New Computational Techniques for Detecting, Learning and Managing Criteria in Design Problems

UNIVERSIDAD
DE MÁLAGA

TESIS DOCTORAL

Manuela Ruiz Montiel

Universidad de Málaga

Julio de 2016

Documento maquetado con TEXIS v.1.0.

New Computational Techniques for Detecting, Learning and Managing Criteria in Design Problems

Memoria que presenta el doctorando

Manuela Ruiz Montiel

para optar al grado académico de Doctora

Dirigida por el Doctor

José Luis Pérez de la Cruz Molina

Programa de Doctorado en Ingeniería del Software e Inteligencia Artificial
Departamento de Lenguajes y Ciencias de la Computación
Escuela Técnica Superior de Ingeniería Informática
Universidad de Málaga

Julio de 2016

El Dr. D. José Luis Pérez de la Cruz Molina, Catedrático de Universidad, del Área de Ciencias de la Computación e Inteligencia Artificial de la Escuela Técnica Superior de Ingeniería Informática de la Universidad de Málaga,

Certifica que,

Dña. Manuela Ruiz Montiel, Ingeniera en Informática, ha realizado en el Departamento de Lenguajes y Ciencias de la Computación de la Universidad de Málaga, bajo su dirección, el trabajo de investigación correspondiente a su Tesis Doctoral titulada:

New Computational Techniques for Detecting, Learning and Managing Criteria in Design Problems

Revisado el presente trabajo, estima que puede ser presentado al tribunal que ha de juzgarlo, y autoriza la presentación de esta Tesis Doctoral en la Universidad de Málaga.

[illegible]

Fdo.: Dr. José Luis Pérez de la Cruz Molina

Málaga, Julio de 2016

A Francis, Juanma y mamá

Agradecimientos

Recuerdo como si fuera ayer la mañana en que me reuní con Pepe Luis para hablar acerca de la posibilidad de hacer el doctorado bajo su tutela. Cuando me explicó que podía desarrollar mi tesis en el ámbito donde la inteligencia artificial se une con el diseño, supe que estaba en el lugar adecuado. Le doy las gracias de corazón por ofrecerme esta posibilidad y sobre todo por la dedicación, disponibilidad y paciencia que ha demostrado desde aquella mañana. Sus sabios consejos han conseguido llevar el desarrollo de esta tesis a buen puerto.

Gracias también a Lawrence, por su inestimable colaboración a lo largo del proceso; su minuciosidad e inteligencia sin duda han enriquecido los resultados de esta tesis. A Eva, por la gran ilusión que siempre ha depositado en mí, y por la pasión que en numerosas ocasiones me ha contagiado. A Mavi, por su gran dedicación y preocupación en que todo salga siempre perfecto.

También quiero dar las gracias a Juan y Javier por su imprescindible aportación a este trabajo. Gracias a vosotros no sólo he podido desarrollar los aspectos de esta tesis relacionados con la arquitectura, sino también aprender mucho acerca de lo que siempre ha sido una de mis vocaciones frustradas (ahora ya un poquito menos).

Thanks to everyone in the Computational Modelling Group of the Vrije Universiteit Brussels for making my stay so comfortable. Gracias a Luis, Adrián y Marjon, por los almuerzos españoles y las cervezas que compartimos. Gracias también a Isabel, por todas las veces que me hizo sentir en Bruselas como si estuviera en casa.

Gracias de corazón a mis compañeros de laboratorio, por estar ahí para apoyarnos mutuamente en los momentos duros y alegrarnos en los felices; y a los compañeros del grupo y del departamento con los que he tenido la suerte de compartir clases y cafés.

Durante todo este tiempo han sido muchas las personas ajenas a la universidad que han vivido en primera persona todo el proceso. Sin ellas, sin la alegría que me produce que existan y sin la ilusión de verlas, nada de esto habría tenido mucho sentido.

Gracias a Lucía, Rocío, Irene (traumatóloga), Irene (ingeniera) y Alicia, por vuestro humor absurdísimo, por el sushi, por los multicumples, por el pádel, por Muse. Gracias a Mar, Marina (medicina) y Marina (derecho). Aunque ahora estamos un poco repartidas por la geografía española, cada vez que nos vemos parece que sólo han pasado cinco minutos desde la última vez. Gracias por estar ahí siempre para poder contároslo todo. Gracias también a todo el grupo de ISP. Nos vemos poco, pero cuando nos juntamos nos lo pasamos pipa.

Gracias a mi encantadora familia, en la cual también incluyo a la familia de Francis, por apoyarme siempre y creer en mí. A Juanma, hermanito, por tu alegría, humor y nobleza. A mamá, por tu pasión, ayuda y dedicación infinitas. Y gracias a Francis, por tu comprensión, apoyo y amor. Tú mejor que nadie entiendes todo lo que implica convivir con un tesinando. Son muchos los proyectos que tenemos en común, y estoy deseando llevarlos a cabo.

¡Gracias!

Contents

List of Figures

List of Tables

Chapter 1

Introduction

What is your definition of "design"?
A plan for arranging elements in such a way as
to best accomplish a particular purpose

Question: Madame L'Amic. Answer: Charles
Eames

Design is a complex task that implies the consideration of requirements of different nature in order to attain a purpose. It involves the integration of technical, economical, social, and environmental needs, among others.

Nowadays, the importance of computers along the design process is unquestionable. Computer-aided design (CAD) tools have expanded the human capabilities, providing designers with accurate mechanisms for representing, describing, visualizing, organizing and simulating design projects.

In this thesis we start from the premise that computers can also play a role in the *synthesis* of design solutions. The term *computational design* has been coined for this area of research (Cagan et al., 2005), which has been accelerated thanks to advances in the understanding of what is design from a computational perspective. From a certain point of view, some aspects of the design process can be understood as a search or exploration over a space of design alternatives (Simon, 1973). The unifying thread of this thesis consists in the idea of employing artificial intelligence techniques for dealing with this design space in different manners. These techniques will have to consider design requirements of different nature, relevant to the purpose that a particular design problem seeks to achieve.

In the following we describe in detail the research scope of this thesis. We also list the different contributions and explain how the thesis is organized.

1.1 Research Scope

This section introduces the research scope of this thesis, which moves along different areas related to computational design. As we explained before, computational design is concerned about the role of computers in non trivial aspects of the design process. Our research focuses on synthesizing new design solutions that fulfil a set of design criteria.

This implies the need for techniques that can explicitly represent and generate a design space, which leads us to the area of so-called generative systems. This thesis focuses in the particular generative framework of shape grammars (Stiny & Gips, 1972), that has been widely used in the area of computational design due to its versatility.

Design criteria can be directly hard-coded into the rules of a generative system, yielding an expert set of rules that always synthesizes feasible solutions. However, expert systems are usually hard to create, modify and maintain. Another alternative is the partial or total externalization of the design criteria, giving room to a more naive generative system that cannot produce feasible solutions by itself. Hence, it needs to be combined with a control method that ensures the production of good designs. In this thesis we will follow this latter approach, and will rely on different computational and artificial intelligence techniques such as search processes and reinforcement learning methods (Sutton & Barto, 1998), always from a multi-criteria perspective, as design usually implies the consideration of multiple requirements of different nature.

In this thesis we delve into two different settings for combining a generative system (particularly shape grammars) with an external control method, depending on how we formalize and employ the design criteria. The first one consists in reducing the number of criteria hard-coded inside the rules, leaving some of the former to be managed in an explicit manner: we directly use them for evaluating shapes at each step of the generative process and guide it towards feasible solutions. The second setting goes one step further, and tries to use the whole set of requirements in a machine learning process that learns a heuristic. Such heuristic will determine how to apply the rules of the generative system. Hence, in the generative process we just rely on this newly discovered actuation information, so the design requirements are used in an implicit manner.

1.2 Contributions

The main contributions of this thesis can be summarized as follows:

1. A classification scheme for possible approaches to computational design based on shape grammars. The different classes will depend on the nature of the involved

rules, the formalization of design criteria and the external control methods

2. The development of a methodology based on the idea of encoding some of the design requirements into rules, while explicitly using others for evaluating the shapes produced along the generative process and guiding it towards good designs

3. The development of a methodology based on the idea of minimizing the design requirements encoded into the rules, and using them inside a machine learning process in order to learn a heuristic that determines how to apply the rules of the generative system

4. The application of the developed methodologies in different scenarios related to the architectural domain

The contributions of this thesis have been presented in various journals, conferences, workshops and seminars. In the following we provide a list of the contributions organized by the type of publication:

Journal Articles

- M. Ruiz-Montiel, J. Boned, J. Gavilanes, E. Jiménez, L. Mandow, J.L. Pérez-de-la-Cruz. Design with shape grammars and reinforcement learning. In *Advanced Engineering Informatics*, vol. 27, issue 2, April 2013, pp 230–245

- M. Ruiz-Montiel, M.V. Belmonte, J. Boned, L. Mandow, E. Millán, A.R. Badillo, J.L. Pérez-de-la-Cruz. Layered shape grammars. In *Computer-Aided Design*, vol. 56, November 2014, pp 104–119

International Workshops and Seminars

- M. Ruiz-Montiel, L. Mandow, J.L. Pérez-de-la-Cruz, J. Gavilanes. Shapes, grammars, constraints and policies. In *SHAPES 1.0 - The Shape Of Things 2011*, J. Hastings, O. Kutz, M. Bhatt, S. Borgo Eds., September 2011, CEUR Workshop Proceedings, Vol. 812

- M. Ruiz-Montiel. Multi-objective Reinforcement Learning. In *European Workshop on Reinforcement Learning (EWRL 11)*, P. Auer, M. Hutter, L. Orseau Eds., September 2013, Dagstuhl Research Online Publication Server: Reinforcement Learning (Dagstuhl Seminar 13321)

National Conferences

- M. Ruiz-Montiel, J. Boned, J. Gavilanes, E. Jiménez, L. Mandow, J.L. Pérez-de-la-Cruz. Proyecto arquitectónico mediante gramáticas de formas sencillas y aprendizaje. In *Conferencia de la Asociación Española para la Inteligencia Artificial*, November 2011
- M. Ruiz-Montiel, L. Mandow, J.L. Pérez-de-la-Cruz. PQ-learning: Aprendizaje por Refuerzo Multi-Objetivo. In *Conferencia de la Asociación Española para la Inteligencia Artificial*, September 2013
- M. Ruiz-Montiel, J. Boned, J. Gavilanes, P. Hidalgo, D. Belmonte, L. Mandow, J.L. Pérez-de-la-Cruz. Proyecto arquitectónico energéticamente eficiente mediante gramáticas de formas y aprendizaje por refuerzo. In *Conferencia de la Asociación Española para la Inteligencia Artificial*, November 2015

1.3 Thesis Outline

This thesis is structured in six chapters, one appendix and a list of references. Apart from this first introductory chapter and a final one containing the conclusions and possible future work, the structure of the thesis is described in the following.

Chapter 2 presents some required antecedents on computational design, shape grammars and reinforcement learning techniques.

The following 3 chapters include the main contributions of this thesis. Chapter 3 classifies the different settings that can be adopted in order to combine shape grammars, control methods and design criteria.

Chapter 4 presents a methodology for explicitly managing design criteria outside a non-expert shape grammar, using them for the evaluation of the produced shapes at each step of the generative process. In this setting, design criteria are divided into design constraints and goals, which are used to guide a search algorithm. We illustrate two examples in which the proposed methodology is put into practice.

Chapter 5 describes a methodology that employs design criteria in order to learn application heuristics for shape grammars inside which the encoded design requirements have been minimized. Two alternatives are presented, depending on how we deal with the multi-objective nature of design problems. Namely, the first alternative starts from the premise that the problem can be scalarized according to a set of design preferences that are known a priori, thus yielding an unique heuristic for the generative system. Two application cases are presented for this alternative. The second alternative does not consider the scalarization premise, and thus learns multiple heuristics that can be used to produce a set of optimal solutions.

1.4 Funding

This dissertation has been possible thanks to the support and funding of the Spanish Ministry of Education through the National Programme for Training Human Resources under the FPU (Formación de Profesorado Universitario) fellowship. Some parts of this work have been supported by and applied to the following National projects: TIN2009-14179 (Spanish Government, Plan Nacional de I+D+i) and FC14-TIN-40 (University of Málaga, Programa de fortalecimiento de las capacidades de I+D+I en las universidades 2014-2015, de la Consejería de Economía, Innovación, Ciencia y Empleo, cofinanciado por el fondo europeo de desarrollo regional (FEDER)).

Chapter 2

Antecedents

What are the boundaries of design?
What are the boundaries of problems?

Question: Madame L'Amic. Answer: Charles Eames

In this Chapter we set up some foundations to build the rest of this thesis. Firstly, it is necessary to define the problem of computational design and gain insight on the different approaches that have been explored in the literature to tackle it. This is addressed in Section 2.1. As the proposed systems in this thesis are based on one of these approaches, namely shape grammars, in Section 2.2 we will focus on this technique and the different systems that have relied on this method to deal with the problem of computational design. A significant fraction of the work in this thesis is based on a combination of shape grammars and a machine learning technique called reinforcement learning, thus in Section 2.3 we will describe this area of artificial intelligence.

2.1 The Problem of Computational Design

2.1.1 What is Design?

The word *design* can be both a noun and a verb, that is, an object and a process. The Cambridge dictionary gives the following noun definition: a design is a *drawing or set of drawings showing how a product is to be made and how it will work and look.* Papalambros & Wilde (2000) also provide a definition as a noun: *a design is an artefact defined by its geometric configuration, the materials used, and the task it performs.* As a verb, the Cambridge dictionary defines designing as *making or drawing plans for something*, or as a synonym for *intend.* Eggert (2004) provides the following definition: *design refers to the actual decision-making processes involved in creating an artefact.* It is clear that the notion of intention or purpose is embedded into the concept of design. In his book *The Sciences of the Artificial*, Simon (1968) proposed the following: *"Everyone designs who devises courses of action aimed at changing existing situations into preferred ones"*.

According to Simon, traditionally, the science disciplines are concerned about natural things. By contrast, engineering is interested in the construction of artefacts and, definitely, how to design. However, he goes further and advocates for the *sciences of the artificial*, which are aimed at understanding and predicting the behaviour of artefacts and the processes for creating them. In this context, the *science of design* is related to the general theory of search in state spaces (Simon, 1973). This idea of exploration of design spaces has been further refined in the literature (Goel, 1994; Goel & Pirolli, 1992; Stouffs, 2006) , and will be described in more detail through the following sections.

