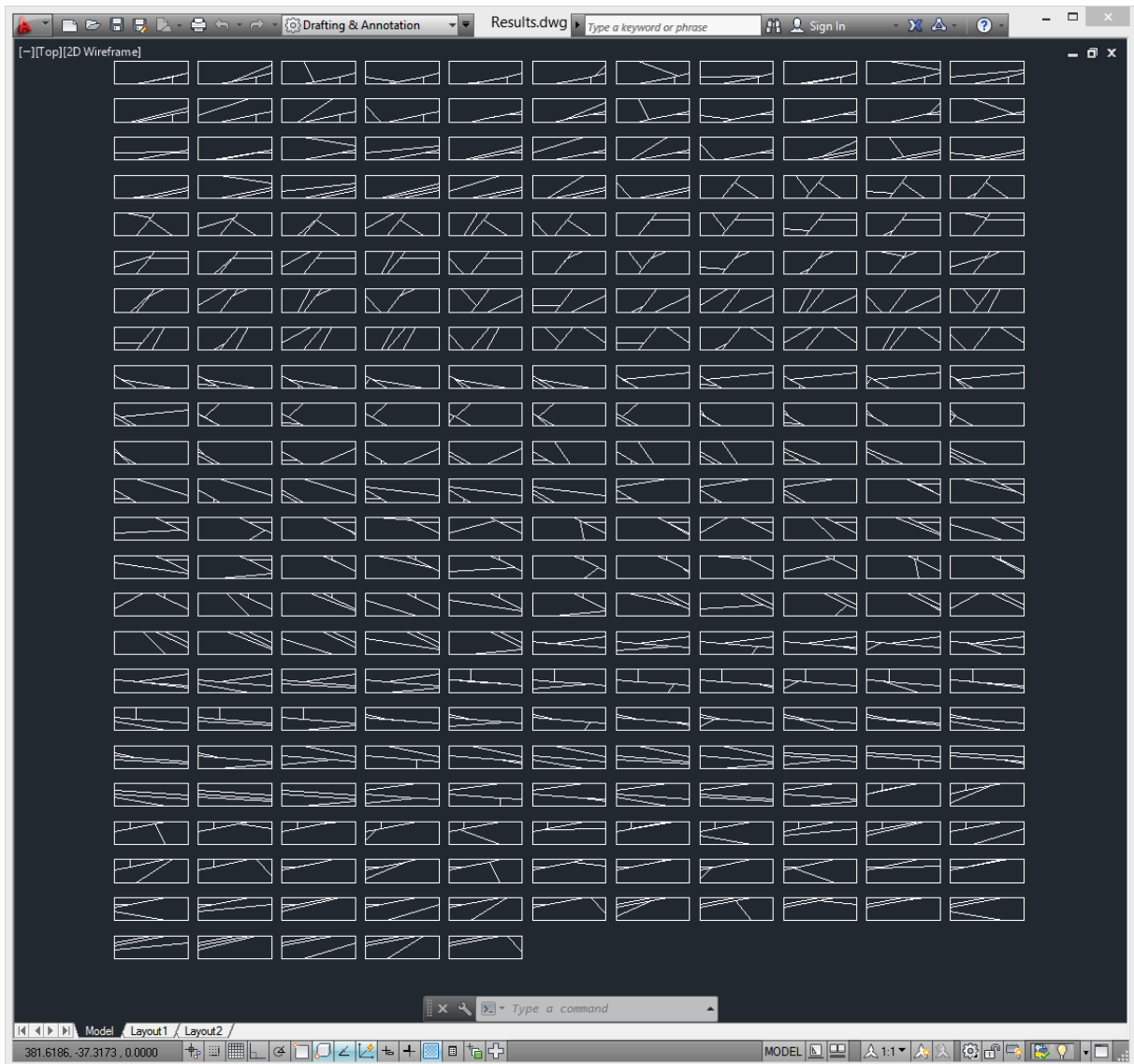


Figure 5.6: Possible designs, showed in AutoCAD, using the single rule implemented for the ice-ray grammar - only presented three possible combinations of splitting values and edges

grammar and a breadth first search without taking into account the area of the facets.

Finally, table 5.1 summarizes the impact of using a simple condition, e.g. the area of the facet that is to be divided, to restrict the application of rules. As shown, when using the specific restriction, there is a dramatic reduction of the generated designs using the breath search.

Table 5.1: Results of depth and breadth first search for the ice-ray grammar with additional time to draw the results in both the AutoCAD and Rhinoceros applications - Its show the number of nodes generated and expanded, and the time is measure in seconds. The * represents that the Rhinoceros application was minimized to avoid the constant refresh and in order to improve its performance.

	No area restriction		With area restriction	
	Depth	Breadth	Depth	Breadth
Expanded:	4	59	4	43
Generated:	20	442	20	198
Search time:	0.008	0.257	0.009	0.126
AutoCAD time:	0.417	8.052	0.321	3.578
Rhinoceros time*:	0.823	24.179	0.879	10.383

5.2 Three-Dimensional Shape Grammar

The second SG implemented fulfilled two objectives: (1) to show that our approach to represent a shape can handle 2D shapes as well as 3D shapes, and (2) to show what are the consequences of using depth and breadth first strategies in the final solution design.

For this grammar we wanted to capture a simple idea, from any shape facets we grab its center point and pull it out in the direction of the normal of the facet, generating as many new facets as the number of edges of the original facet, all converging in one point.

We implemented this grammar with only one rule, where the designs are polyhedrons, and its facets represent the shapes of the grammar. The rule states that, for any shape found in a design, in this case, a facet and a polyhedron, respectively, transform the shape into a pyramid with the apex at some perpendicular distance C , from the centroid of the shape, the symbol \bullet . This rule can be illustrated with the help of two shapes, a tetrahedron and a cube, as shown in figures 5.8a and 5.8b. Note that the symbol \bullet does not represent a label, but is only shown for illustrative purposes.

Using our shape representation we can apply this SG to different initial shapes and without modifications of the shape rule, designs can still be generated, as illustrated in figure 5.9, which also illustrates the consequences in the design solution when using different strategies. This works because shapes are represented as a set of vertices, edges and facets, and the relations between them, which allows the navigation and update of the shape representation in a general way.

In figure 5.10 and 5.11 are shown the results of using the rule of the three dimensional grammar

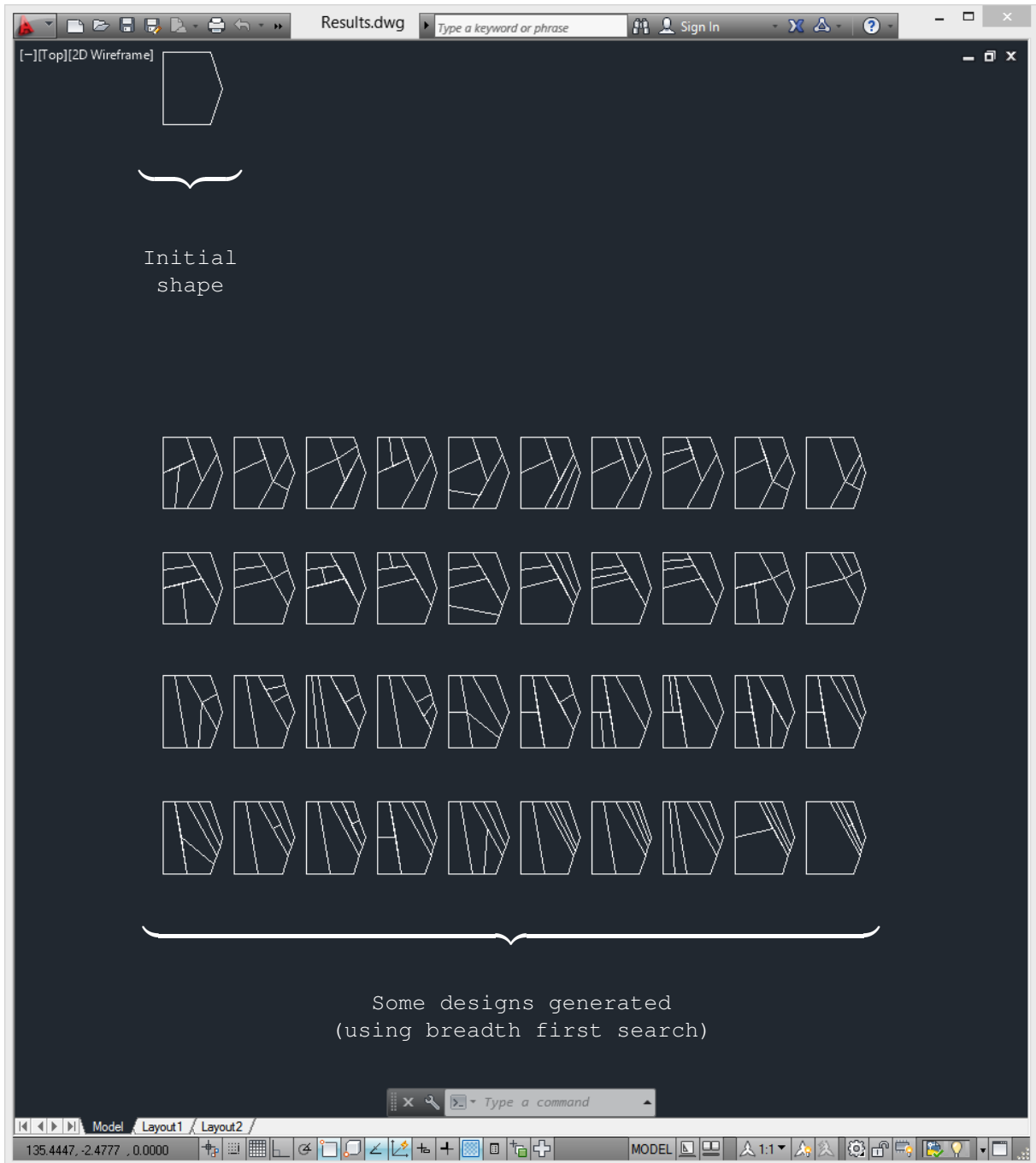
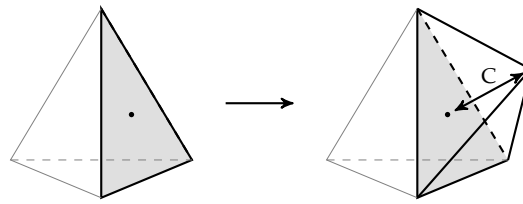
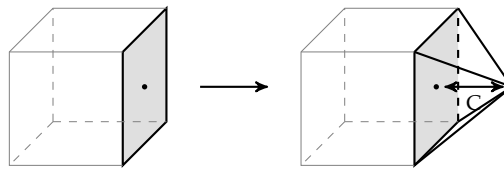