2.1.2 Design and Artificial Intelligence

Computer-aided design systems traditionally help designers in many tasks, such as description, documentation and visualization of their projects. However, these tools can also play a more active role in the design process, by means of artificial intelligence techniques.

Such techniques can be used in a number of design-related tasks, such as evaluating designs generated by humans, improving existing solutions, predicting the behaviour of a given design and even proposing new design solutions. Here we are interested in the last task, which is often referred in the literature as *computational design* or *computational design synthesis* (Cagan et al., 2005; Stanković et al., 2013; Campbell & Shea, 2014) . In the words of Campbell & Shea (2014), *"Computational design synthesis (CDS) is a research area focused on approaches to automating synthesis activities in design. Resulting methods may be fully automated or interactive with the goals of automatically generating a range of alternatives, sparking creativity and innovation, automating tedious or time-consuming engineering tasks, and simply exploring the creative abilities of computational systems"*. Certainly, computers can be considered as a colleague in the design process, as far as they can be programmed to propose solutions or trigger ideas that comply with a given set of design requirements.

The problem of computational design has been approached by artificial intelligence researchers since the 1960s (Simon, 1968; Eastman, 1969). Design problem solving is considered a particularly complex task, since these kind of problems are ill-structured (Simon, 1973), that is, they are characterized by the absence of a unique, well-defined solution. Human experts deal with these problems not only by means of knowledge about the requirements that solutions need to meet, but also with the help of creative thinking, that will lead to many valid designs substantially different between them. The diverse variants of this problem are of interest in many fields like architecture and engineering, where geometric design plays a crucial role.

As stated by (Simon, 1973), in spite of the ill-defined nature of design problems, designers are able to structure design problems correctly and then use a search process over a state space. According to Woodbury & Burrow (2006), "design space exploration *is the idea that computers can usefully depict design as the act of exploring alternatives. This involves representing many* designs*, arraying these represented designs in a network structure termed the* design space*, and exploring this space by traversing* paths *in the network to visit both previously represented designs and to find sites for new insertions into the network"*. Hence, decision techniques based on artificial intelligence can be useful for discovering design solutions, by means of the execution of search methods over the design space.

The definition of design as a process of search or exploration over a design space implies the need for methods that can explicitly represent or generate such space, which are described in the following section.

2.1.2.1 The Automatic Generation of Designs

A *generative design* technique is any algorithmic method with the ability to synthesize shapes. For a particular generative design system, there is an associated design space that can be produced by executing the algorithm or applying the rules that define the system. The application of these rules can also be viewed as a means for moving from one state in the design space to another one.

Several kinds of generative systems taken from disciplines as mathematics and biology are employed as tools for generating forms in design fields like architecture. Some examples are *cellular automata*, *L-systems*, *shape grammars* and *graph grammars*.

A cellular automaton is a discrete model proposed by von Neumann in the late 1950s. It consists of a grid of cells of a finite number of dimensions. Each cell can have one of a finite possible different states, and they change its state according to a rule based on its current state and the ones of the neighbour cells. Probably the most popular application of cellular automata is the Game of Life (Conway, 1970), in which complex patterns can arise from very simple rules. Due to the ability of cellular automata for generating such complex patterns, they have been proposed as a generative method to support architectural design processes (Herr & Kvan, 2007) .

L-systems (Lindenmayer, 1968) are methods that simulate the growth of plants. They modify strings of symbols according to a set of rules and constraints. These symbols are further translated into graphics, by considering them as drawing operators that are used to generate a shape. As cellular automata, their ability to produce complex forms from a simple set of rules makes them an interesting tool for designers.

Shape grammars (Stiny & Gips, 1972) consist of geometric production rules that modify shapes by erasing the shape in left hand side of the rule and inserting the right hand side in a certain shape, both conveniently transformed (see Figure 2.1). A set of rules can be used to synthesize shapes by means of successive applications of shape rules to an initial shape or axiom. Shape grammars have been broadly used in design (McCormack et al., 2004) and architectural research (Cagdas, 1996) . The formalism of shape grammars will be further explained in Section 2.2.

Graph grammars (Gips & Stiny, 1980) consist on a set of rules that implement graph transformations. They perform in a similar way than shape grammars, but manipulating graph structures rather than shapes. They replace vertex and edges, and thus an additional process is needed in order to insert the structure of the right hand side of a rule into a given graph. Many computational design approaches that offer support in the early stage of design use graph grammars as generative process (Stanković et al., 2013) .

These methods are thus used to generate design spaces. In order to perform an exploration or search over this state space, we need to identify and formalize a set of design requirements or criteria. In the following section we characterize these criteria and classify the different ways of considering them along with generative systems.

2.1.2.2 Design Criteria

In traditional optimization systems, the word *criteria* refers to a set of numeric variables that need to be maximized or minimized. Here we employ the general concept of criterion, which according to the Cambridge dictionary is defined as a *standard by which you judge, decide about, or deal with something.*

Mandow & De La Cruz (2000) establish a relation between these definitions. According to them, in the beginning of a design process, design standards (that is, general criteria) are expressed in an informal way, becoming more formal as the design advances. The term criteria, in its traditional optimization sense, refers to these design requirements when they are fully formalized.

Design standards can be used to guide exploration through design spaces. In the following we describe the different ways in which we can combine design criteria with generative design systems.

2.1.2.2.1 Criteria Inside the Rules. One of the alternatives is to embed the criteria into the rules of a generative system. If we device these rules with the foreknowledge of the design criteria that the solutions need to meet, then the whole design space produced by such a system will be made up of valid design solutions. This implies hard-coding the design criteria into the rules of the generative system, leading probably to a great number of complex rules that are difficult to create and modify. In addition, we may be sacrificing the diversity of solutions that the system can yield, as the rules have been crafted from the preconceived idea of how the solutions must be.

In the literature we can find examples of application of the aforementioned generative systems by themselves, that is, with the design criteria embedded inside their rules. For example, cellular automata have been used to generate neighbourhoods in an urban scale (Wolfram, 2002). L-systems have been used for image synthesis and modeling (Palubicki et al., 2009; Parish & Müller, 2001). Many researchers have developed shape grammars for particular design problems. Some recent examples are the works of Garcia & Romão (2015) (for designing chairs), Palacz et al. (2015) (for generating grid structures) and Kielarova et al. (2015) (for jewelry design). Graph grammars have also been employed embedding the design criteria inside their rules. For example, Siddique & Rosen (1999) proposed an approach to develop formal representations of product families and apply graph grammars to transform function structures into product structures, and applied it to a coffee-maker product family. Schmidt et al. (2000) propose a general graph grammar for structure synthesis of mechanisms, and apply it to the particular structural synthesis of epicyclic gear trains. Helms & Shea (2012) also employ a graph grammar in order to synthesize mechatronic systems.

2.1.2.2.2 Explicit Criteria. Another alternative is to deal with explicit criteria, that is, by means of a routine that takes a potential design solution as input and returns a quantitative evaluation as output. The generative system could thus be more relaxed and yield a design space with valid and non-valid solutions, and the routine would be used to perform a systematic search in which designs with a poor evaluation would be discarded.

Such an explicit evaluation routine can be used even if the generative system is not automatized, that is, if is a human designer who generates the design space. This is the case of evaluative design systems, rather than generative ones. For example, Kraft & Nagl (2007) developed a prototype software providing a visual knowledge specification language for design. Graph-based domain ontologies define concepts and relations between them, as well as design rules. The designer can use the concepts and relations of this ontology in order to manually instantiate sketches of conceptual buildings, and the created sketch can then be checked against the rule base specified inside the ontology. Pauwels et al. (2011) use Semantic Web technologies in order to formalize rules about building performance. The process of defining and evaluating a design concept would be similar to the one in the work of Kraft & Nagl (2007), using semantic web technologies instead of graph-based techniques. Grabska et al. (2012) developed a prototype software to support architectural design that extracts symbolic information from graphical sketches drawn by the designer. Using this symbolic information, a set of logic rules will check the validity of the sketches drawn by the designer.

Some works follow this approach along with generative design systems. For example, in the work of Agarwal et al. (1999), the intermediate shapes produced by the rules of a shape grammar are measured by means of cost expressions. Bolognini et al. (2007) use rule-based spatial graph transformations in order to synthesize structural, mechanical and mechatronic systems according to a set of design constraints and termination criteria. An algorithm performs a search over a design archive that is constructed by applying the graph transformations. Wyatt et al. (2012) use elementary operations that modify schema configuration graphs, and apply a constraint-based depth-first search in order to find solutions in the context of product architecture design. Grzesiak-Kopec & Ogorzalek (2014b) use a 3D shape grammar, along with design constraints and goals, in order to solve 3D layout design problems.

2.1.2.2.3 Heuristic that Determines How to Apply Rules. Finally, another alternative is to use an heuristic that tells how to apply the rules of a relaxed generative system. Without such heuristic, the system would not yield valid solutions; by contrast, thanks to the heuristic it can generate a space of valid designs. For example, Müller et al. (2006) use shape grammars to create building mass models, and control the application of rules by means of a manually assigned priority. Other works aim at automatically learning a heuristic. For example, Cagan & Mitchell (1993) developed the *shape annealing* method, that guides the rule derivation of a shape grammar according to an energy function. This method can be found in several approaches like the ones of Shea & Cagan (1997), Shea & Cagan (1999b) and Shea & Cagan (1999a), and also combined with graph grammars (Schmidt & Cagan, 1997; Starling & Shea, 2005; Lin et al., 2009). Other examples can be found in the context of evolutionary programming, in which sequences of rules leading to feasible solutions are found via a *fitness* function (Gero et al., 1994; Jin & Li, 2007; Caldas, 2008; Wu et al., 2008). As in the previous alternative, that uses explicit routines for performing a search over the design space, shape annealing and evolutionary programming techniques also imply the definition of an evaluation routine (in the form of energy and fitness functions, respectively), but they are employed to learn an heuristic that guides the application of rules.

2.2 Shape Grammars as a Tool for Computational Design

As we mentioned in the previous section, shape grammars have been widely employed in generative computational design systems. They are a computational formalism for the generation of geometric shapes introduced by Stiny and Gips (Stiny & Gips, 1972; Stiny, 1980, 2006). Informally, shape grammars are a set of rules that govern the composition of simple geometric and symbolic elements to generate complex shapes.

Shape grammars have been used in the literature for many tasks related to architectural or industrial design. In architecture, the works of Stiny & Mitchell (1978) and Cagdas (1996) have managed to devise shape grammars that capture and explain the architectural style of certain buildings, and thus create designs similar to the original ones. The work of Duarte (2005) provides shape grammars that generate designs according to a given housing program. In the context of industrial design, shape grammars have been used to design artefacts taking into account sets of design requirements. Some examples include the grammars proposed by Lee & Tang (2009a) to generate compact digital cameras, considering different combinations of form criteria; the one proposed by Agarwal et al. (1999) for the generation of coffee-makers, taking into account different cost-related aspects; and the one devised by McCormack et al. (2004) to capture a car's brand identity. Shape grammars have also been used in the context of process planning. Brown et al. presented a method for formalising manufacturing information trough shape grammars, providing semantics such that a given derivation of the grammar can be interpreted as a process plan (Brown et al., 1996). More recently, Shea et al. (2010) developed a framework for an autonomous design-to-fabrication system that automatically fabricates customized parts ; in this framework, the shape rules encode primitive movements of the machine tool.

In the following we address the formal aspects needed to understand how shape grammars work. Then, we will discuss the reasons that make shape grammars such a suitable framework for computational design, and we will describe the main kinds of shape grammars, and some existing systems that rely on them in order to produce design solutions.

2.2.1 Formal Aspects of Shape Grammars

A *segment* or *line* $l = \{p_1, p_2\}$ is defined by any pair of distinct points p_1 and p_2, the so-called *end points* of the line. A *shape* is defined by a finite set of distinct *maximal* lines, i.e. lines that are not part of longer lines inside the shape. The representation of a shape is thus unique.

A *labelled shape* σ is an ordered pair $\sigma = \langle s, P \rangle$ where s is a shape and P is a finite set of labelled points. A labelled point (p, A) is a point p with a symbol A. A labelled shape s_1 is a sub-shape of another labelled shape s_2 ($s_1 \leq s_2$) if and only if every line and every labelled point of s_1 is in s_2.

Formally, a shape grammar is a tuple $\langle S, L, R, I \rangle$ where:

- S is a finite set of shapes
- L is a finite set of symbols
- R is a finite set of rules $\alpha \rightarrow \beta$, where α is a non-empty labelled shape and β is a labelled shape
- I is a non-empty labelled shape, called *initial shape* or *axiom*.

A rule $\alpha \rightarrow \beta$ applies to a shape γ when there is a transformation τ such that $\tau(\alpha)$ is a sub-shape of γ. Usually, τ is a general geometric transformation. In this work, transformations involve translations, rotations and reflections.

The result produced by the application of a rule $\alpha \rightarrow \beta$ to a labelled shape γ under transformation τ is given by the expression $\gamma - \tau(\alpha) + \tau(\beta)$. This new labelled shape is obtained substituting some occurrence of $\tau(\alpha)$ inside γ with $\tau(\beta)$.

Figure 2.1 displays a rule and one sample derivation, i.e., a possible sequence of shapes generated by successive applications of the rule to an axiom γ (which in this case is a square). The first rule application detects the left part of the rule (α) inside the axiom, but under a rotation transformation τ. Hence, the right part of the rule (β) is rotated under the same transformation $\tau(\beta)$ and, when the expression $\gamma - \tau(\alpha) + \tau(\beta)$ is performed, the resulting shape is the second one in the derivation shown in Figure 2.1(b). When the rule is applied again, one possibility is that the small square that has appeared due to the first application is now detected, as it is exactly the left part of the rule, but scaled and rotated as well. The third time the rule is applied, the smaller square that has appeared in the third shape is detected under another scaling and rotating transformation, producing the fourth and last shape in our derivation.

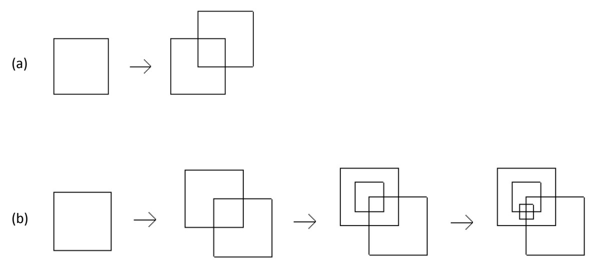


Figure 2.1: A rule (a) and one derivation starting from an initial squared shape (b)

Although this derivation of rules can be performed manually, the advantage of using a computer shape grammar interpreter is obvious: the process is much faster and less error-prone, allowing a quick exploration of the space of designs that a shape grammar can yield.

Chau et al. (2004) compared 21 implementations of shape grammars interpreters up to 2004. More recently, McKay et al. (2012) gathered some of the most relevant interpreters up to 2011. In this last review, the systems are evaluated according to a set of requirements for shape grammar implementations derived from the works of Gips (1999) and Chau et al. (2004). Most of the evaluated tools are generic in the sense that they are not aimed at a particular field of design, and provide graphic means to deal with shape rules. To the best of our knowledge, the most recent contributions in this area are the works of kang Li et al. (2009), Hoisl & Shea (2011), Trescak et al. (2012), Correia et al. (2012) and Grasl & Economou (2013).

2.2.2 Suitability of Shape Grammars to Computational Design

In words of Knight (2015), *"shape computations are based on perception and action. They are about seeing and drawing with basic spatial elements, using one's eyes and hands, to make shapes. From this point of view, shape grammars ally well with contemporary theories of making in the social sciences and humanities"*. In this line, shape grammars are a natural ground for computational design, in which the computer assumes the role of the eyes and hands of the designer.

Also, the wide appeal of shape grammars comes from their versatility. It is well known that they are capable of producing any possible shape (Stiny, 1975). Additionally, this formalism presents specific advantages. They provide an intuitive method for shape definition. They are also *compact*, in the sense that they can generate sophisticated and unexpected designs with just a few rules (Rowe, 1987). As Chase pointed out, design systems based on grammars have a great potential to automate and explore many design alternatives without tedious work (Chase, 2002). These systems can help designers to focus on unexpected designs that otherwise could be easily overlooked. This potential is subject to the presence of a computer in charge of automatically, quickly execute shape grammars, as pencil-and-paper execution might be tedious and therefore useless for the sake of discovering many new solutions.

Early shape grammars succeeded in capturing the essence of different architectural styles and helped to validate the formalism. However, the development of adequate methodologies to create and control the application of shape grammars when trying to achieve design goals remains an open research topic. In particular, the use of shape grammars for architectural or engineering design implies taking into consideration constraints and goals for the designed artefact.

The aforementioned issues of shape grammars can be summarized by means of a

	Helpful	**Harmful**
Internal origin	*Strengths* -Versatility -Intuitive method -Compacity -Able to produce unexpected shapes -They can be automated in order to produce many design alternatives	*Weaknesses* -Creation and control of SG might be difficult when trying to achieve design goals -SG execution needs considerable computational expenses
External origin	*Opportunities* -General improvement in computers' capabilities -Increasing presence of computers in design processes	*Threats* -Reluctance of design practitioners to use SG for their work

Table 2.1: SWOT analysis of the use of shape grammars for design

SWOT (*Strengths, Weaknesses, Opportunities and Threats*) analysis, used to evaluate the venture of choosing the technology of shape grammars for design tasks. A SWOT analysis involves identifying the internal and external factors that are favourable and unfavourable to achieve the objective, that is, the use of shape grammars for design (see Table 2.1).

2.2.3 Types of Human-SG-Design Interaction

In this section we describe two different approaches on how to design shape grammars. According to Knight, "Different approaches to connecting grammars and goals have been suggested. One approach is direct. It involves writing rules with the foreknowledge that the generated designs will meet, or start to meet, given goals. In order to do this, the behaviours and outcomes of rules must be predictable in some way" (Knight, 1999, pag. 7). This approach leads to *expert* shape grammars, designed by a design expert such an architect or industrial designer.

The other approach that Knight points out is an indirect one, in which "grammars are developed without a clear idea of their outcomes. An automated search ant test strategy is then used to explore the space of designs generated, sampling designs and testing them to see if they meet given goals". That is, the solution space yielded by these *naive* shape grammars contains both feasible and infeasible designs, therefore we need an extra control mechanism in order to generate good solutions.

In Section 2.1.2.2 we described the different ways in which design criteria can be combined with generative design systems. The first approach consists on embedding the expert knowledge inside the rules of the generative system. In this context, this is equivalent to expert shape grammars. The second and thirds approaches deal with design criteria in a different manner, leaving them outside the rules and managing them through search processes or by learning how to apply the rules in order to fulfil them. As anticipated by several examples that we provided in Section 2.1.2.2, these alternatives are amenable to be combined with naive shape grammars.

In the following sections we describe the corpus of work developed in both the expert and naive approaches.

2.2.3.1 Expert Shape Grammars

The execution of an expert shape grammar would lead to feasible designs, where feasibility is guaranteed by the structure of the shape grammar itself. We could call these *expert* shape grammars, since they include much expert design knowledge hard-coded inside their rules. There exist many references to expert grammars in the literature. Many works aim to design architectural or engineering objects in accordance with a particular style or brand identity (Stiny & Mitchell, 1978; Cagdas, 1996; Flemming, 1987; Pugliese & Cagan, 2002; McCormack et al., 2004). Figure 2.2 displays a fragment of the Palladian grammar used by Stiny & Mitchell (1978) to generate 2-dimensional plans of villas similar to those of the famous XVI century architect Andrea Palladio (see Figure 2.3).

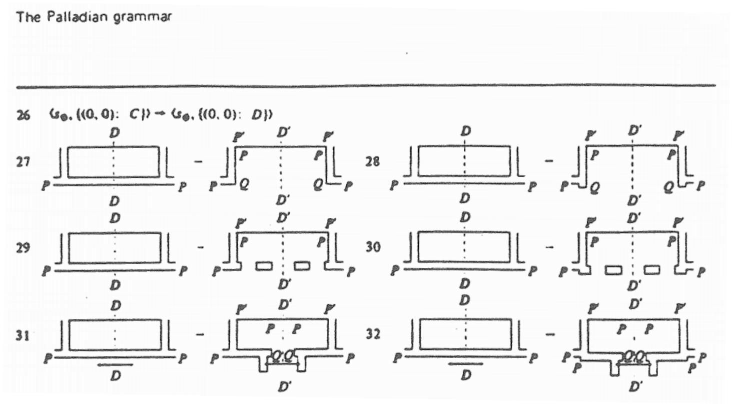

The Palladian grammar
26 $\langle s_\bullet, \{(0,0): C\}\rangle \rightarrow \langle s_\bullet, \{(0,0): D\}\rangle$
27
28
29
30
31
32
D
D'
P
P'
Q

Figure 2.2: Seven rules of the Palladian grammar (reproduced from Stiny & Mitchell (1978))

Some of the methods actually used to include expert knowledge inside shape grammars comprise the following:

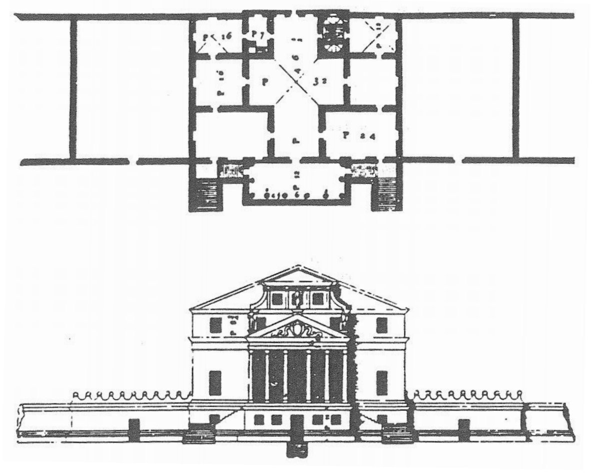


Figure 2.3: The Villa Malcontenta as drawn by Palladio (reproduced from Stiny & Mitchell (1978))

1. Shapes in rules. This is the most direct mechanism, where desired final shapes arise from the accumulation of shapes present in the rules.

2. Control marks. Special labels added to the "real" shapes are typically used to control several aspects of rule execution, such as the order in which rules are applied, the set of rules that can be applied at a certain moment, or the way rules can be applied (i.e., transformations that can be applied to their left-hand side).

A different approach is taken by Duarte in his discursive grammars (Duarte, 2005): each conventional shape grammar rule is accompanied by a description rule that uses extra symbolic information to give semantic to the involved shapes, thus allowing more control possibilities. More recently, the works of Duarte et al. have been formalized by a methodology in which several models are present: (1) a model for the formulation of a design program, (2) a model for generating designs according to the program (which is usually represented by a shape grammar), and (3) a model for evaluating solutions that is used to analyse, compare and rank alternative designs produced by the model used in (2). This methodology is applied, for example, to the generation of urban designs (Duarte et al., 2012).

However, the knowledge engineering effort involved in the creation and modification of expert systems is important. Expert shape grammars, as well as rule-based expert systems in general, can become very difficult to create, modify and maintain. It is generally acknowledged that a "[...]mixture of knowledge types, together with the lack of adequate justifications of the different rules, makes the maintenance of such knowledge bases very difficult and time consuming" (Studer et al., 1998). Some authors have given insight in the creation process of shape grammars. For example, Beirão et al. (2011) show how specific shape grammars can be created from generic ones in the field of urban design. Another approach consists on automating the creation of shape rules by means of statistical analysis (Orsborn et al., 2008) .

Another issue related to the common mechanisms to introduce expert knowledge inside rules is that they promotes the use of deterministic shape grammars that tend to produce certain kind of shapes that are known a priori. A trade-off arises between the ability of a shape grammar to innovate and the feasibility of the produced solutions. In a correctly formulated expert shape grammar, any arbitrary execution of its rules will ideally produce feasible designs, since the shape grammar designer will have predicted in some way the nature of these outcomes. If we sacrifice the divergence capacity of shape grammars in the pursuit of predictability, we are at risk of missing one of the main reasons for using this formalism as a design framework, that is, the possibility of obtaining many unforeseen, innovative solutions.

2.2.3.2 Naive Shape Grammars

A different approach calls for the combination of simpler shape grammars with specific methods to search the design space or guide the generation process. By simpler or *naive* shape grammar we understand that the design requirements are not completely hard-coded inside the rules, and thus an arbitrary execution of these rules, without any additional guiding mechanism, would not guarantee feasible designs. Naive grammars could lead to a high diversity in results, while the guidance methods would guarantee the fulfilment of design requirements. This approach avoids the usual difficulties in creating, modifying and maintaining traditional, expert shape grammars, leaving the guarantees of feasibility mainly in hands of the control mechanism.

A possible alternative is to deal design criteria in an explicit manner, that is, by means of a routine that evaluates design solutions. The shape grammar could thus be more relaxed and yield a design space with valid and non-valid solutions, as the evaluative routine is used to guide the generative process towards good designs. The works of Grzesiak-Kopec & Ogorzalek (2014b); Grzesiak-Kopeć & Ogorzałek (2014); Grzesiak-Kopec & Ogorzalek (2014a) explicitly manage design constraints and goals in order to perform a search along a design space created by a 3D shape grammar. Designs that violate some constraints are discarded, and the ones that comply with them are tested according to the design goals, accepting a given design as a solution if it fulfils every goal. They apply this system to solve design layout problems.

Other works seek to obtain a rule-application heuristic by means of artificial intelligence techniques such as simulated annealing or evolutionary computation.

Simulated annealing (Kirkpatrick et al., 1983) is a stochastic optimization technique that emulates the physical process of metal annealing, i.e. intense heating and then gradual cooling until a low-energy equilibrium state is reached. *Shape annealing* (Cagan & Mitchell, 1993; Shea & Cagan, 1999b) applies simulated annealing to a shape grammar derivation process. In this context, a shape that minimizes the value of a given objective function, interpreted as the *energy* of the shape, is sought. A set of constraints additionally determines valid transitions in the application of the shape grammar rules.

At every step of rule derivation, a rule leading to a feasible new shape is randomly selected. If the new shape obtained applying this rule has lower energy than the current one, then the transition is automatically accepted. However, if the new shape has higher energy, the transition is only accepted with a probability that depends on the energy difference and the *temperature* of the process. Initially, when the temperature is high, the process can easily escape local optima this way. As the process goes on and the temperature is reduced, the chances of escaping deeper (low-energy) optima are reduced. In an infinite process where the temperature asymptotically approaches zero, the probability of being trapped in a global minimum approaches one. In shape annealing, the stochastic sequence of rule applications goes on until the current shape cannot be further improved after a number or trials, or a limit on the number of rule derivations is reached. The algorithm reverses rules if at a certain moment every applicable rule violates any of the constraints. The *temperature profile* that determines how quickly the process cools down is a critical parameter in simulated annealing.

Shape annealing has been applied mainly to structural design problems such as roof truss (Shea & Cagan, 1999b,a) or dome design (Shea & Cagan, 1997). The use of shape annealing can lead to a feasible near-optimal design if the algorithm parameters are correctly configured. Apart from parameter tuning, the underlying knowledge engineering is to a great extent very easy: we just need to compute the objective function and test whether the constraints are violated. One important drawback is that each run of the algorithm leads only to a *single* solution. If we aim to obtain many distinct designs we have to run the optimization algorithm again and again. This approach does not learn *how to* generate designs, it just finds a solution each time it is executed.

Genetic algorithms are an optimization technique that emulates the process of natural selection. An initial population of individuals (*phenotypes*) is represented as strings (*chromosomes* or *genotype*). Individuals are then selected and their strings combined to produce new individuals in a stochastic process that takes into account their fitness to a given objective function (survival of the fittest). Ideally, this iterative process produces populations with good or near-optimal individuals. Gero et al. (1994)

describe two different applications of genetic algorithms to shape grammars. In the first approach (routine design), they seek the execution order (or derivation) of the grammar rules that optimizes a given set of constraints. This can be done using rule sequences as genotypes, and evaluating the fitness of the resulting designs. In the second, more ambitious one, grammar rules themselves are encoded for manipulation by the genetic algorithm. This way, new grammars are produced that could possibly lead to better or innovative designs. Gero & Kazakov (1996) used the technique of *genetic engineering* in order to evolve shape rules by identifying rule sub-sequences that appear in the good individuals of the populations and not in the bad ones. These sub-sequences were used to make up new, complex rules that were combined with a standard genetic algorithm in order to allocate sets of tiles according to given requirements. Ang et al. (2006) also combined a shape grammar and a genetic algorithm in order to design Coca-Cola bottles, by means of parametric rules and by codifying the parameters of the rule sequences inside the genotype. A similar but more complex system is described by Lee & Tang (2009b). It uses genetic programming in order to determine rule parameters for the problem of designing the body of a camera. Chouchoulas (2003) also uses a genetic algorithm to evolve rule sequences in order to produce apartment buildings required to meet certain criteria and O'Neill et al. (2010) employ a genetic algorithm to design shelters.