Figure 5.7: The initial shape used for the results shown in table 5.1, and some designs generated when using the breadth first search without the area restriction

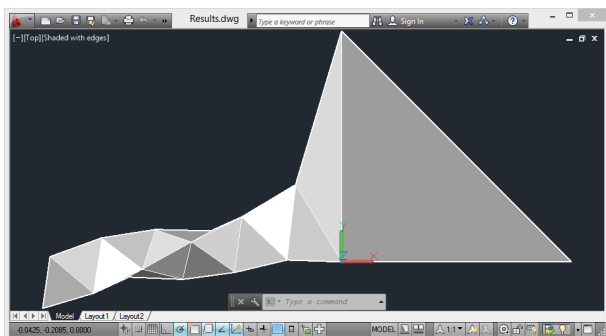


(a) Three-dimensional shape grammar rule, example with a tetrahedron

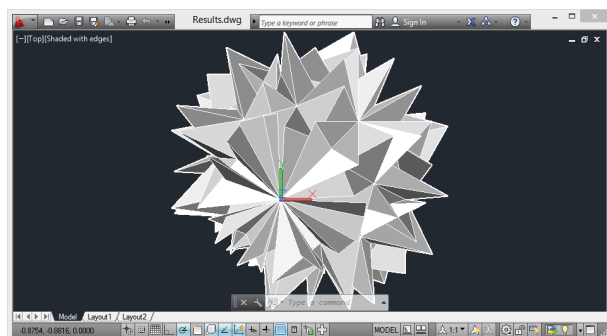


(b) Three-dimensional shape grammar rule, example with a cube

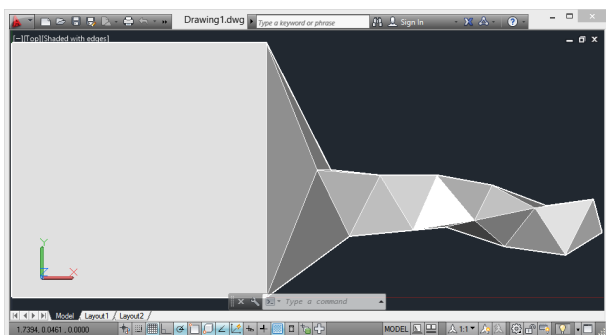
Figure 5.8: Three-dimensional shape grammar rule - transform a facet into a pyramid



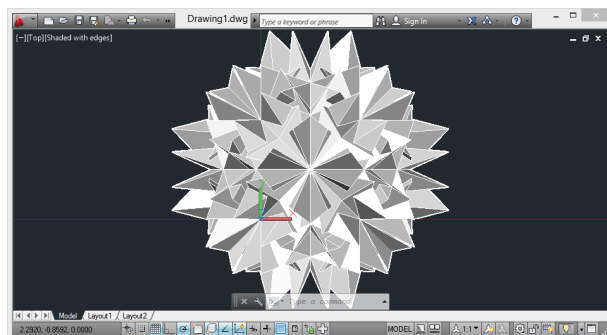
(a) A tetrahedron after applying depth first search



(b) A tetrahedron after applying breadth first search



(c) A cube after applying depth first search



(d) A cube after applying breadth first search

Figure 5.9: Results of three-dimensional shape grammar example using different search strategies

with different values to the rule parameter C . In both figures each row uses a different set for the values of the parameter C , while the columns represent the results after the application of the rule to all the facets of previous shape.

5.3 *Malagueira Shape Grammar*

The Malagueira SG is the third and final SG implemented to evaluate DESIGNA. Before entering the details of the implementation, we do a brief overview to help understand the basic concepts used hereafter.

5.3.1 Prologue

The Malagueira SG is the designing grammar used for experimentation and evaluation of a discursive grammar, as described previously (see chapter 2). It is a parametric SG and it is concerned with the generation of a housing solution that matches a design brief (DB). This SG encodes the rules laid out by the Architect Álvaro Siza for the design of the Malagueira houses, in Évora, Portugal.

It is an analytical SG, because it was inferred from a set of existing designs, representing the language or style. But, because it can also be used to generate new houses in the language, although it is not a full grammar developed from scratch to generate entirely new designs, the grammar is more than a mere analytical. Quoting the author (Duarte, 2001):

“It is reasonable to consider that it spans between analytical and original grammars.”

Briefly, the Malagueira SG was developed by Duarte (Duarte, 2001) to generate design solutions for the mass customization of housing. These solutions, represented as houses, are described by a layout, which is composed by spaces, organized in functional zones. A layout also describes the topology of a house and allows the answering of questions like: “Is the living room near the kitchen?”. In the Malagueira houses a layout is organized in four functional zones: patio, living, service and sleeping, and the set of spaces include: kitchen, living-room, bedroom, bathroom, closer, laundry, pantry, circulation, stairs, transitional space, and terrace.

Like in many large developments, an individual house is never alone, but clustered together in a housing block, where the position in the block where the house is located is called a lot. In the Malagueira project, houses are clustered in rows of two, which results that each house is always surrounded by neighbors, and where the front of the lot is always facing the street. For each house, this is called its urban context, and in Malagueira a house can have four possible urban contexts: (a) street only at the front, (b) street at the front and on the side, (c) street at the front and back, and (d) street at the front and back, and on the side. This is illustrated in figure 5.12.

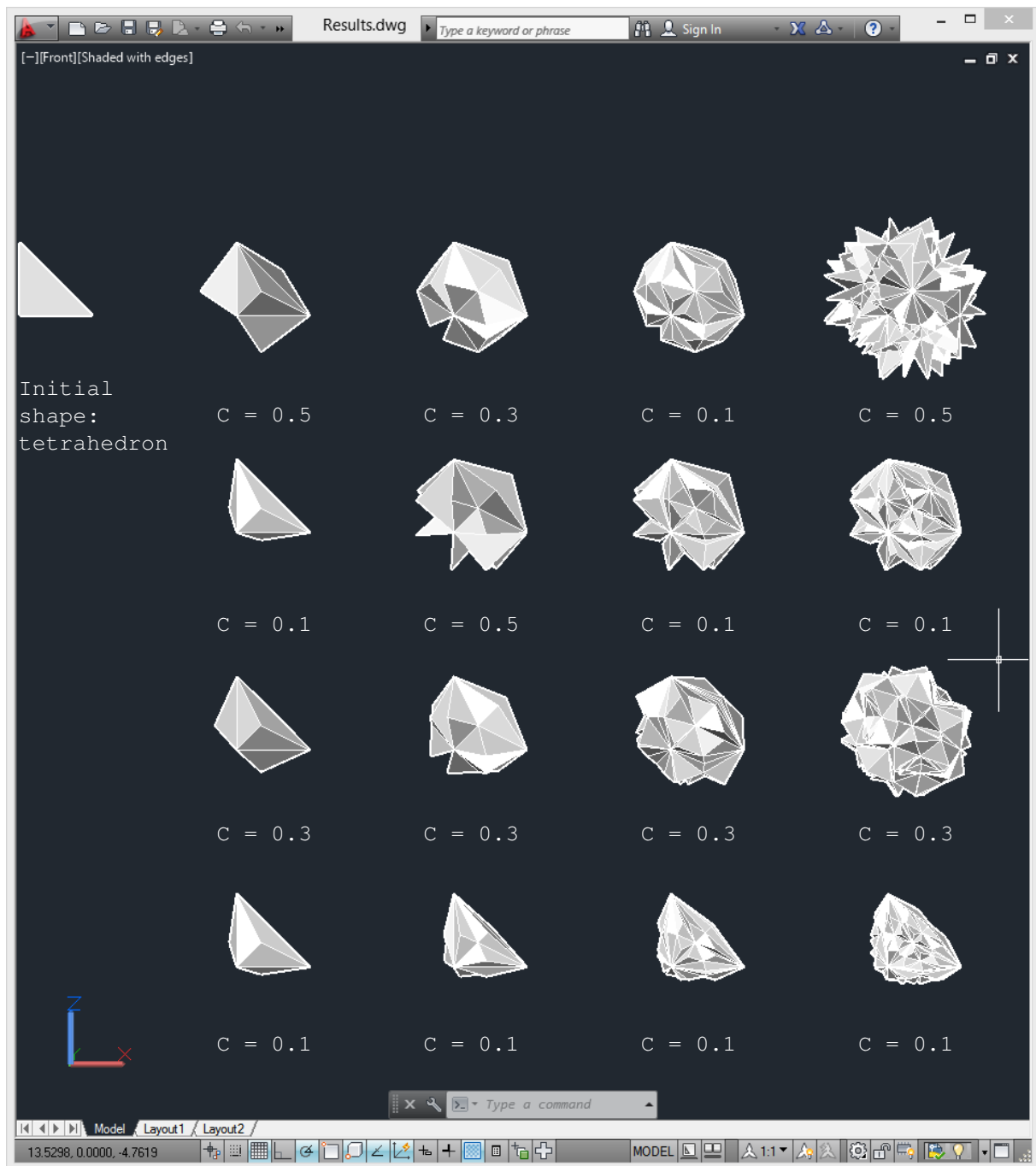


Figure 5.10: Design solutions from the 3D shape grammar - a tetrahedron as the initial shape