An advantage offered by all evolution-based approaches is simpler knowledge engineering. Feasibility of resulting designs is enforced by a fitness function that evaluates them according to a set of criteria. Among the downsides we must mention the usual, inherent difficulty of parameter setting.

The set of generated solutions is often restricted in the sense that it has a relatively small size compared to the diversity potential of shape grammars. Some works Lee & Tang (2009b); Ang et al. (2006) offer just a single solution to a given problem (an optimal or near-optimal one), while others Gero et al. (1994); Gero & Kazakov (1996); Chouchoulas (2003) provide few distinct, feasible solutions (less than 10), which are picked from the last generation.

2.3 Reinforcement Learning

As explained in Section 2.2.3, one of the approaches to connecting shape grammars and design goals is to develop naive shape grammars (that is, that do not encode every design requirement inside their rules) and combine them with an extra mechanism in order to search for good designs inside the solution space.

In this thesis, a significant part of the work is devoted to the development of a system that combines these naive shape grammars with a guiding mechanism that learns how to apply the rules. More than exploring and searching the solution space, we propose to discover which rule derivations can produce feasible solutions.

Reinforcement learning (Bertsekas & Tsitsiklis, 1996; Kaelbling et al., 1996; Sutton & Barto, 1998; Wiering & van Otterlo, 2012) is an area of machine learning that deals with how to learn which actions to take in a given environment, in order to maximize a given long-term *reward*. Learning occurs through interaction with the environment, in particular, receiving positive or negative rewards after certain actions. Consider for example a baby learning to build a toy brick tower. S/he will perceive positive or negative rewards when the tower stands or falls respectively. Another example are the successful game playing programs that have acquired master levels in games like Backgammon from self-play. In this case, the only reward received from the environment is the win/draw/lose state after the game is over.

Reinforcement learning has certain advantages over supervised learning. Unlike supervised techniques, reinforcement learning does not need externally provided examples; it just requires interaction with the environment. Such feature can be very useful when it is impractical to obtain representative examples of all the situations in which the agent is involved. Additionally, reinforcement learning allows dealing with uncertainty, making possible to consider more realistic, non-isolated environments. Due to these benefits, reinforcement learning has been successfully applied to many fields like robotics (Piggott & Sattar, 1994; Mataric, 1997; Gaskett et al., 2000; Smart & Pack Kaelbling, 2002; Tedrake et al., 2004; Abbeel et al., 2007; Endo et al., 2008; Erden & Leblebicioglu, 2008; Nemec et al., 2010; Peters et al., 2011; Buchli et al., 2011), games (Tesauro, 1995; Thrun, 1995; Bertsekas & Tsitsiklis, 1996; Baxter et al., 2000; Schraudolph et al., 2001; Levinson & Weber, 2001; Schaeffer et al., 2001; Hu & Wellman, 2003; Droste & Furnkranz, 2008; Gelly & Silver, 2008; Thiery & Scherrer, 2009; Veness et al., 2009; Szita et al., 2010; Mnih et al., 2015), scheduling systems (Zhang & Dietterich, 1995; Aydin & Oztemel, 2000; Wang & Usher, 2005), computer vision (Asada et al., 1996; Peng & Bhanu, 1998; Paletta & Pinz, 2000; Michels et al., 2005), dynamic channel allocation in cellular networks (Singh & Bertsekas, 1997; Lilith & Dogancay, 2004; Senouci & Pujoile, 2003) or medicine (Zhao et al., 2009; Istepanian et al., 2009).

Formally, reinforcement learning techniques solve *Markov Decision Processes* (MDP). A MDP is defined by a set S of states, a set A of actions, a transition function $P : S \times A \times S \to [0,1]$, where $P(s, a, s')$ is the probability of going from s to s' when executing a, and a reward function $R : S \times A \times S \to \mathbb{R}$, where $r = R(s, a, s')$ is the expected immediate reward obtained in such case. Sutton & Barto (1998) classify MDPs as *continuing*, if there is not a special final state and thus the actions continue to be applied ad infinitum, or as *episodic*, if such special final state exists and hence, once it is reached, no more actions are applied and the task is terminated. Episodic tasks can be reformulated as continuing by changing the nature of the final states, converting them into *absorbing* ones. In absorbing states, only one action (that leads to the absorbing state itself) can be applied, yielding a reward of zero. This episodic vs. continuing approach contrasts with the traditional mathematical perspective, in which MDPs are classified as *infinite-horizon*, *indefinite-horizon* or *finite-horizon* tasks. Infinite-horizon tasks are equivalent to continuing MDPs, and indefinite-horizon tasks, in which an indefinite number of actions is taken until a special state is reached, correspond to episodic MDPs. Finite-horizon tasks are MDPs in which the number of taken actions is fixed a priori.

The solution to a reinforcement learning problem is a *policy*, which is basically a rule that indicates which action must be taken at every step. The execution of a policy from a given state s_0 leads probabilistically to a sequence of states $s_0, s_1, \ldots, s_i, \ldots$ For each state s_t in the sequence the *discounted accumulated return* is given by an *additive, expected return* $R_t = \Sigma_{k=0}^{\infty} \gamma^k r_{t+k+1}$. Hence, independently of its immediate reward (which depends solely on the environment), each state has an associated *value*, given by the expected return, that reflects the potential of the state for future rewards, assuming that we will follow the current policy in the future. At the same time, the rational action to take at a given state is the one that leads to a new state with maximum value. Therefore, the policy determines the value of states and vice-versa. An *optimal policy* is one that maximizes the expected value of all states and, in consequence, the expected long-term reward.

A *deterministic* policy is one that, at every step, determines an action to take with probability 1, so for a given state we have only one choice. In addition, if the policy is *stationary*, its outcome only depends on the current state. A deterministic and stationary policy can be seen as a mapping from states to actions, formalized as $\pi : S \to A$. For infinite-horizon MDPs with additive rewards there is always an optimal policy that is stationary and deterministic (Howard, 1960).

A *non-deterministic* or *stochastic* policy is a function $\pi : S \times A \to [0, 1]$ that for each state s and action a specifies the probability of taking a in s. Obviously, for every $s \in S, \sum_{a \in A} \pi(s, a) = 1$. A priori there is no reason for excluding non-deterministic policies, but in some cases they can be considered as unethical (it can be the case, for example, of a policy that determines the way of treating a patient in a clinical context).

If the outcome of a policy also depends on the history of past visited states, then it is *non-stationary.*

There exist three elementary classes of methods for solving a MDP (Sutton & Barto, 1998):

1. *Dynamic programming.* This term gathers a collection of algorithms that can be used when a perfect *model* of the environment is given. By model we understand the transition probabilities between states (that is, the probability of reaching a state s' from a state s following the action a) and the expected rewards of each possible transition. These methods usually *bootstrap*, that is, they update values from other previous value estimations.

2. *Monte Carlo* methods. Unlike the previous approach, they do not assume a perfect model of the environment, so they can learn through experience. They do not bootstrap, so it is necessary to wait until a final outcome (the *accumulated* reward) is available to update values.

3. *Temporal-Difference* learning. TD learning combines ideas from both dynamic programming and Monte Carlo methods: as the former, they learn by bootstrapping, but as the latter, they can also learn from raw experience, without a model of the environment. The simplest TD method is called *TD(0)* and estimates the value function of a given policy. TD algorithms can also learn the optimal policy. For example, *Sarsa* (Sutton & Barto, 1998) is a method that learns the same policy that is used to explore the state space, by improving it along the episodes (it is an *on-policy* method). Another example is *Q-learning* (Watkins, 1989) (see Figure 2.4), which is an *off-policy* method since it learns a policy independent from the one employed to explore the state space. For this reason, Q-learning has minimal convergence requirements, so it has been intensively researched and successfully used in many works (Piggott & Sattar, 1994; Asada et al., 1996; Paletta & Pinz, 2000; Gaskett et al., 2000; Abounadi et al., 2000; Smart & Pack Kaelbling, 2002; Senouci & Pujoile, 2003; Hu & Wellman, 2003; Wang & Usher, 2005; Zhao et al., 2009; Istepanian et al., 2009).

Dynamic programming techniques are of limited utility because the great computational expense introduced by full sweeps over the state space. A sensible option is to implement a model-free learning technique (that is, Monte Carlo or TD), that only sweeps along the states inside the experienced paths. By means of *elegibility traces* we can unify Monte Carlo methods and TD learning, as they allow to control the level of bootstrapping by means of a parameter $\lambda \in [0, 1]$. If $\lambda = 0$, then the method is equivalent to raw TD. If higher values of λ are used, then we reduce the bootstrapping level, bringing the method closer to a Monte Carlo technique.

In this thesis we use a Q(λ) technique. In the following will describe the pure TD approach (Q-learning), and then we will explain the general Q(λ) method.

2.3.1 Q-learning

In Q-learning, each pair (*state*, *action*) is associated with a long-term *value* $Q(s,a)$, and the policy is determined by the following rule: "in state s take the action a that yields a pair (s,a) of maximum value", as given by the following expression,

$$argmax_a Q(s,a)$$

The direct approach is to store Q-values in a table. In Q(0), when moving from state s to state s' through action a, the value $Q(s,a)$ is updated according to the following expression,

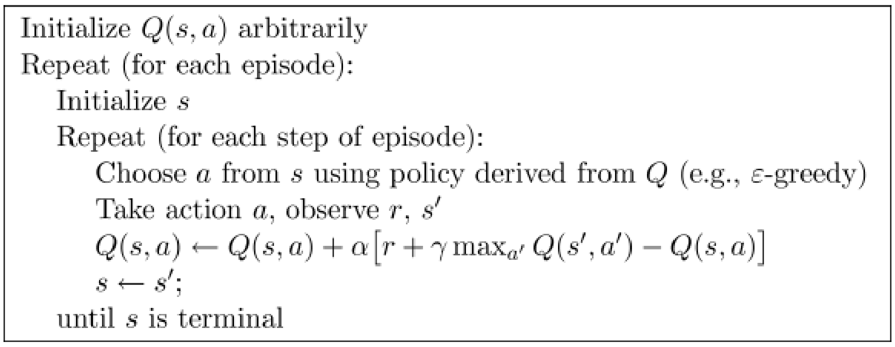
$$Q(s,a) \leftarrow Q(s,a) + \alpha \times \delta$$

where $\alpha \in [0,1]$ is the so-called *learning rate* that determines speed of convergence, and δ is the temporal difference. This is defined as follows,

$$\delta = r + \gamma \times max_{a'} Q(s',a') - Q(s,a)$$

where r is the immediate reward at state s' (if any), $max_{a'} Q(s',a')$ is the maximum value currently achievable at state s', and $\gamma \in [0,1]$ is the so-called *discount rate* that measures how much the agent disregards future rewards in favour of immediate ones. A value of $\gamma = 1$ is usual when long-term future rewards are sought.

As we can see in Figure 2.4, during the learning process it is usual to follow an exploitation-exploration strategy. Most of the time, the algorithm exploits the policy learned so far, that is, at the majority of steps it chooses the best valued action according to the table. However, from time to time, the algorithm chooses a random action in order to explore states that would otherwise never be visited by the current policy. This kind of strategy is called ϵ-greedy, since at each step a random action is



```
Initialize $Q(s,a)$ arbitrarily
Repeat (for each episode):
    Initialize $s$
    Repeat (for each step of episode):
        Choose $a$ from $s$ using policy derived from $Q$ (e.g., $\varepsilon$-greedy)
        Take action $a$, observe $r$, $s'$
        $Q(s,a) \leftarrow Q(s,a) + \alpha\big[r + \gamma \max_{a'} Q(s',a') - Q(s,a)\big]$
        $s \leftarrow s'$;
    until $s$ is terminal
```

Figure 2.4: Q(0) algorithm (reproduced from Sutton & Barto (1998))

chosen with a small probability ϵ.

2.3.2 Elegibility Traces and Q(λ)

In Q(λ), λ refers to the use of elegibility traces. As explained in the book of Sutton & Barto (1998), one of the possible ways to view elegibility traces is to consider them as a bridge between Monte Carlo methods and TD techniques (the *forward* view). When we estimate the values associated to a policy π by means of a Monte Carlo method, backups are based on the full sequence of rewards obtained from the initial state to the final one. On the other hand, if we use TD methods, the backup is based on the obtained reward, and uses the value of the next reached state for estimating the remaining rewards. An intermediate method would backup the value of a state based on an intermediate number of observed rewards.

Another way to view elegibility traces is mechanistically (the *backward* view), which is useful for implementation purposes. The described theoretical view is not directly implementable because at each step we would need to use knowledge of what will happen many steps later, because we use an intermediate number of future rewards to perform the backups, not only the one that has been just obtained. By this backward view we can implement the intermediate methods by means of an incremental mechanism.

This mechanism involves an additional variable associated with each state-action pair: its elegibility trace. On each step t, where the visited state-action pair is (s_t, a_t) the elegibility traces of every pair are updated according to an expression similar to the following one:

$$E_t(s,a) = \begin{cases} \gamma\lambda E_{t-1}(s,a) + 1 & \text{if } s = s_t \wedge a = a_t \\ \gamma\lambda E_{t-1}(s,a) & \text{otherwise} \end{cases} \tag{2.1}$$

Traces measure how recently each state-action pair has been visited, so they determine the extent to which a pair is eligible for undergoing updating changes. The equivalence between the theoretical and mechanistic views is proved In Chapter 7.4 of Sutton & Barto (1998). One method for combining elegibility traces and tabular Q-learning is Watkins's $Q(\lambda)$ (Sutton & Barto, 1998) (Figure 2.5).

2.3.3 Function Approximation

In many practical cases, the number of state-action pairs is so large that cannot be stored in a table. The solution is to use a function approximator to learn the $Q(s,a)$ values. This is done selecting a number of relevant *features* f_i to describe each pair (s,a), and then postulating the $Q(s,a)$ values as a function of those features.

```
Initialize $Q(s,a)$ arbitrarily and $e(s,a)=0$, for all $s,a$
Repeat (for each episode):
    Initialize $s$, $a$
    Repeat (for each step of episode):
        Take action $a$, observe $r$, $s'$
        Choose $a'$ from $s'$ using policy derived from $Q$ (e.g., $\varepsilon$-greedy)
        $a^* \leftarrow \arg\max_b Q(s',b)$ (if $a'$ ties for the max, then $a^* \leftarrow a'$)
        $\delta \leftarrow r + \gamma Q(s',a^*) - Q(s,a)$
        $e(s,a) \leftarrow e(s,a) + 1$
        For all $s,a$:
            $Q(s,a) \leftarrow Q(s,a) + \alpha\delta e(s,a)$
            If $a' = a^*$, then $e(s,a) \leftarrow \gamma\lambda e(s,a)$
                       else $e(s,a) \leftarrow 0$
        $s \leftarrow s'$; $a \leftarrow a'$
    until $s$ is terminal
```

Figure 2.5: Watkins's Q(λ) algorithm (reproduced from Sutton & Barto (1998))

For example, we can postulate a linear function,

$$Q(s,a) = \theta_1 \times f_1(s,a) + \theta_2 \times f_2(s,a) + \cdots + \theta_n \times f_n(s,a)$$

where $f_i(s,a)$ is the i-th feature of the state produced by applying action a to state s, and θ_i is the i-th coefficient of the function Q. The coefficients θ_i of this function are learned at each reinforcement learning step according to the following gradient-descent rule, that takes into account the temporal difference δ,

$$\theta_i \leftarrow \theta_i + \alpha \times \delta \times f_i(s,a)$$

The use of function approximators introduces certain complexity but, at the same time, the use of features to represent states has certain advantages. Many different states can share the same set of features, so when we update the value for a given set of features, we are learning (generalizing) over all similar pairs (*state*, *action*). Thus, if an adequate set of features is selected, proper Q-values can be learned even for states that were never visited during the learning stage.

Function approximation leads to some convergence issues when it is used with a bootstrapping method (Thrun & Schwartz, 1993), especially when it is an off-policy one (Sutton & Barto, 1998). Certainly, this combination can lead to divergence and an infinite mean square error, since it tends to overestimate values. As suggested by Thrun & Schwartz (1993), the use of TD(λ) techniques rather than pure TD ones can reduce the effects of overestimation, since by means of the parameter λ we can control the level of bootstrapping. In addition, according to Thrun & Schwartz (1993),

```
Initialize $\vec{\theta}$ arbitrarily
Repeat (for each episode):
    $\vec{e} = \vec{0}$
    $s, a \leftarrow$ initial state and action of episode
    $\mathcal{F}_a \leftarrow$ set of features present in $s, a$
    Repeat (for each step of episode):
        For all $i \in \mathcal{F}_a$: $e(i) \leftarrow e(i) + 1$
        Take action $a$, observe reward, $r$, and next state, $s$
        $\delta \leftarrow r - \sum_{i \in \mathcal{F}_a} \theta(i)$
        For all $a \in \mathcal{A}(s)$:
            $\mathcal{F}_a \leftarrow$ set of features present in $s, a$
            $Q_a \leftarrow \sum_{i \in \mathcal{F}_a} \theta(i)$
        $\delta \leftarrow \delta + \gamma \max_a Q_a$
        $\vec{\theta} \leftarrow \vec{\theta} + \alpha \delta \vec{e}$
        With probability $1 - \varepsilon$:
            For all $a \in \mathcal{A}(s)$:
                $Q_a \leftarrow \sum_{i \in \mathcal{F}_a} \theta(i)$
            $a \leftarrow \arg\max_a Q_a$
            $\vec{e} \leftarrow \gamma \lambda \vec{e}$
        else
            $a \leftarrow$ a random action $\in \mathcal{A}(s)$
            $\vec{e} \leftarrow 0$
    until $s$ is terminal
```

Figure 2.6: A linear version of Watkins's Q(λ) with binary features (reproduced from Sutton & Barto (1998))

imposing bounds on the discount rate can also be useful. In Figure 2.6 we can see a linear version of Watkins's $Q(\lambda)$ with binary features.

2.3.4 Multi-Objective Reinforcement Learning

Most work in the area of reinforcement learning is aimed to solve MDPs with scalar rewards. However, many problems are best formulated as multicriteria MDPs where rewards are vectors, and components stands for different, possibly conflicting scalar objectives. These are usually referred to as Multiobjective Markov Decision Processes (MOMDPs).

A MOMDP is defined by a set S of states, a set A of actions, a transition function $P : S \times A \times S \to [0, 1]$, where $P(s, a, s')$ is the probability of going from s to s' when executing a, and a reward function $R : S \times A \times S \to \mathbb{R}^n$, where vector $\vec{r} = R(s, a, s')$ is the expected immediate reward obtained is such case. A *policy* is a function $\pi : S \to A$ that selects an action $\pi(s) \in A$ for each state $s \in S$. The execution of a policy from a given state s_0 leads probabilistically to a sequence of states $s_0, s_1, \ldots, s_i, \ldots$ For each state s_t in the sequence the *discounted accumulated return* is given by an *expected vector return* $\vec{R}_t \in \mathbb{R}^n$, $\vec{R}_t = \Sigma_{k=0}^{\infty} \gamma^k \overrightarrow{r}_{t+k+1}$.

Different multicriteria decision paradigms can be applied to reinforcement learning. We can either start from the assumption that there exists a scalarization function than can be used to collapse the returns to a scalar value, or either from the premise that the solution is configured by a Pareto set of optimal policies.

2.3.4.1 Solving MOMDPs by Means of Scalarization Functions

Roijers et al. (2013) propose a MOMDP problem taxonomy based on the first of the aforementioned assumptions, that is, that we can use a scalarization function to aggregate the vector returns into scalar values, either explicit or implicitly. This perspective is what they call the *utility-based* approach, as opposed to the *axiomatic* approach, that consists on directly discovering the Pareto front of optimal policies.

The taxonomy of Roijers et al. (2013) is configured according to three aspects. Firstly, they classify the problems either as *single-policy* or as *multiple policy.* The former approach can be useful when the preferences (or relative weights) for each objective are known a priori. However, there are situations in which such preferences are not known a priori. The solution is then to find a set of so-called *nondominated* policies, i.e. policies that cannot improve one objective without worsening at least another one. Even in this case, scalarization functions can be used to approximate the set of optimal policies, by discovering an optimal solution for each possible weight setting. Secondly, they make a distinction according to whether the obtained policies are allowed to be stochastic or not. Thirdly, they also pay attention to the nature of the employed scalarization function, namely *linear* or *monotonically increasing* if linearity cannot be assumed.

With these three criteria (single vs. multiple policies, deterministic vs. stochastic policies and linear vs. non-linear scalarization function), they propose a MOMDP problem taxonomy in which the nature of the optimal solutions for each kind of problem is characterized (see Figure 2.7).

In Figure 2.7 we can see that the solutions for single-policy approaches always consist on a unique policy. If we use linear scalarization, then the resulting optimal policy is stationary. In addition, the policy is also deterministic, because, as we mentioned, for additive, infinite-horizon single-policy MDPs we can stick to deterministic, stationary policies (Boutilier et al., 1999), so we do not even need stochastic ones.

However, as noticed by Roijers et al. (2013), in the multi-objective setting this does not always hold, and we need non-stationary and stochastic policies. In the

	single policy (known weights)		*multiple policies (unknown weights or decision support)*	
	deterministic	stochastic	deterministic	stochastic
linear scalarization	one deterministic stationary policy (1)		convex coverage set of deterministic stationary policies (2)	
monotonically increasing scalarization	one deterministic non-stationary policy (3)	one mixture policy of two or more deterministic stationary policies (4)	Pareto coverage set of deterministic non-stationary policies (5)	convex coverage set of deterministic stationary policies (6)

Figure 2.7: MOMDP problem taxonomy (reproduced from Roijers et al. (2013))

single-policy approach, when monotonically increasing functions are used and only deterministic policies are allowed, then the optimal solution is a deterministic non-stationary policy, as it can be better than a stationary one (White, 1982). If stochastic policies are allowed, then the resulting policy is in fact a stochastic one, made up from mixing two or more stationary policies.

As stated by Roijers et al. (2013), in the multiple-policy approach, if preferences over objectives can be expressed by means of linear scalarization, we can also stick to deterministic stationary policies (we do not need stochastic ones, even if they are allowed), but now the optimal solution is a set rather than a unique policy. A direct approach is to use a linear scalarization function and repeatedly solve the problem with different sets of weights. The work of Castelletti et al. (2002) is one of the first approaches in this sense, tackling the problem by running a TD algorithm several times with different weights. Natarajan & Tadepalli (2005) deal with the problem in a similar manner, but reusing policies for similar configuration of weights. Other approaches aim to learn the set of policies in a parallel manner, so the weights (and thus the linear scalarization function) are implicit (Barrett & Narayanan, 2008; Hiraoka et al., 2008; Mukai et al., 2012). The set of solutions that can be calculated this way is usually called the set of *supported* solutions (or *convex* set) in multicriteria decision theory. However, in problems where the Pareto front is non-convex, it is well known that in general not all Pareto optimal solutions are supported (Vamplew et al. (2008))

If we use a nonlinear scalarization function then we can capture solutions that are missed in the linear setting. According to the taxonomy of Roijers et al. (2013), if only deterministic policies are allowed, then the solution is the whole set of non-dominated policies (that is, the Pareto coverage set), and the policies in the set need to be non-stationary, as they can dominate stationary ones. If stochastic policies are allowed, then is enough to compute a convex coverage set of deterministic and stationary policies, since we can obtain the rest of non-dominated policies by mixing policies of this set of supported solutions. A direct approach is to explicitly use a non-linear scalarization function and, as in the linear case, repeatedly solve the problem with different sets of weights. For example, any Pareto-optimal solution can be optimal with respect to the Chebyshev norm given the adequate parameters. Some approaches using nonlinear scalarization functions in reinforcement learning can be found in the works of Shelton (2001), Vamplew et al. (2011) and Van Moffaert et al. (2013b). Nonetheless, an explicit nonlinear scalarization function might be hard to obtain. Moreover, in the particular case of the Chebyshev norm, not all optimal solutions with respect to this scalarization funtion are necessarily Pareto-optimal. Another alternative is trying to estimate sets of Pareto-optimal policies simultaneously, and thus the scalarization function would be implicit. One example is the work of Handa (2009), that combines ideas of conditional random fields, clustering, and evolutionary multi-objective optimization in a new algorithm (EDA-RL). The new method was shown to simultaneously acquire two different policies when applied to a sample MORL problem.

2.3.4.2 Solving MOMDPs by Discovering Pareto Fronts

In contrast, if we start from the premise that the solution to a multi-objective MDP is the Pareto front of policies, then techniques that directly aim to find this set must be sought. In fact, we can see the aforementioned work of Handa (2009) as an approach fitting in this *axiomatic* perspective.

Another example fitting in this approach is the work of Van Moffaert & Nowé (2014). They developed an algorithm that discovers the whole set of Pareto-optimal policies in a parallel manner, by separating the expected immediate reward from the set of expected future rewards vectors, allowing them to converge separately. The sets of policies for every (state, action) pair can be produced at execution time, performing a vector-sum of the learned expected immediate reward and the set of expected future rewards vectors. Their method can be seen as a TD extension of the multi-objective dynamic programming technique developed by White (1982), which discovered the Pareto front of non-stationary policies for an infinite-horizon MOMDP.

Chapter 3

Control, Synthesis and Criteria

Can the computer substitute for the designer?
Probably, in some special cases, but usually
the computer is an aid to the designer

Question: Madame L'Amic. Answer: Charles Eames

In this chapter we propose a classification scheme for possible approaches to computational design based on a particular generative system, and we use it as a framework to organize and introduce the different contributions of this thesis.

In Section 2.1.2.2 we described different alternatives to combine design criteria with generative design systems. Design criteria can be (1) hard-coded inside the rules, (2) explicitly used to evaluate potential solutions and perform a systematic search of the design space or (3) used to discover a heuristic that determines how to apply the generative rules of the system. In the context of shape grammars, each alternative leads to a particular kind of shape rules, as we explained in Section 2.2.3: hard-coding design requirements inside shape rules yields the so-called expert shape grammars, while control techniques that leave the design criteria outside the rules are naturally combined with naive shape grammars. However, this separation between expert and naive grammars is not sharp, as we can establish different shades of *naiveness*: the more design requirements are not considered in the process of designing a shape grammar, the more the generated shapes (by the grammar itself) will divert from the ideal solutions, and thus the more naive the shape grammar can be considered. Regarding the external control processes that can be combined with generative systems, the wide spectrum of techniques described along the previous chapter (from direct search techniques to more complex methods like simulated annealing or genetic algorithms) also suggests a classification of these systems based on their sophistication. Although

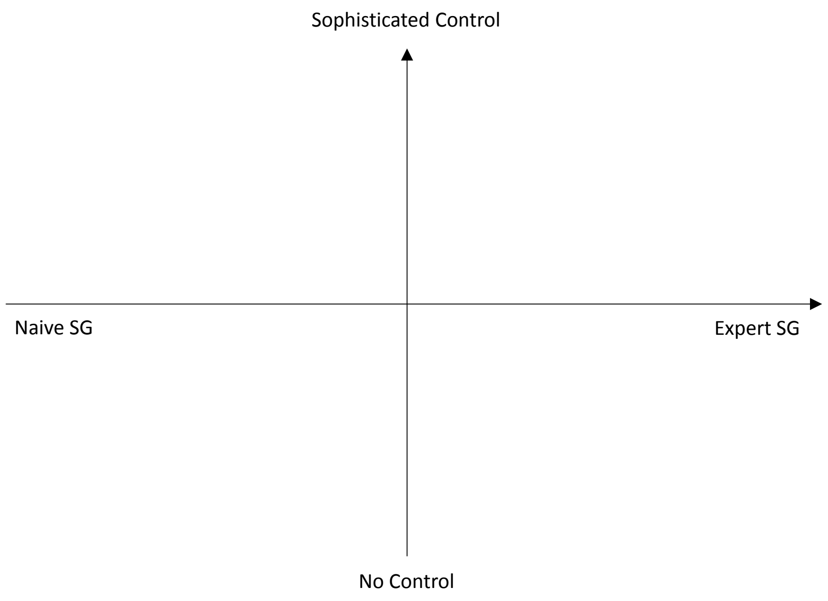


Figure 3.1: Axes for classifying different approaches combining shape grammars and control

the complexity of a control method is an elusive measure, it is useful for comparing the different approaches to computational design that will be described here.

In this thesis we explore different approaches that combine shape grammars and control methods used to generate good design solutions. We propose to use the two dimensions outlined in the previous paragraph in order to classify the different approaches, that is: the naiveness of the involved grammars and the sophistication of the chosen control method. We can represent this classification scheme in a graphic manner, with the help of two perpendicular axes that yield four sections, as seen in Figure 3.1. In this figure, the horizontal axis stands for the naiveness or expertness of the shape grammar(s) involved in a given system. The left end represents a totally naive shape grammar, that is, a grammar designed without considering design requirements at all. The more expert a shape grammar is, the more is located to the right, being the right end the location for pure expert shape grammars. The vertical axis stands for the complexity of the control method used in combination with the shape grammar, being the bottom end representative of non-existent guidance methods, and the top end for sophisticated control mechanisms.