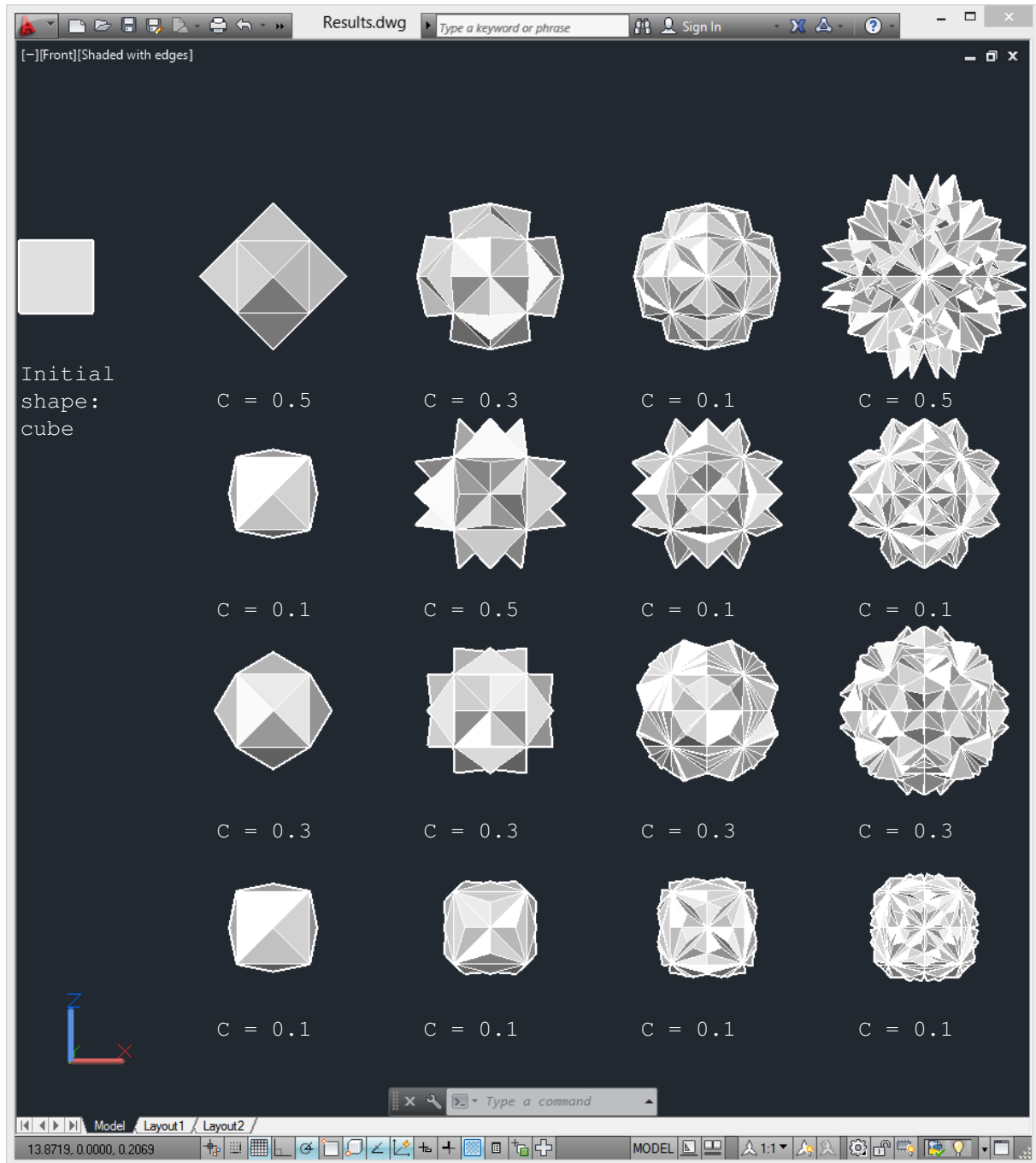


Figure 5.11: Design solutions from the 3D shape grammar - a cube as the initial shape

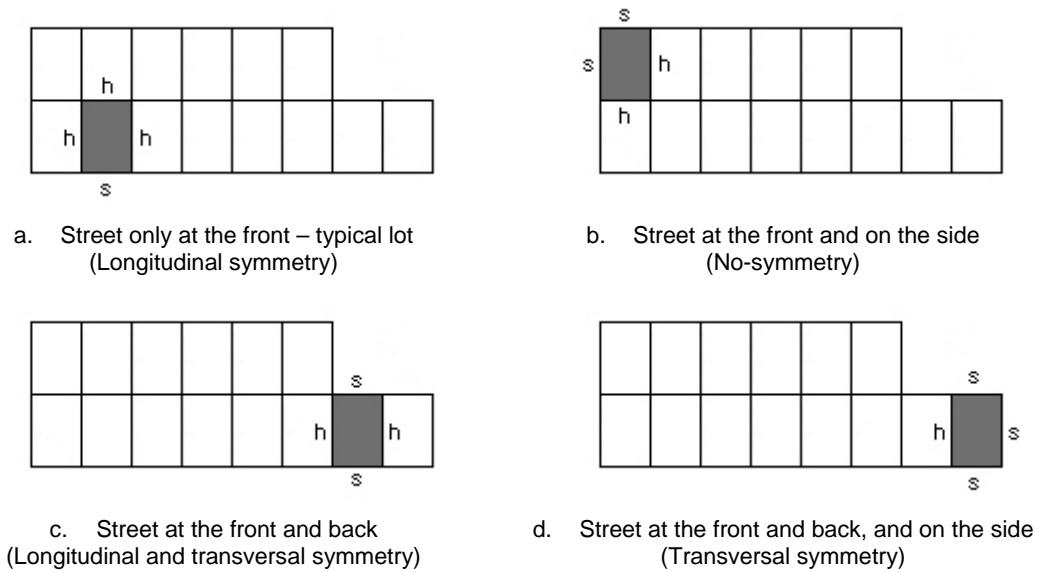
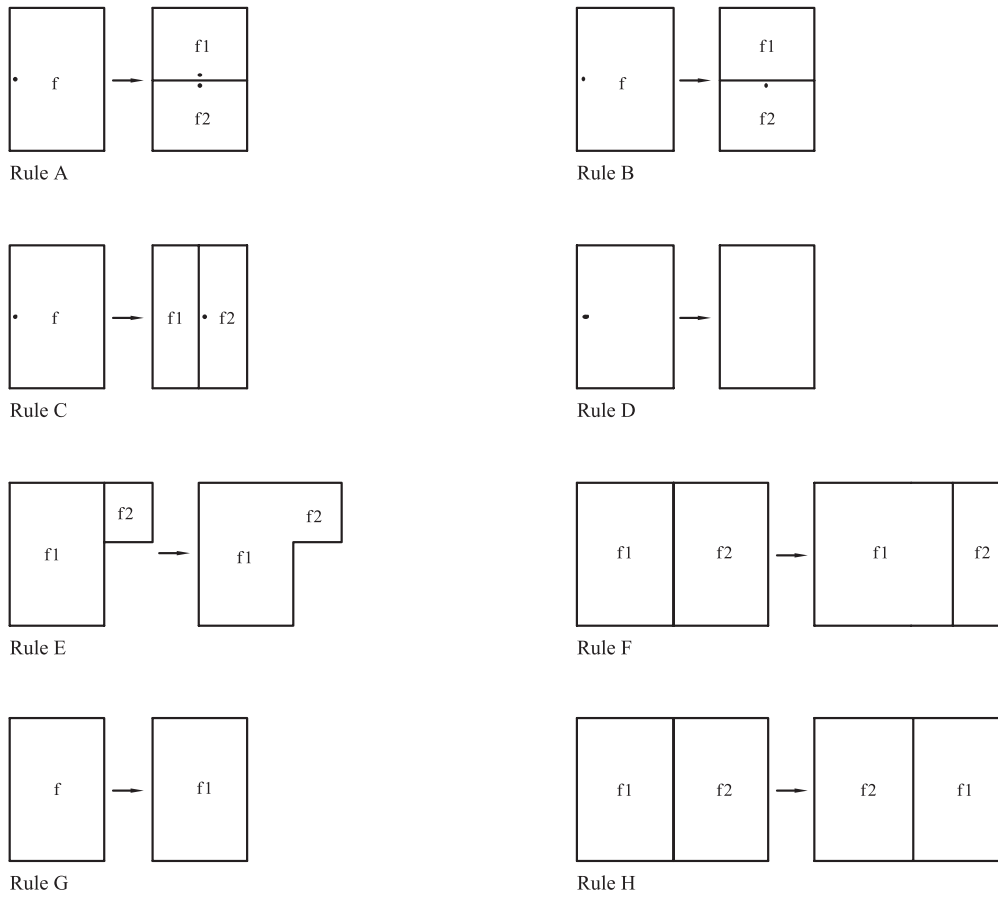