Design systems based on pure expert shape grammars do not need an additional control method, as the results produced by this kind of grammars are always valid. Hence, in our classification scheme, pure expert grammars would be allocated in the bottom-right corner, as seen in Figure 3.2.

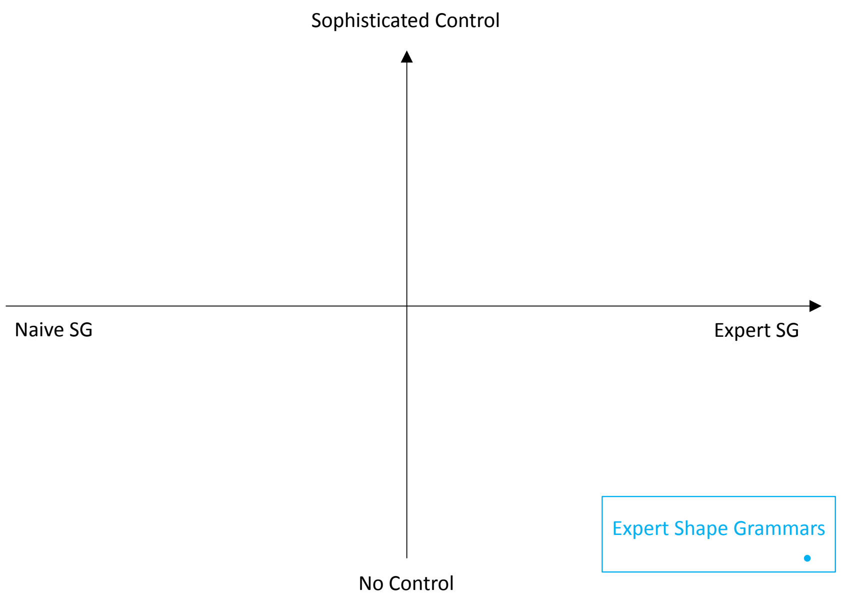


Figure 3.2: Location of a design system based on a pure expert shape grammar

As we explained in Section 2.2.3, one of the main concerns about pure expert shape grammars is that they are very difficult to create and modify. Another one is their tendency to produce the same type of shapes, as the grammar designer has to foresee the outcome of the possible derivations, leading to a predictability that, while assuring the correctness of the obtained shapes, hinders diversity and the possibility of unexpected solutions.

Other design systems based on more naive shape grammars can cope with these matters, at the expense of the validity of the produced shapes. To deal with this issue they need some kind of guidance method for controlling the rule application, so these systems would fall in some point located at the left of and above the point representing a pure expert shape grammar. In this thesis we propose two different computational design systems that use not purely expert shape grammars. They are introduced in the next two sections, and will be described in detail trough the rest of the thesis. In addition, in Section 3.3 we briefly describe an approach in which naive shape rules are not combined with an additional control method. Such system was developed in order to study some aspects related to randomness in design processes.

3.1 Expert SGs and Sophisticated Control

Shape grammars can be designed with certain level of expertness, in the sense that they can encode some, but not all, of the design requirements that solutions need to fulfil. Reducing the amount of design requirements that have to be considered inside the shape grammars can help to deal with the aforementioned issues related with the use of totally expert shape grammars.

A possible approach is to divide the design requirements into two categories, one for design criteria that are easier to specify geometrically (for example, the necessity of certain shapes to be present in the solutions) and another one for design requirements that are suitable to be expressed in an explicit, symbolic manner (for example, requirements about the distance between two particular elements in the design). On one hand, the design demands falling in the first category can be naturally declared inside the shape rules. On the other hand, the needs belonging to the second group are managed in an external search process, performing an automated test over the derived shapes in order to guide the generation towards feasible solutions.

In order to perform this test, we propose to use logic predicates that represent design requirements. This predicates will be divided into *constraints* and *goals*, depending on how they are managed during the search process. Every time a rule is applied, and thus a new shape is yielded, each constraint and goal is evaluated over it, obtaining *true* if it is fulfilled and *false* if it is not. The system discards those shapes that violate any of the constraints and returns a solution when, in addition to not violating the constraints, the generated shape complies every design goal. The details of the particular search algorithm used in the system will be described in Chapter 4.

We also propose to enhance the shape grammars with *layers*, which are a very common structuring instrument in CAD tools. In our context, they can help grammar designers in the creation process, as they allow for a better visualization and a cleaner legibility of the rules. Certainly, a shape grammar designer can group different geometric elements in distinct layers, according to what they represent (for example, if the shape grammar is intended to produce house floor plans, a layer can be used for the walls, another for the doors, etcetera). The definition of constraints and goals can also benefit from this structuring device, as we can describe predicates that only refer to the geometric elements in one particular layer (for example, if we want to impose some constraints on door measures, then the associated predicate only has to check the elements inside the *door* layer).

This system based on layered shape grammars, constraints and goals would be located in the top-right quadrant of our classification scheme (Figure 3.3), and will be described in detail in Chapter 4.

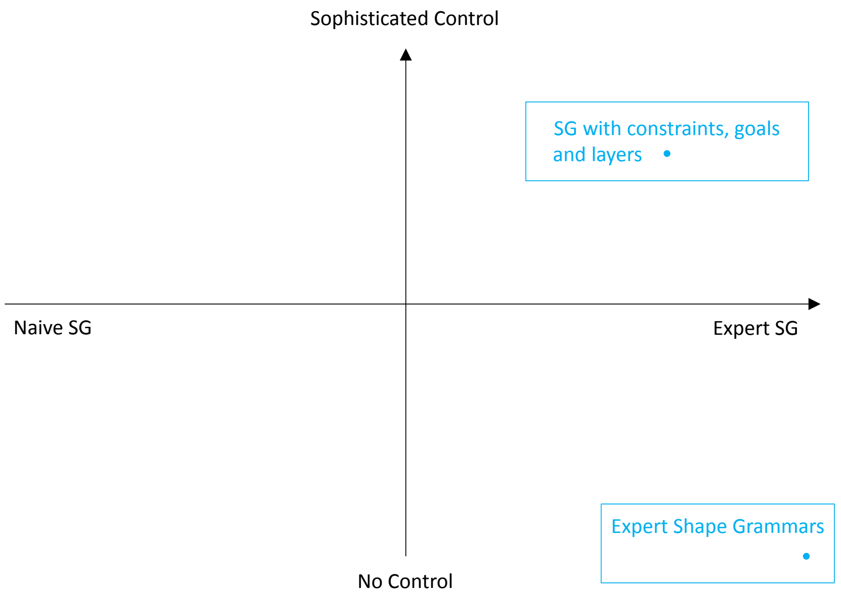


Figure 3.3: Location of our system based on shape grammars, constraints, goals and layers

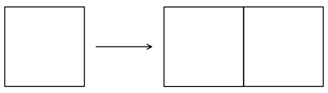


Figure 3.4: A simple additive rule working with squared tiles

3.2 Naive SGs and Sophisticated Control

The level of expertness hard-coded inside the shape grammars can also be minimized. One way of achieving this is by drawing inside them just atomic elements that will be aggregated to make up the solutions. For example, we can work with an additive rule that puts an squared tile next to an existing one (Figure 3.4). The benefits of this approach are the utter ease for designing the shape grammars, and the great diversity and non-expected solutions that can be obtained.

As opposed to the previous approach, in which the shape rules encoded some design requirements, now it is not enough to just *manage* criteria. The state space is now much bigger, as the rules just deal with atomic elements that can yield a lot of different shapes, being the majority of them infeasible from the point of view of the considered design problem. If, as in the previous section, we just implement a direct search process that discards non-valid solutions and keeps on searching until a feasible solution is reached, the time requirements would be huge, not only due to the size of the state space, but also because the test for determining if a shape is valid would be now much more complex, as it would have to consider every existent design requirement, and not a subset as in the approach of Section 3.1.

Here we propose an approach in which an algorithm *learns* how to apply the shape grammar, determining the more suitable rule to be applied at each derivation step. This is managed through a reinforcement learning process in which the design requirements are codified as rewards. The system learns by experience, reinforcing those rule applications that lead to a high reward, and punishing those that yield a low one. If scalar techniques are used, a generalization mechanism can also integrated with the learning algorithm, so it is not necessary to explore the whole state-action space. As we explained in Section 2.3, this is accomplished by means of features that represent relevant characteristics of a shape. Hence, two different shapes can share the same set of features, meaning that they are similar and can thus be treated equally (of course, starting from the premise that the features are well chosen).

In this system we have to clearly distinguish the learning process from the derivation process. During the first one, a learning algorithm determines which features must be pursued (or, equivalently, which rule has to be applied) at each derivation step, in order to obtain high rewards. As we mentioned before, for computing the rewards we need to explicitly use the design requirements. However, during the derivation process we do not use the rewards, but the learned priority of rules. Hence, the design requirements themselves are only needed in the learning process, in order to discover the criteria (that is, the priority of rules) that will be used during the derivation of the shape grammar. (Of course, this does not prevent us from still using design requirements in the definition of features for the shapes, but it is not necessary, as we could have *naive* features that only refer to basic geometric properties of the shape). This method contrasts with the one introduced in the previous section, in which the design requirements are directly managed during the derivation process.

This approach based on naive shape grammars and reinforcement learning would be located in the top-left quadrant of our classification scheme (Figure 3.5), and will be described in detail in Chapter 5.

3.3 Naive SGs Without Control

In the context of computational design systems based on shape grammars, if the rules are naive then we need an extra control method in order to generate feasible solutions. However, using naive rules in a random setting can still be of certain theoretical interest. Certainly, the use of randomness in design has been considered and analyzed by some researchers, usually by means of case studies (Goldschmidt, 1994) or computer

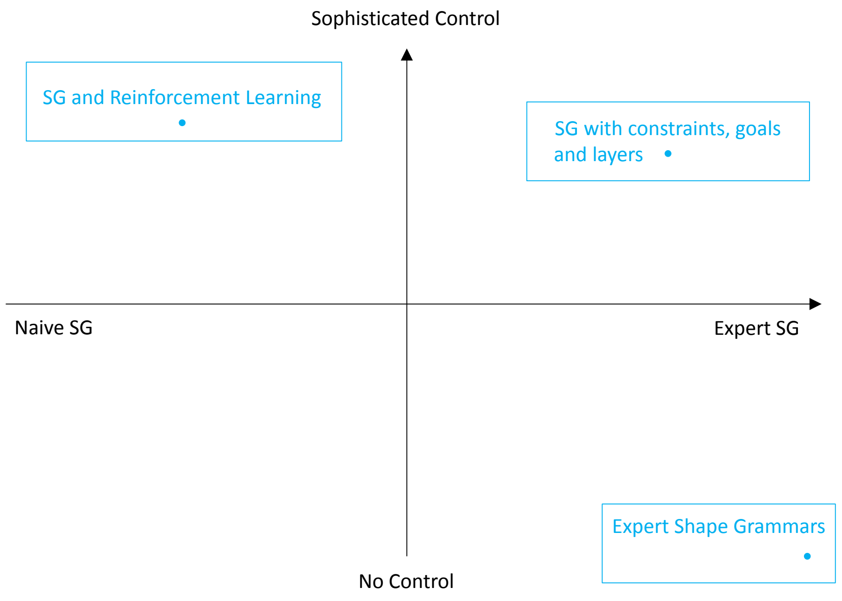


Figure 3.5: Location of our system based on shape grammars and reinforcement learning

based simulations (Petre et al., 2006). Randomness can help designers in the initial stages of the process or even can be present in the final result for the sake of visual pleasure (Verbeeck, 2006, p. 73).

Along these lines, it could be interesting to carry out a study of what happens when naive shape grammars are used without an additional control mechanism, letting the users apply the rules randomly and also providing means for manually derivating rules (Belmonte et al., 2014). This system would be located in the bottom-left quadrant of our classification scheme, as it lacks a control method (see Figure 3.6). The results of such research could be of interest when implementing new computational design tools.

3.3.1 An Experiment on Naive SG Without Control

In this experiment we tried to answer the following research questions:

1. To what extent novice designers are willing to perform random explorations in a design space?

2. To what extent do they prefer random to controlled results?

3. Can “good” designs be obtained by means of partially random processes?

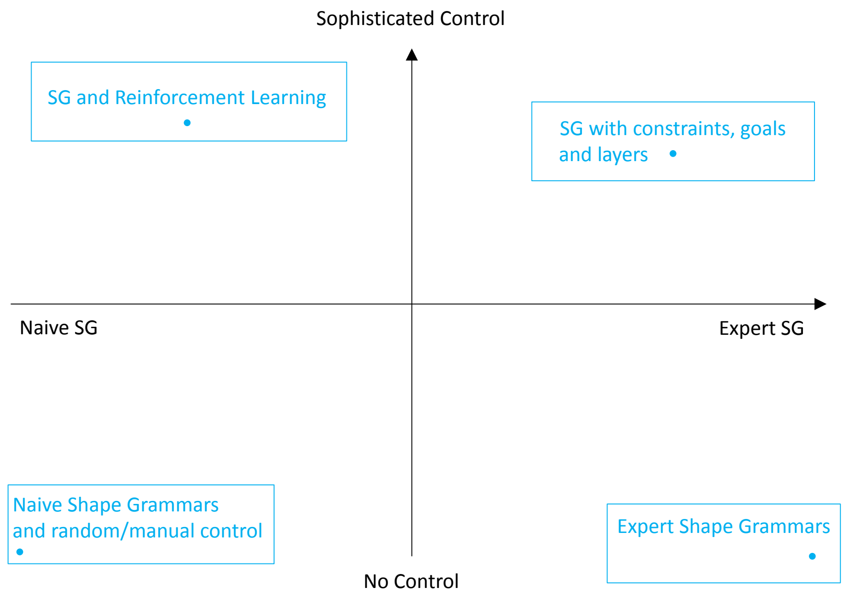

Sophisticated Control
SG and Reinforcement Learning
SG with constraints, goals and layers
Naive SG
Expert SG
Naive Shape Grammars and random/manual control
Expert Shape Grammars
No Control

Figure 3.6: Location of our system based on naive shape grammars and random/manual control

To this end, an interactive computer tool was implemented and offered to a group of 60 fourth-year students of Architecture. Students had to use the tool to design mega-structures, and their outcomes were analysed in terms of randomness and quality.

The tool, named `MG-Shade`, was developed specifically for this study in order to interactively generate mega-structures by aggregating 3-dimensional cells. A complete user's guide for `MG-Shade` is available on the web[1]. In Figure 3.7 we can see a screen shot of the graphical interface of `MG-Shade`.