Figure 5.12: Malagueira urban contexts of houses

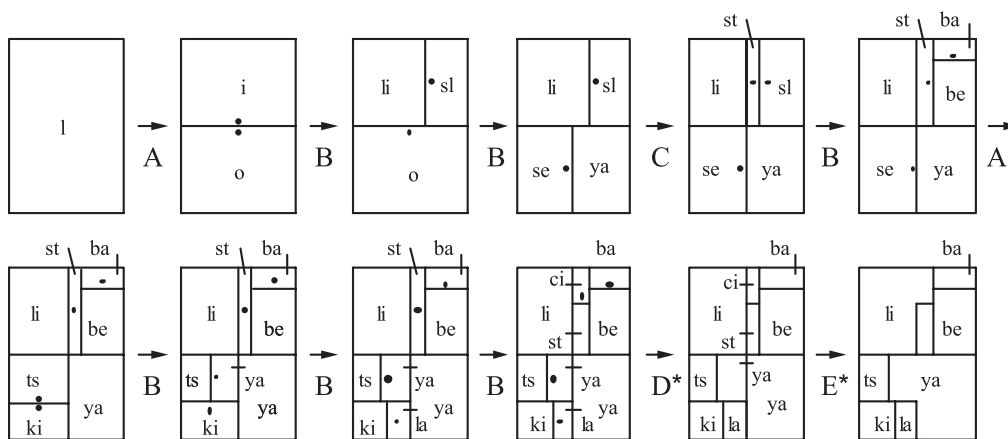
As an example, in figure 5.13 are shown a simplified set of Malagueira SG rules and a partial derivation of an existing layout (figures taken from (Duarte, 2001)). The set of rules (figure 5.13a) include rules for: (1) dissection [rules *A, B, C*], (2) connecting [*E*], and (3) extending [*F*] rectangles, (4) delete labels [*D*], (5) assigning a function [*G*], and (6) permuting functions [*H*]. Note that one of the labels in these rules, the • symbol, marks where the next dissection may occur in the space. Therefore, rules *A, B*, and *C*, dissect a space in two, where with rule *A* we can continue with more dissections in both of the resulting spaces, but in rules *B* and *C*, we can only dissect one of the two spaces. The derivation shown in figure 5.13b starts with the initial lot, the space with the label *l*, which is dissected in two using the rule *A*, creating the inside [*i*] and the outside [*o*] zones. Then, after applying the rule *B*, the inside is dissected in the living *li* and the sleeping *sl* functional zones. This process continues until a layout is found or no more rules could be applied, where the * represents that the same rule was applied several times. The meaning of labels in this figure is as follows: *l*-lot, *i*-inside zone, *o*-outside zone, *li*-living zone, *sl*-sleeping zone, *se*-service zone, *ya*-yard zone, *be*-bedroom, *ba*-bathroom, *ki*-kitchen, *ts*-transitional space, *la*-laundry, *pa*-pantry, *ci*-circulation, *st*-stairs.

Finally, while there are no direct labels described in the rules that dictate an order, the generation of a layout can be divided into steps, which also represent the usual steps used by Architects when designing projects. These steps can be described as:

- *Introduce initial shape* - retrieves the initial data from the DB and initializes the description of the design solution. The data collected includes lot dimensions, urban context, zones and respective spaces, among other. This step also generates the initial shape, representing an empty layout.
- *Locate functional zones* - this step begins by dissecting the layout in zones. The first zone to be lo-



(a) Simplified Malagueira shape grammar rules



(b) Partial derivation of an existing layout

Figure 5.13: Simplified Malagueira shape grammar rules and derivation

cated is the patio, because this zone has a major impact on functional organization by constraining windows and doors placement (Duarte, 2001). Next, the rest of the zones are located by dissecting the remaining layout.

- *Define circulation scheme* - at this step, the circulation is defined. If, for example, the house has two floors, a stair must be located, and the second floor pierced. Another example is: if the house has a backyard, then is necessary to locate a backyard corridor to connect the front of the house to the backyard.
- *Divide zones into spaces* - the basic layout of the house is generated in this step. The zones are dissected until there is no space/room to be located.
- *Introduce details* - in this step the openings for windows and doors are generated and the walls thickness are adjusted if necessary.

5.3.2 Shapes

As in the previous ice-ray grammar, a design in the Malagueira SG is also implemented as a 2D polyhedron, which represents the layout of a house, where the edges represent walls and the facets represent spaces or functional zones of a house, accordingly to their labels. For example, if a facet has a label *bedroom*, then this facet represents the space bedroom in the house.

With this representation we provide a simple abstraction of a house in 3D, which also reflects how an Architect would start to work in a project, with a 2D layout of a house. Moreover and more importantly, this representation mimics how the original SG was defined and how it operates. First, it was defined as a set of rectangles, and second, the main operation to generate designs is the dissection of rectangles.

Another important advantage, which is a direct consequence of using this shape representation, is the lack of need to represent the topology of the house in another data-structure. This happens because the polyhedron topology reflects the topology of a house. For example, if a facet with the label *kitchen* is topological connected with the facets with labels *living-room* and *yard*, then we know that, that the space kitchen is related with the space living-room and the space yard.

As already seen, to generate layouts the Malagueira grammar rely in the dissection of rectangles, which could occur between any two edges. This was explored previously in the ice-ray grammar, but from analysing the Malagueira rules we realize that, (1) the lot always seems to have an orientation given by the edge facing the street, which is always the lower edge of the initial rectangle, and (2) the dissection operations always occur horizontally or vertically in relation to this edge.

Given these two observations and the fact that a facet is defined by a frontier of halfedges in a counterclockwise way, we opt to give each facet a local orientation defined by the axis that goes through

both vertices of its first halfedge and opposing halfedge, and with origin in the vertex of the opposing halfedge. With this assumption we define two abstract operators that dissect a facet in relation to this "special" edge, where both operators create a new facet and a new edge parallel or perpendicular to this axis, representing a horizontally or a vertically dissection, respectively.

These operators bring two advantages, (1) the dissection of a facet can only occur at a certain value from the local origin, at most it can only occur near zero or the length or width of the facet, and (2) it reduces the symmetries of the dissections, allowing it to occur in only one direction.

In summary, the initial shape of the grammar is a rectangular facet in a polyhedron of eight by twelve units, where its first edge define a local orientation that dictates how the operators, that dissect a facet horizontally or vertically, behave.

To illustrate how these operators work and the consequences of its applications, consider the example shown in the figure 5.14. Here, an initial rectangle is consecutively dissected, once horizontally, twice vertically and again horizontally. In the figure we see the relations of each facet first halfedge as the red arrows, and the values at which the dissection occurs from each local origin, identified by the blue arrows. Finally, note that the red arrows always keep the same orientation axis and its origin consistently across the operators applications.

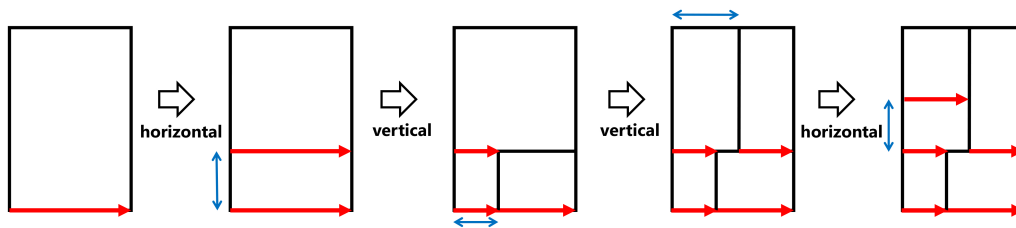


Figure 5.14: Example of application of the abstract operators defined

5.3.3 Rules

As discussed previously, while there is no predefined order in which the rules are to be applied, the generation of a layout can be divided into steps. These, abstract and group a set of rules into logical units relating the function of the rules to what an Architect would do when generating layouts of a house. To take advantage of this, and instead of using a big set of rules together with labels to dictate the order to apply rules, we define a new strategy based in the recursive application of these steps that apply and condense a subset of the rules at each step.

Therefore the steps implemented, and the order in which they are applied are: (1) introduce the initial shape, (2) locate the inside and outside zones, (3) locate backyard corridor, if needed, (4) locate patio, (5) locate remaining functional zones, (6) locate the spaces, and finally (7) make the windows and

doors openings and introduce the respective windows and doors.

To help understand in more detail, consider the five rules in the first step as defined in the Malagueira grammar. These rules first introduce one initial auxiliar shape, then create the slab, introduce the walls, and finally adjust the wall thickness in all sides of the house, accordingly to the urban context. First, to avoid mixing both 2D and 3D representations, mixing layouts and the thickness of the walls, respectively, we opt to delay the 3D representation until we want to show a solution to the end user, where the wall thickness is saved in a label of each edge, accordingly to the urban context. This way we can always know what are the wall thicknesses without mixing 2D and 3D representations. Consequently, this initial step can be condensed in just one rule, simplifying its implementation, which creates an initial rectangle of eight by twelve meters with the proper labels in facets and edges.