Each student had to sketch six projects and had to write a report documenting the creation process. In figures 3.8 and 3.9 we can see examples of two megastructures, one created by manually controlling the rule applications, and another one randomly generated, respectively.

From his/her six sketches, each student had to select the "best" one, giving the reasons for their choice. Then the instructor selected the ten best students, according to feasibility and degree of detail of their projects, among other criteria. Independently of this assessment, each of the projects was inspected and graded according to its degree of randomness. Grading was performed by a team of two judges that should agree on a common score. Judges took into account both the design process, as described by students in their reports, and the final look of generated sketches. The scale was as

[1]`http://www.lcc.uma.es/~perez/ntidapa/`

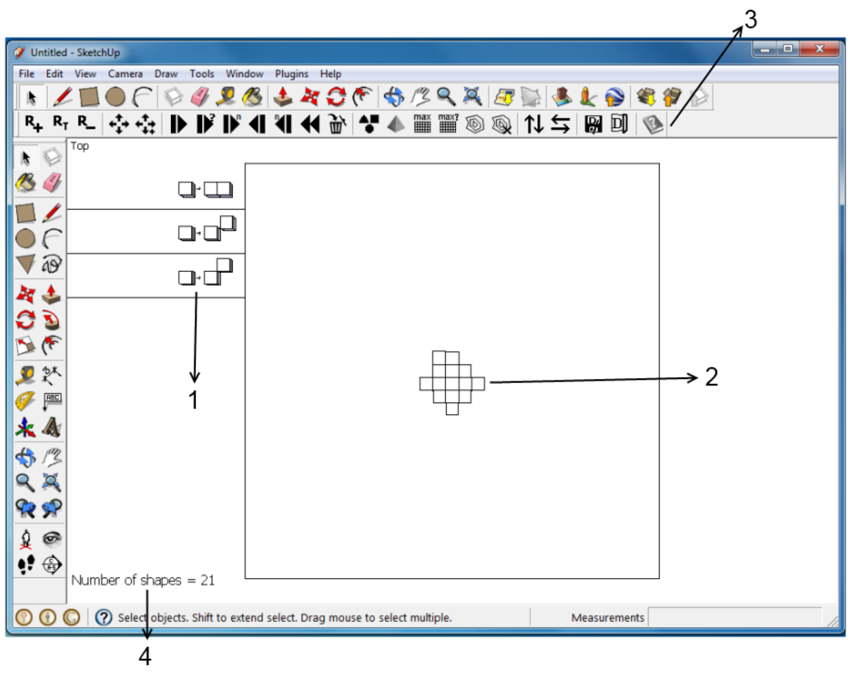

Untitled - SketchUp
File Edit View Camera Draw Tools Window Plugins Help
Top
Number of shapes = 21
Select objects. Shift to extend select. Drag mouse to select multiple.
Measurements
1
2
3
4

Figure 3.7: Graphical interface of `MG-Shade`

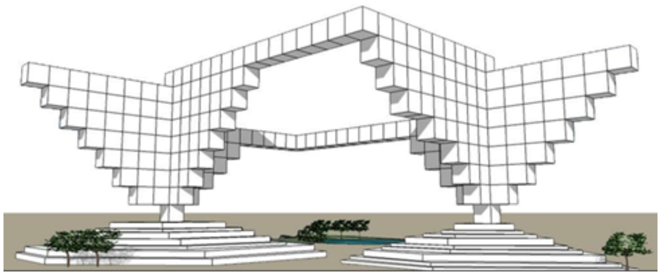


Figure 3.8: A controlled mega-structure generated by a student

follows:

1 (Almost) total control
2 More controlled that random
3 More random than controlled
4 (Almost) total randomness

Hereafter, the consensus score of each project will be referred as the randomness value (r). Figure 3.8 displays a project that has been totally controlled ($r = 1$), as shown both by the written report and the visual inspection. In contrast, Figure 3.9 displays a totally random project ($r = 4$).

The global average value for r was 2.53. The mode is $r = 3$, since 32.49% of the designs were produced for this value.That means that 32.49% of the designs were generated following a more random rather than controlled process. Moreover, the vast

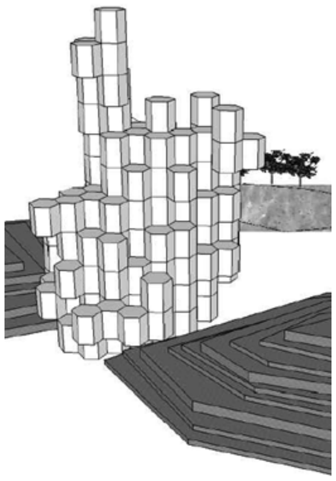


Figure 3.9: A random mega-structure generated by a student

majority of the designs (80.67% = 28.85% + 32.49% + 19.33%) were produced using randomness to a certain extent (r = 2, 3 or 4). This shows that most participants in this experiment were willing to explore the design space by applying randomness, confirming some previous results reported in the literature.

We also came to the conclusion that if the student generates a totally controlled design, then it will be preferred one in three times. On the contrary, if s/he generates a totally random or more random than controlled design, then it will be preferred approximately one in ten times.

With respect to the analysis of randomness in "good" designs, we considered that "good" designs were those generated by students selected by the instructor, and specifically the preferred ones. Average value for "good" designs is $r = 2.07$ and the mode is $r = 1$ with a frequency of 40.00%. On the other hand, the sum of the rest of frequencies, standing for designs in which randomness has been present to a certain extent, is higher (60% = 28.33% + 16.67% + 15%); so the majority of "good" designs present a certain degree of randomness. Hence, another conclusion was that a certain degree of randomness was present in many of the "good" designs, but the weight of randomness in this subset is somehow reduced with respect to the whole set of designs.

To conclude, we can state that, while exploring randomness, students must take into account functional and constructive considerations, and surely that is the reason why they have chosen to develop designs that are not the most random ones. Users seem to be open to explore randomness, so the tool should provide a certain amount of that; however, the tool should also allow to control and tame such randomness.

This experiment, located in the bottom-left quadrant of our classification scheme, suggests that users are willing to consider suggestions provided by the computer, and also that in order to lead to good designs, these suggestions should not be totally aleatory, but controlled in some manner. In the following sections we will focus on the approaches falling on the top-right and top-left quadrants, where more or less sophisticated forms of controlling the chosen generative system (namely shape grammars) are used.

Chapter 4

A Computational Design System for Managing Criteria in Design Problems

Does the creation of design admit constraint?
Design depends largely on constraints

Question: Madame L'Amic. Answer: Charles Eames

In this chapter we describe in detail the system advanced in Section 3.1, based on the idea of encoding some of the design requirements into rules while leaving others to be managed in an external search process employing logic predicates (Ruiz-Montiel et al., 2011b, 2014). As far as we know, this approach was the first one guiding the application of generative rules with logic predicates. The posterior works of Grzesiak-Kopec & Ogorzalek (2014b); Grzesiak-Kopeć & Ogorzałek (2014); Grzesiak-Kopec & Ogorzalek (2014a) also employ rules combined with predicates, in the context of layout design.

A predicate is either a constraint or a goal. The search process will discard shapes that violate some constraint, and will return a solution when a shape fulfils every goal. Two are the reasons that justify the separation between design criteria that are codified inside the rules and those that are expressed as predicates. Firstly, when dealing with design problems, we can find design requirements of different nature, and it becomes clear that some of these criteria are easier to specify by means of their codification into visual rules, while others adapt better to symbolic mechanisms. The designer should be allowed to express design criteria both visually, in the form of shape rules, and also explicitly, in the form of design constraints and goals. Secondly, as the amount of design requirements to be considered inside the rules is reduced, the shape grammars are easier to create and maintain, and they can produce more varied solutions than a pure expert shape grammar.

In addition, the system allows the use of layers in order to design and visualize rules. This structuring device can help shape grammar designers in the rule creation process, as they can take advantage of the layer facilities provided by CAD tools. An additional advantage is that layers decrease the time requirements of the subshape recognition algorithm, as we can have several simple shapes separated in distinct layers instead of a complex one inside a single layer.

In Section 4.1 we explain in detail the search process used to explore the design space, based on design constraints and goals, and in Section 4.2 we introduce the concept of layered shape grammars. In Section 4.3 we describe a software tool that integrates layered shape grammars with the search process. We will illustrate how the system works by means of two examples (sections 4.4 and 4.5). Finally, in Section 4.6 we discuss the performance of the system from the the point of view of time.

4.1 Constraints and Goals

Our system considers logic predicates during the execution of the shape grammar. Logic predicates are boolean-valued functions that return `true` or `false` depending on whether the statement they represent is satisfied or not. We consider two kinds of logic predicates:

1. Constraints: predicates that must be satisfied at every step of any rule derivation.
2. Goals: predicates that must be satisfied at the end of any rule derivation. They control termination. The design is considered satisfactory as soon as goals are attained. In such case, the derivation sequence can terminate.

The space of shapes that can be generated through the repeated application of a shape grammar to an initial shape is a directed, possibly cyclic graph with shapes at its nodes. A link connecting a shape s_1 to a shape s_2 indicates that s_2 can be obtained by applying some rule, with a concrete transformation, to s_1. There are many links from each node as possible pairs (*rule*, *transformation*) are applicable to the corresponding shape. There are two special kinds of nodes:

- Goal nodes: nodes representing a shape that fulfils all the goals, without violating any constraint.
- Terminal nodes: nodes representing a shape that either does not allow more rule applications (due to its geometry), or violates some of the current constraints.

```
Algorithm: Depth-first search (depthFirstSearch)
Input: an initial shape $s$
Output: A shape $\gamma$ (if exists) that fulfils all the goals and does not violate any constraint
begin
    if isGoalNode($s$)
        result = $s$
    else if isTerminalNode($s$)
        result = failure
    else
        children = getChildren($s$)
        found = false
        while (notEmpty(children) and not found)
            child = takeChild(children)
            solution = depthFirstSearch(child)
            if isShape(solution)
                result = solution
                found = true
            end
        end
        if not found
            result = failure
        end
    end
    return result
end
```

Table 4.1: Algorithm for depth-first search

The process of generating a shape that complies with a given set of predicates can thus be formulated as a search over this graph. In order to perform such search we could use any search algorithm. Here we use a *depth-first* search due to its low memory requirements and the possibility of imposing a limit on the reached depth along the search process. Table 4.1 shows the pseudo code for a recursive algorithm that performs depth-first search. The process starts at the root node of the derivation tree (see Figure 4.1), checking if it is already a goal node (with the method `isGoalNode`) or a terminal one (with the method `isTerminalNode`). Otherwise it will explore the derivation tree, descending as deep as possible along each branch before backtracking, until it finds a goal node. In case that no goal nodes are found, the algorithm will return *failure*. The method `getChildren` computes all the possible shapes that can be obtained from a given shape s, applying the rules of the shape grammar. The method `takeChild` takes a node of a given list and removes it from the collection. If the result of the recursive call is a shape, then the algorithm has found a solution. Otherwise, the result is **failure** and there does not exist any solution in that branch.

Note that constraints prune the search tree, effectively reducing the space that needs to be explored.

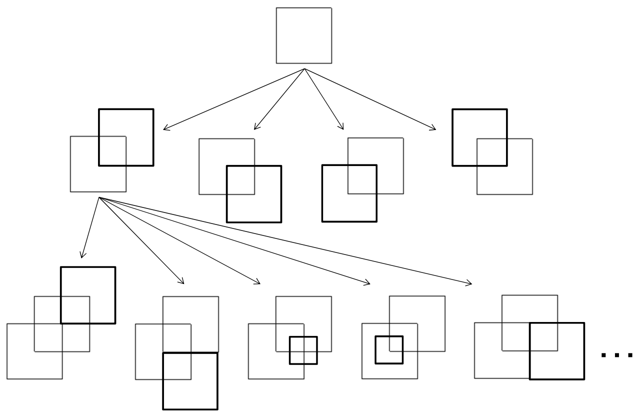


Figure 4.1: Fragment of the derivation tree for the application of the rule in Figure 2.1

Let us illustrate the use of logic predicates with a simple example over the tree sketched in Figure 4.1. Suppose that we need to produce shapes with the rule in Figure 2.1, satisfying two design requirements:

1. The shape must not contain segments larger than one meter.

2. The shape must contain at least ten squares.

The user/designer has to formalize each requirement as a constraint or goal. Let us formalize the first one as a constraint and the second one as a goal. With these premises, a possible path the algorithm may follow through the tree is depicted in Figure 4.2. At the first level, the axiom (which is a one meter side square) does not violate the constraints but does not meet the goal, so the algorithm goes deeper. The same happens in the second level. In the third level, the algorithm now finds two nodes with shapes that violate the constraint, and are marked as terminal. The third shape in this level does not violate the constraints but still does not attain the goal, so the algorithm explores this node, finding a solution in the fourth level.

As we have seen in this example, some requirements are specified in terms of exact metric (it is the case of the constraint, stating that the shape must not contain segments larger than one meter). Although shape grammars involve dimensionless rules in the sense that the size of the pattern to be found (that is, the left part of the rule) does not matter, these rules apply to real shapes that can in fact have dimensions, as is the case of the axiom in the previous example. Since the requirements always refer to measurable shapes, there is no conflict in establishing predicates involving exact metric.

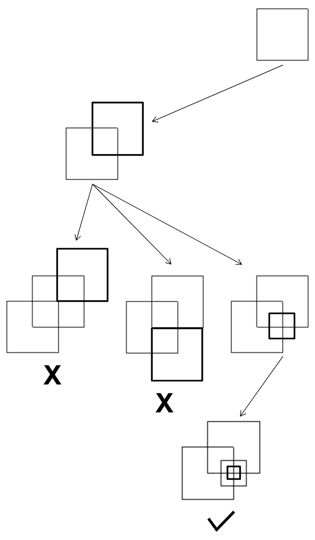


Figure 4.2: Depth-first search of the tree of rule in Figure 2.1, with the specified predicates.

Constraints need to be defined with care. When a constraint is violated, the depth-first search algorithm prunes all the paths that could be followed from the offending node. If there were a valid solution inside some of those paths, the algorithm would never reach it. This could happen, for example, in the example depicted in Figure 4.2, if the grammar had an additional rule that removed squares. As a general guideline, a predicate can be formulated as a constraint when we are sure that once it is violated, there is no possible derivation that will satisfy it again. Otherwise, it is better to define the predicate as a goal. If the grammar contained rules that remove shapes, this property would not be present.