All the steps work in similar fashion, by dissecting rectangles. The first step just introduces a new rectangle with the given dimensions, as just described. The second divides horizontally the initial rectangle, a lot, in relation to the edge facing the street (in this implementation, the first halfedge of the rectangle). The resulting rectangles are labelled *inside* and *outside* according to the features of the house. The third step, and if the house has a back yard, dissects vertically the facet with the label *inside*, which must be facing the street. The dissection occurs, left or right in relation to the local origin of the facet, depending if there are houses on each side of the lot. If there are houses on both sides, then two solutions are generated, one for the left and one for the right dissection. Otherwise the dissection occurs on the side where there is a house. The next three steps are described next and the last step, which introduces the windows and doors, happens when a solution is presented to the user, and is described in the next section.

To locate the functional zones we dissect vertically both the *outside* and *inside* zones and assign the respective labels starting with the yard, because it is the zone that restricts the most a layout. Note that both the functional zones and the spaces, are located in descending order of their areas, and both follow a similar approach: if there is only one zone or space to be located, and if the facet has at least the required area, then the zone or space is assigned to the facet. If not, we continue to dissect and assigning zones and spaces until only one is left.

Finally, when we are locating the spaces in the functional zones through dissections, we have to take into account three aspects. First, for this approach to work we had to assume that the total area of a space to be located was the sum of the areas of the given space with the associated included and delimited spaces. Has an example, consider a house with only one bedroom. The grammar/regulations defines for this house a kitchen with five square meters, and an included laundry and pantry spaces with two and zero point seventy five squared meters, respectively. Because they are included spaces, representing that does not exist a physical wall separating the space, we opt to represent the area of the kitchen as of seven point seventy five square meters and disregard these included spaces when locating the kitchen in the layout. Second, and to avoid an infinite number of solutions, we also restrict the

values where dissections can occur. And, third, to avoid strange spaces, that is, spaces that respect the regulations but that an Architect would never do, we implement with the grammar the functionality to filter spaces that do not respect the ratio of a given constant. This ratio is calculated as the fraction of $\frac{width}{length}$, if the length of the space is bigger than its width, or $\frac{length}{width}$ otherwise. In the end this allows us to avoid that the final spaces, with smaller areas, to become long and mostly narrow spaces. Afterall no one wants a house in which a bathroom has one meter of width and three meters of length.

5.3.4 Output

To create the final 3D model of a house we take the 2D layout and apply a set of operations to: (1) create the slab, (2) create the walls, (3) introduce the windows and doors. To create the slab, we just create a box corresponding to the lot dimensions and the slab thickness. With the creation of the walls there is a problem, the edges in the 2D layout may represent any of the center or both the sides of a wall with thickness. To simplify, we choose that the edges will represent the center of the walls and its thickness are all the same. This decision also simplifies the placement of the windows and doors, because these will be already in the center of the wall. Finally, the windows and doors openings are carved directly in the walls with boolean operations provided by Rosetta, which are followed with their placement with extruded surfaces. We also implemented a series of helper functions to display houses in matrices and along the axes.

In figure 5.15 is shown the evolution of the generation of a house layout in 3D. Each design, from left to right, shows the introductions in a house of: (1) the slab and enclosing walls (the initial shape), (2) the central wall (locating the inside and outside zones) (3), (4) an (5) of more walls (locating the backyard corridor, functional zones and spaces), and finally (6) of the openings and placing the windows and doors.

In figure 5.16 is shown one final result. And in figure 5.17 are shown sixteen possible designs of a backyard house.

5.4 Discussion

SG were developed initially by Stiny and Gips as a system to perform computations with shapes, but, as shown with the implementation of the ice-ray grammar and the Malagueira grammar, SGs still depend in textual descriptions.

As a result, from more than one hundred rules available in the Malagueira grammar, we implemented less than one fifth of the rules, mainly because the rules described by this grammar present many ambiguities and even some errors. For the human reader, which copes well with errors and omissions, he/she can understand most of the rules and generate a design of a house, as demonstrated with

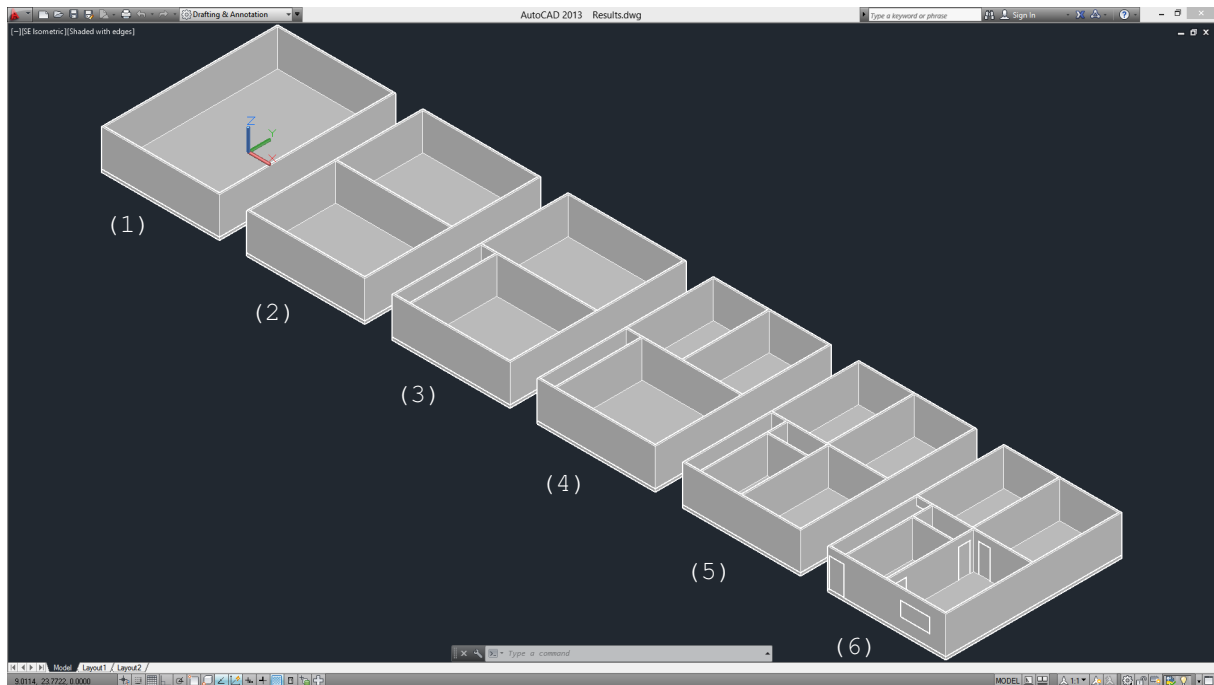


Figure 5.15: Evolution of design solution using Malagueira grammar - (1) initial slab and walls, (2) locate inside/outside, (3) locate backyard corrido, (4) locate zones, (5) locate spaces, and (6), open and introduce windows and doors

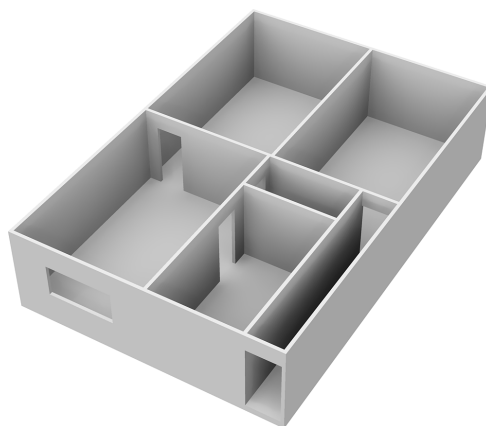


Figure 5.16: One final design solution of a house with a back yard

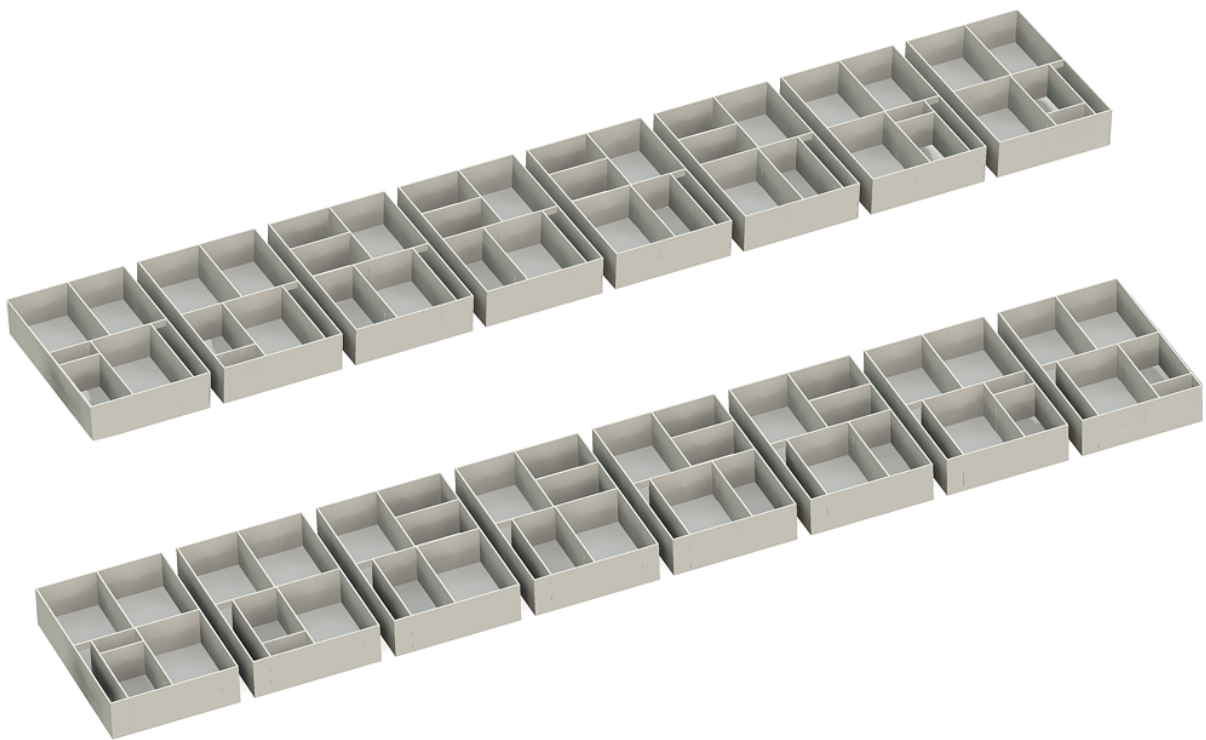


Figure 5.17: Sixteen possible solutions for a backyard house

the experiments documented in (Duarte, 2001) and (Duarte, 2007). But, for these rules to be implemented in current systems, they need to be revised to make them rigorous and unambiguous. The step, at least for the Malagueira was already taken with this work.

To support the previous statement, we present some examples of ambiguities and errors, as present in (Duarte, 2001) and (Duarte, 2007):

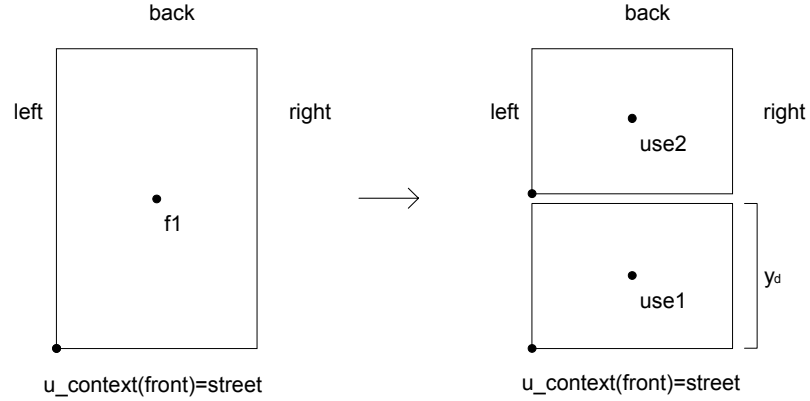
- In rule 5, illustrated in figure 5.18, *locate inside/outside zones on the first floor*, while in the text it is referenced that the lot is dissected only between two values, six or seven meters, in the description part of the rule, it shows another value, five.
- Again in rule 5, the variable α_8 is described with the values *backyard* and *frontyard* but actually, this variable describes the number of floors. Even if this variable was corrected in the rule to reference the variable α_7 , the yard location, it would still contain an error: if the house has the yard in the back, the zone facing the street could not be the *outside*, as stated in the rule.
- In rules 6 and 7, there is the same error with variable α_8 as in rule 5.
- If the house is to have a balcony, there is no rule to dissect a shape and generate a balcony. The human reader, with his common sense, can cope with this omission, and create the balcony if necessary. But the SG interpreter will not just create something that was not specified as a rule.
- In rule 9, *Locate the backyard corridor on the first floor*, the orientation of the figure it seems wrong, because in all the other rules, the urban context is in the bottom of the shape, and in this rule is on the left. As a result, the label • is in the wrong corner, making the rule to never to be identified and fired in a SG interpreter. Again, the human reader can cope with this and make the correct adjustment by introducing the backyard corridor, but the SG interpreter would not produce any corridor.¹
- As in the previous item, rule 10 seems to have an orientation contrary to the previous rules defined. The problem is not the absence of the urban context description, but the position of the label • that suggests the error. The correct position of this label, and without changing the shape orientation, this should be positioned in the upper left corner. Moreover, and even that the label position is corrected, one more error exists that would prevent the generation of correct design solutions. The variable α_5 should be five and not two, changing the meaning of the rule to assign a patio to the outside zone if the house has five bedrooms, and not two bedrooms.
- In rule 13, in the second row, third column, the left lower label *zone1* of the lot, should be *zone2*. Without this change, the solutions design would not be fully generated as intended.

¹Note that in the implementation of Malagueira using DESIGNA, this rule was corrected - the corridor can be introduced in either sides of the house, as stated by the urban context.

- Other rules, such as rules 81 *Erase axis of symmetry*, 82 *Erase invisible patio opening*, and others, can even not make sense if one does not have already generate a design. Also, questions arise such, where was the axis of symmetry created? What are and where are the invisible openings?
- One final detail is the very small rectangle present in many rules (2, 6, 7, 8, 11, 19, 20-26, 28, 36, and so on). This label seems to indicate some form of a coordinate system or even the location of the lot in a block, but given similar rules, like the rule 10 and rule 11, why only in the rule 11 there is such label?

All this suggests that, from the point of view of one who is going to implement such SG rules in a SG interpreter, he/she will not have an easy job.

R5: Locate inside/outside zones on the first floor



$$\begin{aligned}
\alpha_1 &\leftarrow \alpha_1 \\
\alpha_8 &\leftarrow \alpha_8 \\
\alpha_9 &\leftarrow \alpha_9, \forall \alpha_1, \alpha_8 = \text{frontyard} \wedge \alpha_9 = \text{true} \\
&\quad \Rightarrow \text{use1} = \text{outside1} \wedge \text{use2} = \text{inside1} \wedge y_d = 6.00 \wedge a_{in} = a_{\text{use2}} \wedge a_{ou} = a_{\text{use1}} \\
\forall \alpha_1, \alpha_8 = \text{frontyard} \wedge \alpha_9 = \text{false} \\
&\quad \Rightarrow \text{use1} = \text{outside1} \wedge \text{use2} = \text{inside1} \wedge y_d = 7.00 \wedge a_{in} = a_{\text{use2}} \wedge a_{ou} = a_{\text{use1}} \\
\alpha_1 &= \langle \text{street}, ?\text{use}, \text{street}, ?\text{use} \rangle, \forall ?\text{use} \wedge \alpha_8 = \text{backyard}, \forall \alpha_9 \\
&\quad \Rightarrow \text{use1} = \text{inside1} \wedge \text{use2} = \text{outside1} \wedge y_d = 7.00 \wedge a_{in} = a_{\text{use1}} \wedge a_{ou} = a_{\text{use2}} \\
\alpha_1 &= \langle \text{street}, ?\text{use}, \text{street}, ?\text{use} \rangle, \forall ?\text{use} \wedge \alpha_8 = \text{backyard}, \alpha_9 = \text{true} \\
&\quad \Rightarrow \text{use1} = \text{outside1} \wedge \text{use2} = \text{inside1} \wedge y_d = 6.00 \wedge a_{in} = a_{\text{use1}} \wedge a_{ou} = a_{\text{use2}} \\
\alpha_1 &= \langle \text{street}, ?\text{use}, \text{street}, ?\text{use} \rangle, \forall ?\text{use} \wedge \alpha_8 = \text{backyard}, \alpha_9 = \text{false} \\
&\quad \Rightarrow \text{use1} = \text{outside1} \wedge \text{use2} = \text{inside1} \wedge y_d = 5.00 \wedge a_{in} = a_{\text{use1}} \wedge a_{ou} = a_{\text{use2}} \\
\delta_{13} &\leftarrow \delta_{13} - \langle [(f1, id_{f1}, \emptyset, ((x_{f1}, y_{f1}, z_{f1}), dx_{f1}, dy_{f1}, dz_{f1}, a_{f1}) > \\
&\quad + \langle [(use1, id_{f1}, \emptyset, ((x_{f1}, y_{f1}, z_{f1}), dx_{f1}, dy_{f1} - (dx_{f1} - y_d + 2 \cdot 0.10), dz_{f1}, dx_{f1} \cdot dy_{f1} - (f1_{dy} - y_d + 2 \cdot 0.10)), \\
&\quad [(use2, \max(id) + 1, \emptyset, ((x_{f1}, y_{f1} + y_d, z_{f1}), dx_{f1}, dy_{f1} - y_d, dz_{f1}, dx_{f1} \cdot dy_{f1} - y_d)] > \\
\delta_{15} &\leftarrow \delta_{15} + \langle \text{available}, (f1_{dx} \cdot 0.20, a_{in}, - (a_{in} + f1_{dx} \cdot 0.20), - f1_{dx} \cdot 0.20), - A_u / A_g + A_u - f1_{dx} \cdot 0.20 / A_g > \\
\delta_{17} &\leftarrow \delta_{17} - \langle [id_{f1}, id_{?space}, \text{adjacent}], ?space \in \{\text{front}, \text{left}, \text{back}, \text{right}\} \\
&\quad + \langle [id_{\text{inside1}}, id_{?left}, \text{adjacent}], \\
&\quad \quad [id_{\text{inside1}}, id_{?right}, \text{adjacent}], \\
&\quad \quad [id_{\text{outside1}}, id_{?left}, \text{adjacent}], \\
&\quad \quad [id_{\text{outside1}}, id_{?right}, \text{adjacent}], \\
&\quad \quad [id_{\text{inside1}}, id_{?space1}, \text{adjacent}] \\
&\quad \quad [id_{\text{outside1}}, id_{?space2}, \text{adjacent}] \\
&\quad \quad \alpha_8 = \text{frontyard} \Rightarrow ?space_1 = \text{back} \wedge ?space_2 = \text{front} \\
&\quad \quad \alpha_8 = \text{backyard} \Rightarrow ?space_1 = \text{front} \wedge ?space_2 = \text{back} \\
\delta_{20} &\leftarrow \delta_{20} + \langle [\text{wall}, \max(id) + 1, (\text{inside}, \text{outside}), ((x_{f1}, y_d - 0.10, z_{f1}), dx_{f1}, 0.20, dz_{f1}, dx_{f1} \cdot dz_{f1})] > \\
\delta_{24} &\leftarrow \delta_{24} + \text{wall_cost} (dx_{f1} \cdot dz_{f1}, \text{unit_cost} (\text{wall}, 0.20, \text{material})) \\
\alpha_{25} &\leftarrow \alpha_{25} + \langle [R4, 0] >
\end{aligned}$$

Figure 5.18: Malagueira shape grammar rule number 5 - Locate inside/outside zones on the first floor.

6 Conclusions

Because shape grammars are visual by nature and current computer systems are symbolic, an implementation of a SG interpreter in these systems poses many difficulties. We described some of these difficulties on chapters 3 and 4, where we discussed the problems of shape representation, control of rule application and visualization of the generated designs. As was summarized in chapter 2, and in spite of several decades of research, there is still a lack of SG interpreters, particularly, those that can handle three-dimensional shapes.

Moreover, as was explained in chapter 4, blind application of SG rules can cause a combinatorial explosion of solution, thus making it even more difficult to produce useful results.

Finally, there is the problem of rule ambiguity. The human being can usually understand incomplete and ambiguous SG rules, inferring the missing parts or correcting the wrong ones without much trouble. That is not the case with current computer systems, making the implementation of SG rules much more difficult. As a result, current SG interpreters usually provide (1) limited shape representation, (2) poor control of rule application, and (3) in their majority, do not support directly the designers standard CAD tool to present or further work the solutions

To overcome some of the problems that affect current SG interpreters, we proposed a new SA for a SG interpreter focusing in the following fundamental features: (1) labeled shapes based on a rigorous geometry kernel, (2) rules as transition operators, and (3) integration in the typical workflow of the designer.

Using our approach, we showed that is possible to implement a SG interpreter, which we named DESIGNA, by: (1) using a sophisticated and extensible geometric kernel to represent shapes in combination with customizable precise numeric operations and correct shape predicates, (2) describing rules as transition operators in a state space of designs, (3) providing a set of search algorithms that apply transition operators to shapes to generate other shapes, and (4) using Rosetta to provide a bridge to CAD applications so that generated designs can be visualize or further processed.

To evaluate DESIGNA we wrote three SGs, demonstrating the applicability of the SG interpreter to different design problems. We also showed that: (1) the shape representation chosen is flexible enough to represent 2D and/or 3D shapes, and supports features that were not available natively, e.g., detection of collinear edges in a facet, (2) while DESIGNA provides several strategies to apply rules, it also allows the user to specify his own strategy, and finally (3) the use of Rosetta not only provides an abstraction

over the CAD tools but also promotes the portability of SGs across different CAD applications.

6.1 *Future Work*

Emergence is still a hot topic of research, particularly, for shape grammars. In general, emergence can be very empirical and it is up to the designer to tell what and how the system recognizes as emergent. In the end, DESIGNA provides the basic tools and some techniques to implement emergence of shapes. One possible path would be linking CGAL to a graph transformation system and thus simplify the description of rules and allow better support to emergence.

One issue to be resolved is the management of memory in the wrapper between the C++ and the Racket environment. While the objects allocated within the Racket environment are garbage collectable, the memory allocated in the C++ counterpart is not automatically managed.

We would like to see more SGs implemented using DESIGNA, e.g., the coffee maker (Agarwal & Cagan, 1998) and the Queen Anne houses grammar (Flemming, 1987). It will be also interesting to develop a user interface to create shapes and describe rules visually, as it would make the use of DESIGNA much simpler for a designer.

Finally and regarding the Malagueira grammar, it is our goal to continue to extend the set of rules implemented.

Bibliography

- Agarwal, M., & Cagan, J. (1998). A blend of different tastes: the language of coffeemakers. *Environment and Planning B: Planning and Design*, 25(2), 205–226.
- Agarwal, M., Cagan, J., & Stiny, G. (2000). A micro language: generating mems resonators by using a coupled form - function shape grammar. *Environment and Planning B: Planning and Design*, 27(4), 615–626.
- Baumgart, B. (1972). *Winged edge polyhedron representation* (Tech. Rep.). Stanford, USA: Stanford University.
- Baumgart, B. (1975). A polyhedron representation for computer vision. In *Proceedings of the may 19-22, 1975, national computer conference and exposition* (pp. 589–596). New York, NY, USA: ACM.
- Brown, K. N., McMahon, C. A., & Williams, J. H. S. (1994). A formal language for the design of manufacturable objects. In *Proceedings of the ifip tc5/wg5.2 workshop on formal design methods for cad* (pp. 135–155). New York, NY, USA: Elsevier Science Inc.
- Buelinckx, H. (1993). Wren's language of city church designs: a formal generative classification. *Environment and Planning B: Planning and Design*, 20(6), 645–676.
- Cagdas, G. (1996). A shape grammar: the language of traditional turkish houses. *Environment and Planning B: Planning and Design*, 23(4), 443–464.
- Carlson, C. (1993). *Grammatical programming: an algebraic approach to the description of design spaces*. Unpublished doctoral dissertation, Carnegie Mellon University, USA.
- Carlson, C., Woodbury, R., & McKelvey, R. (1991). An introduction to structure and structure grammars. *Environment and Planning B: Planning and Design*, 18(4), 417–426.
- Cenani, S., & Cagdas, G. (2006). Shape grammar of geometric islamic ornaments. In *Proceedings of the 24th conference on education in computer aided architectural design in europe* (pp. 290–297).
- Chase, S. C. (1989). Shapes and shape grammars: from mathematical model to computer implementation. *Environment and Planning B: Planning and Design*, 16(2), 215–242.

- Chase, S. C. (2010, July). Shape grammar implementations: the last 35 years. In *Workshop shape grammar implementation: From theory to useable software on fourth international conference on design computing and cognition*. University of Stuttgart, Stuttgart, Germany.
- Chau, H. (2002). *Preserving brand identity in engineering design using a grammatical approach*. Unpublished doctoral dissertation, School of Mechanical Engineering, University of Leeds, England.
- Chau, H., Chen, X., Alison, M., & Alan, d. P. (2004). Evaluation of a 3d shape grammar implementation. *First International Conference on Design Computing and Cognition (DCC'04)*, 357–376.
- Chien, S., Donia, M., Snyder, J., & Tsai, W. (1998). Sg-clips: A system to support the automatic generation of designs from grammars. *CAADRIA 1998: Proceedings of The Third Conference on Computer Aided Architectural Design Research in Asia*, 445–454.
- Chiou, S.-C., & Krishnamurti, R. (1995). The grammar of taiwanese traditional vernacular dwellings. *Environment and Planning B: Planning and Design*, 22(6), 689–720.
- Chiou, S.-C., & Krishnamurti, R. (1996). Example taiwanese traditional houses. *Environment and Planning B: Planning and Design*, 23(2), 191–216.
- Chomsky, N. (1957). *Syntactic structures*. The Hague: Mouton.
- Colakoglu, B. (2005). Design by grammar: an interpretation and generation of vernacular hayat houses in contemporary context. *Environment and Planning B: Planning and Design*, 32(1), 141–149.
- Correia, R., Duarte, J., & Leitão, A. (2010, July). Malag: a discursive grammar interpreter for the online generation of mass customized housing. In *Workshop shape grammar implementation: From theory to useable software on fourth international conference on design computing and cognition*. University of Stuttgart, Stuttgart, Germany.
- Correia, R., Duarte, J., & Leitão, A. (2012, September). Gramatica: A general 3d shape grammar interpreter targeting the mass customization of housing. In *Proceedings of the 30th conference on education in computer aided architectural design in europe* (Vol. 1, pp. 489–496). Faculty of Architecture, Prague, Czech Republic.
- Downing, F., & Flemming, U. (1981). The bungalows of buffalo. *Environment and Planning B*, 8(3), 269–293.
- Duarte, J. (2001). *Customizing mass housing: a discursive grammar for siza's malagueira houses*. Unpublished doctoral dissertation, Department of Architecture, Massachusetts Institute of Technology, USA.
- Duarte, J. (2005a). A discursive grammar for customizing mass housing: the case of siza's houses at malagueira. *Automation in Construction*, 14(3), 265–275.

- Duarte, J. (2005b). Towards the mass customization of housing: the grammar of siza's houses at malagueira. *Environment and Planning B: Planning and Design*, 32(3), 347–380.
- Duarte, J. (2007). *Personalização de habitação em série: Uma gramática discursiva para as casas da malagueira do siza*. Fundação Caloust Gulbenkian.
- Duarte, J., & Correia, R. (2006). Implementing a description grammar for generating housing programs online. *Construction Innovation Journal on Information and knowledge Management in Construction*, 6(4), 203–216.
- Duarte, J., & Rocha, J. (2006). A grammar for the patio houses of the medina of marrakech. towards a tool for housing design in islamic contexts. In *Proceedings of the 24th conference on education in computer aided architectural design in europe* (pp. 860–866).
- Duarte, J., Rocha, J., & Soares, G. (2007, October). Unveiling the structure of the marrakech medina: A shape grammar and an interpreter for generating urban form. *Artif. Intell. Eng. Des. Anal. Manuf.*, 21(4), 317–349.
- Duarte, J., & Simondetti, A. (1997). *Basic grammars and rapid prototyping*. Lahti, Finland.
- Flemming, U. (1981). The secret of the casa giuliani frigerio. *Environment and Planning B*, 8(1), 87–96.
- Flemming, U. (1987). More than the sum of parts: the grammar of queen anne houses. *Environment and Planning B: Planning and Design*, 14(3), 323–350.
- Gardner, M. (1970). Mathematical games: The fantastic combination of john conway's new solitaire game "life". *Scientific American*, 120–123.
- Ghali, S. (2008). *Introduction to geometric computing*. Springer London, Limited.
- Gips, J. (1974). A syntax-directed program that performs a three-dimensional perceptual task. *Pattern Recognition*, 6, 189–199.
- Gips, J. (1975). *Shape grammars and their uses: artificial perception, shape generation and computer aesthetics*. Birkhäuser.
- Gips, J. (1999). Computer implementation of shape grammars. *Workshop on Shape Computation*.
- Grasl, T., & Economou, A. (2011). Grape: using graph grammars to implement shape grammars. In *Proceedings of the 2011 symposium on simulation for architecture and urban design* (pp. 21–28). Boston, Massachusetts, USA: Society for Computer Simulation International.
- Heisserman, J. (1992). *Generative geometric design and boundary solid grammars*. Unpublished doctoral dissertation, Carnegie Mellon University, USA.

- Heisserman, J. (1994, March). Generative geometric design. *IEEE Comput. Graph. Appl.*, 14(2), 37–45.
- Jowers, I., & Earl, C. (2011). Implementation of curved shape grammars. *Environment and Planning B: Planning and Design*, 38(4), 616–635.
- Knight, T. (1980). The generation of hepplewhite-style chair-back designs. *Environment and Planning B*, 7(2), 227–238.
- Knight, T. (1981). The forty-one steps. *Environment and Planning B*, 8(1), 97–114.
- Knight, T. (1986). Transformations of the meander motif on greek geometric pottery. *Design Computing* 1, 29–67.
- Knight, T. (1989a). Color grammars: designing with lines and colors. *Environment and Planning B: Planning and Design*, 16(4), 417–449.
- Knight, T. (1989b). Transformations of de stijl art: the paintings of georges vantongerloo and fritz glarner. *Environment and Planning B: Planning and Design*, 16(1), 51–98.
- Knight, T. (1990). Mughul gardens revisited. *Environment and Planning B: Planning and Design*, 17(1), 73–84.
- Knight, T. (1994). *Transformations in design*. Cambridge, England: Cambridge University Press.
- Knight, T. (2000). *Shape grammars in education and practice: History and prospects*. Boston, Massachusetts, USA.
- Knight, T., & Stiny, G. (2001). Classical and non-classical computation. *Information Technology*, 5(4), 355–372.
- Koning, H., & Eizenberg, J. (1981). The language of the prairie: Frank lloyd wright's prairie houses. *Environment and Planning B*, 8(3), 295–323.
- Krishnamurti, R. (1982). *Sgi: a shape grammar interpreter* (Tech. Rep.). Milton Keynes: Centre for Configurational Studies, The Open University.
- Krishnamurti, R., & Earl, C. F. (1992). Shape recognition in three dimensions. *Environment and Planning B: Planning and Design*, 19(5), 585–603.
- Krishnamurti, R., & Giraud, C. (1986). Towards a shape editor: the implementation of a shape generation system. *Environment and Planning B: Planning and Design*, 13(4), 391–404.
- Li, A. (2001). *A shape grammar for teaching the architectural style of the yingzao fashi*. Unpublished doctoral dissertation, Department of Architecture, Massachusetts Institute of Technology, USA.

- Lindenmayer, A. (1968). Mathematical models for cellular interaction in development. *Journal of Theoretical Biology*, 18, 280–315.
- Lopes, J. (2012). *Modern programming for generative design*. Unpublished master's thesis, Instituto Superior Técnico.
- Lopes, J., & Leitão, A. (2011). Portable generative design for cad applications. In *Proceedings of the 31st annual conference of the association for computer aided design in architecture* (pp. 196–203). Banff, Alberta, Canada.
- Mayer, R. (2003). *A linguagem de oscar niemeyer*. Unpublished master's thesis.
- McCormack, J., & Cagan, J. (2002). Designing inner hood panels through a shape grammar based framework. *Artificial Intelligence in Engineering Design, Analysis and Manufacturing*, 16(4), 273–290.
- McCormack, J., Cagan, J., & Vogel, C. (2004). Speaking the buick language: capturing, understanding, and exploring brand identity with shape grammars. *Design Studies*, 25, 1–29.
- McGill, M. (2001). *A visual approach for exploring computational design*. Unpublished master's thesis, Department of Architecture, Massachusetts Institute of Technology.
- McKay, A., Jowers, I., Chau, H. H., Pennington, A., & Hogg, D. C. (2008). Computer aided design: An early shape synthesis system. In X.-T. Yan, W. J. Ion, & B. Eynard (Eds.), *Global design to gain a competitive edge* (pp. 3–12). Springer London.
- Moon, J. (2007). *Shape grammar for mies van der rohe's high-rise apartment*. Unpublished master's thesis, Department of Architecture, Massachusetts Institute of Technology.
- Neperud, B., Lowther, J., & Shene, C.-K. (2007, December). Visualizing and animating the winged-edge data structure. *Computers and Graphics*, 31(6), 877–886.
- Orsborn, S., Cagan, J., Pawlicki, R., & Smith, R. C. (2006). Creating cross-over vehicles: Defining and combining vehicle classes using shape grammars. *Artificial Intelligence in Engineering Design, Analysis and Manufacturing*, 20, 217–246.
- Parish, Y. I. H., & Müller, P. (2001). Procedural modeling of cities. In *Proceedings of the 28th annual conference on computer graphics and interactive techniques* (pp. 301–308).
- Piazzalunga, U., & Fitzhorn, P. (1998). Note on a three-dimensional shape grammar interpreter. *Environment and Planning B: Planning and Design*, 25(1), 11–30.
- Post, E. (1943). Formal reductions of the general combinatorial decision problem. *American Journal of Mathematics*, 65(2), 197–215.

- Prats, M., Earl, C., Garner, S., & Jowers, I. (2006, June). Shape exploration of designs in a style: Toward generation of product designs. *Artif. Intell. Eng. Des. Anal. Manuf.*, 20(3), 201–215.
- Pugliese, M., & Cagan, J. (2002). Capturing a rebel: Modeling the harley-davidson brand through a motorcycle shape grammar. *Research in Engineering Design*(13), 139–156.
- Russell, S., & Norvig, P. (2010). *Artificial intelligence: A modern approach*. Prentice Hall.
- Said, S., & Embi, M. (2008). A parametric shape grammar of the traditional malay long-roof type houses. *International Journal of Architectural Computing*, 6(2), 121–144.
- Sener, S. M., & Gorgul, E. (2009). A shape grammar algorithm and educational software to analyze classic ottoman mosques. *A—Z Journal of Faculty of Architecture*, 5(1), 12–30.
- Stiny, G. (1977). Ice-ray: a note on the generation of chinese lattice designs. *Environment and Planning B*, 4(1), 89–98.
- Stiny, G. (1980a). Introduction to shape and shape grammars. *Environment and Planning B*, 7(3), 343–351.
- Stiny, G. (1980b). Kindergarten grammars: designing with froebel's building gifts. *Environment and Planning B*, 7(4), 409–462.
- Stiny, G. (1981). A note on the description of designs. *Environment and Planning B*, 8(3), 257–267.
- Stiny, G. (1991). The algebras of design. *Research in Engineering Design*, 2, 171–181.
- Stiny, G. (1992). Weights. *Environment and Planning B: Planning and Design*, 19(4), 413–430.
- Stiny, G., & Gips, J. (1972). Shape grammars and the generative specification of painting and sculpture. In *C v freiman (ed.) information processing 71 (amsterdam: North-holland)* (pp. 1460–1465).
- Stiny, G., & Mitchell, W. J. (1978). The palladian grammar. *Environment and Planning B*, 5(1), 5–18.
- Stiny, G., & Mitchell, W. J. (1980). The grammar of paradise: on the generation of mughul gardens. *Environment and Planning B*, 7(2), 209–226.
- Stouffs, R. (1994). *The algebra of shapes*. Unpublished doctoral dissertation, Carnegie Mellon University, USA.
- Tapia, M. (1996). *From shape to style. shape grammars: Issues in representation and computation, presentation and selection*. Unpublished doctoral dissertation, University of Toronto, Toronto.
- Trescak, T., Esteva, M., & Rodriguez, I. (2010, July). A shape grammar interpreter for rectilinear forms. In *Workshop shape grammar implementation: From theory to useable software on fourth international conference on design computing and cognition*. University of Stuttgart, Stuttgart, Germany.

- Wang, Y. (1998). *3d architecture form synthesizer*. Unpublished master's thesis, Department of Architecture, Massachusetts Institute of Technology.
- Wong, W., & Cho, C. (2004). A computational environment for learning: basic shape grammars. *International Conference on Computers in Education*, 287–292.
- Wu, Q. (2005). Bracket teaching program: A shape grammar interpreter. *Automation in Construction*, 14(6), 716–723.