4.2 Layers

Layers are a common structuring device in practical CAD tools. From a theoretical point of view, experts acknowledge the need of using *partial descriptions* of a design. Partial descriptions of designs have to be *superimposed* in order to make up the whole picture. A design is composed of several partial descriptions that have no meaning in themselves, that is, they are fragments of the design. They have to interact and overlap in order to form the whole meaning. Kotsopoulos, starting from the theoretical background of *product shape algebras* (Stiny, 1992), introduced this notion as a thinking-graphic device when working with shape grammars (Kotsopoulos, 2005). Stiny also introduced the concept of multiple *tuples* (or *channels*, as Yue and Krishnamurti explain in a recent work (Yue & Krishnamurti, 2013)) in shape grammars as a technique to deal with the difficulty of distinguishing segments embedded in the maximal lines (Stiny, 1975). Here we provide an implementation for these ideas and define a new subshape detection algorithm for layered shape grammars.

As we explained in the introduction of this chapter, layers allow shape grammar designers to take advantage of the layer facilities provided by CAD tools. Firstly, layers provide a cleaner visualization mode in which certain layers can be displayed or hidden in the canvas, focusing the designer's attention on the particular segments and points that need to be dealt with at any given time. Secondly, legibility of the shape grammar can also be improved thanks to the aggregation of related segments and points into a single layer with a meaningful name. An unlayered shape grammar, in contrast, would hinder the interpretation of each segment or point. For example, a door-related segment might be mistaken for a wall, hampering further modifications of the shape grammar.

4.2.1 Formalization

Formally, we define a layered shape grammar as a 5-tuple $\langle S, L, R, I, La \rangle$ where:

- S is a finite set of shapes
- L is a finite set of symbols
- R is a finite set of rules $\alpha \rightarrow \beta$, where α is a non-empty layered shape and β is a layered shape
- I is a non-empty layered shape, called *initial shape* or *axiom*.
- La is a finite set of n layer names, $(La_0, La_1, \cdots, La_{(n-1)})$

A layered shape is a set of n labelled shapes $(\sigma_{La_0}, \sigma_{La_1}, \cdots, \sigma_{La_{(n-1)}})$. That is, there is a labelled shape defined for each layer. In Figure 4.3 we can see an example of a layered shape (the bottom-left cross represents the origin of coordinates).

A rule applies to a layered shape γ when there is a transformation τ such that $\tau(\alpha_{La_i})$ is a sub-shape of γ_{La_i}, that is, $\tau(\alpha_{La_i}) \leq \gamma_{La_i}$, for every layer La_i. Note that the applied transformation τ must be the same in every layer. The rest of the arithmetic for labelled shapes is naturally extended to layered labelled shapes in a similar way, applying the operators in parallel to the shapes of every layer.

In Figure 4.4 we can see a layered shape grammar with the content of each layer shown separately. The grammar is composed of three rules that are divided into three layers.

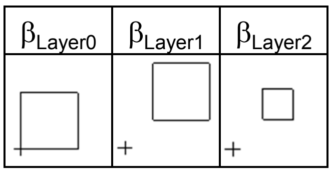


Figure 4.3: A layered shape ($n = 3$)

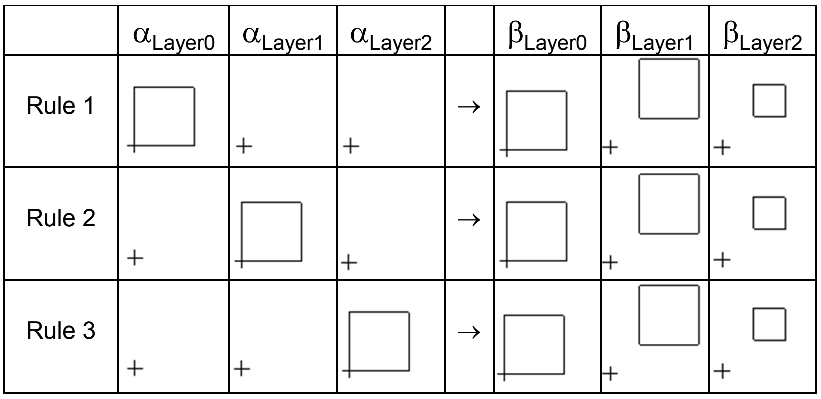


Figure 4.4: A layered shape grammar

While defining and understanding a layered shape rule is apparently more complicated than in the case of a standard shape rule, this shortcoming can be circumvented by integrating the layered shape grammar interpreter into a CAD tool with layer facilities. These kind of tools usually allow viewing different layers separately, as well as any combination of them at once. As explained in Section 4.3, we have built our interpreter on a commercial tool that offers such layer capabilities.

4.2.2 Layered Subshape Detection Algorithm

Shape grammar interpreters spend most of their time solving the so-called *subshape detection problem*: given two shapes α and γ, find all transformations t such $t(\alpha) \leq \gamma$. A reference algorithm for this problem was proposed by Krishnamurti (1981). We have extended this algorithm to the layered case.

Let us start describing the original algorithm. It works with a fixed set of three points in α. This set, called DP_α, is mapped to every plausible combination of three points (a so-called *triplet*) inside γ, and if both sets form similar triangles, then the six points are used to calculate a transformation t. If $t(\alpha) \leq \gamma$, then t is added to the solution set. The set DP_α must be carefully chosen, because the number of triplets inside γ depends on the type of points involved in DP_α. For example, consider the shapes α and γ in Figures 4.5 and 4.6, respectively (in this work, labels are represented by means of circles). In 4.6 we can find 22 intersection points and 4 white labels. If the

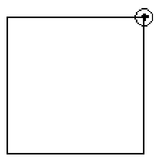


Figure 4.5: A shape α

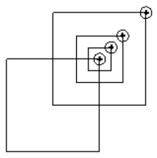


Figure 4.6: A shape γ

set DP_α is composed of three intersection points (namely, three corners of the square in Figure 4.5), then the number of triplets in γ is $22 \times 21 \times 20$. On the other hand, if DP_α is composed of two intersection points (two corners) and the white label, then the number reduces to $22 \times 21 \times 4$, so the most efficient DP_α is this second one.

If we have a shape γ with n layers $(La_0, La_1, \cdots, La_{(n-1)})$, our algorithm computes all the DPs for $\alpha_{La_i}, 0 \leq i \leq (n-1)$. We can compute the number of triplets for every three-point set, and run the original algorithm only in the layer that requires less combinations; as the subshape relation must hold in every layer simultaneously, we can stick to one layer in order to compute all the possible transformations. Table 4.2 shows the pseudo code for this algorithm. The method `getTransformations`(α, β, DP) in our algorithm performs the original Krishnamurti's algorithm (Krishnamurti, 1981) for subshape detection. The inputs for this original algorithm are shapes with presumably less points and segments than the complete shape resulting from the union of every layer, making the computation faster. In the Section 4.6 we provide a detailed analysis of time performance.

4.3 The ShaDe Tool

ShaDe [1] is a generative design tool that relies on layered constrained shape grammars in order to propose design solutions: it handles layered shapes that are intended to represent a design that meets a set of logic constraints and goals.

In this section we focus in the architecture of ShaDe and some relevant aspects of its interface. A more detailed description can be found in the user's guide and technical reference (Ruiz-Montiel, 2012b,a).

[1]ShaDe can be downloaded from `http://www.lcc.uma.es/~perez/ntidapa/`.

```
Algorithm: Layered Subshape Detection (layeredSubshapeDetection)
Input: $\alpha$, $\gamma$
Output: A collection of transformations $t$ such that $t(\alpha) \leq \gamma$
begin
    $n$ = numberOfLayers($\gamma$)
    for $i$ in $[0..n-1]$ do
        $DP$ = computeMostEfficientDP($\alpha_{La_i}$, $\gamma_{La_i}$)
        if (size($DP$) == 3)
            combinationsNumber = computeCombinationsNumber($DP$,$\gamma_{La_i}$)
            if isMinimumNumberFound(combinationsNumber)
                bestLayer = $La_i$
                bestDP = $DP$
            end
        end
    end
    if bestLayer
        result = getTransformations($\alpha_{bestLayer}$, $\gamma_{bestLayer}$, bestDP)
    end
    return result
end
```

Table 4.2: Algorithm for layered subshape detection

ShaDe is built on Trimble SketchUp, a popular, freely-available 3D modelling software that can be extended with Ruby (Ruby Language, 2015) plugins. ShaDe has been implemented as one of these plugins.[2] This approach presents several adavantages:

1. Providing a generative tool integrated with a commercial CAD tool can help to overcome the alleged confinement of design generative software to the academic world (Hanna & Barber, 2001)

2. It is generally acknowledged that design practitioners tend to think visually (Hoisl & Shea, 2011). ShaDe incorporates the sophisticated tool set of Sketchup in the graphic interface of its grammar interpreter, so designers can program shape rules by directly manipulating graphics with the help of visual tools, instead of using a programming language

3. Dynamically supported Ruby scripts provide support for the logic predicates used to control shape grammar application

In Figure 4.7 we can see the architecture of ShaDe. There are four ShaDe-specific modules: (1) Layered Shape Grammar Interpreter module, (2) Design Generation Module, (3) ShaDe Interface and (4) Predicate Generation Module. These are covered in the next sections.

[2] For efficiency reasons, the shape grammar algorithms are not executed within the Ruby interpreter that comes along with SketchUp, but by an external Ruby command.

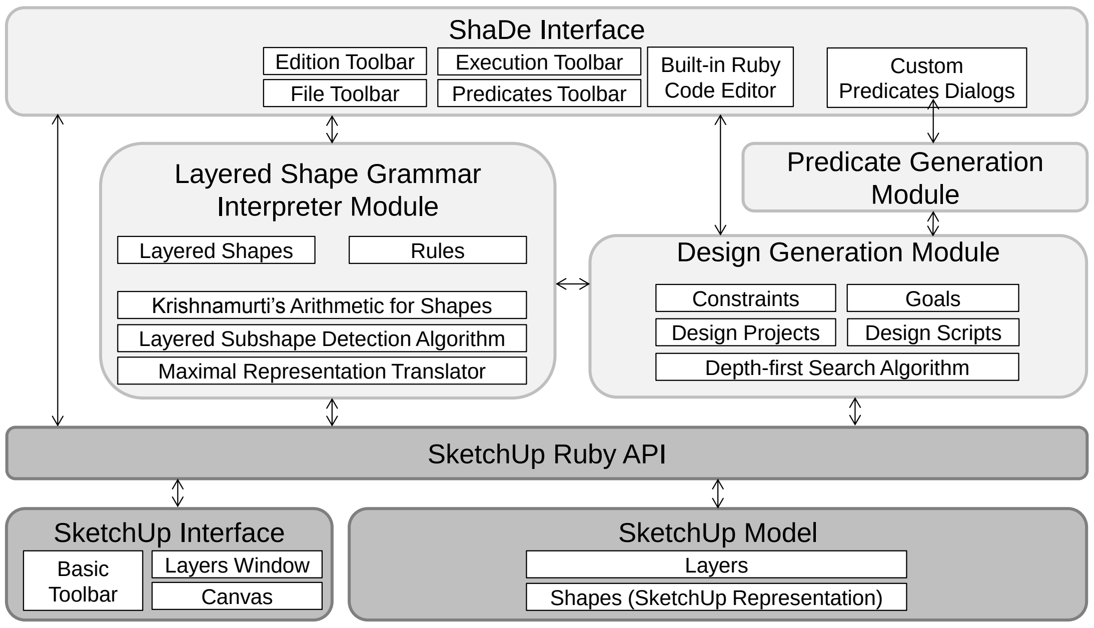

ShaDe Interface
Edition Toolbar
Execution Toolbar
File Toolbar
Predicates Toolbar
Built-in Ruby Code Editor
Custom Predicates Dialogs
Layered Shape Grammar Interpreter Module
Layered Shapes
Rules
Krishnamurti's Arithmetic for Shapes
Layered Subshape Detection Algorithm
Maximal Representation Translator
Predicate Generation Module
Design Generation Module
Constraints
Goals
Design Projects
Design Scripts
Depth-first Search Algorithm
SketchUp Ruby API
SketchUp Interface
Basic Toolbar
Layers Window
Canvas
SketchUp Model
Layers
Shapes (SketchUp Representation)

Figure 4.7: ShaDe architecture

4.3.1 Layered Shape Grammar Interpreter Module

The *Layered Shape Grammar Interpreter* (LSGI) module relies on our layered subshape detection algorithm, as well as in the arithmetic algorithms developed by Krishnamurti (Krishnamurti, 1980), in order to implement the application of shape grammars. As the SketchUp representation of shapes is not maximal in general, ShaDe implements a translation method that receives the SketchUp entities that constitute a shape and returns the maximal representation for the same shape. This method is invoked every time a new shape is created or edited inside the graphical interface. When a layered shape needs to be painted, the LSGI module communicates with the Ruby API in order to create the corresponding SketchUp geometric entities.

Layered shape grammars are implemented by means of the entities that SketchUp provides for dealing with layers. We use the so-called *Layer* entity of SketchUp in order to organize the maximal representation of the shapes; for each layer we keep the structures that store the maximal representation of the corresponding shapes.

4.3.2 Design Generation Module

The *Design Generation* (DG) module takes into account the design requirements (specified as logic predicates) in order to propose solutions. The solution space to explore, configured as a tree-like structure (see Section 4.1) is generated thanks to the LSGI module.

The logic predicates that control the depth-first search described in Section 4.1 are implemented as Ruby scripts that are invoked every time a node is checked. Users can (de)activate both constraints and/or goals at any moment.

ShaDe allows the association of a shape grammar with a set of logic predicates. This is done by means of the *project* entity. Projects can be executed in two different modes: (1) applying a number of rules and (2) applying rules until the design goals are fulfilled. When (1) is chosen as execution mode, the goals of the project are not taken into account, applying a maximum depth search limit.

Design scripts can be used to establish sequences of projects. Each project in a script is executed from the shape produced by the previous project. The first project starts from an *axiom*, i.e., a shape provided by the user. Scripts allow to specify different phases in the generation of a design, as we will illustrate in sections 4.4 and 4.5.

The DG module can also use functions of the SketchUp Ruby API in order to implement the behaviour of constraints and goals.

4.3.3 ShaDe Interface

The Interface of ShaDe relies partly on the already provided canvas and basic drawing/layer tools of SkethUp. It adds several specific toolbars that gather different commands that deal with ShaDe entities.

In Figure 4.8 we can see a screenshot of the graphical interface of ShaDe. In the following, we describe the user interaction with ShaDe.

1. *Rule Canvas.* Rules appear at the left side of the canvas. All drawing tools can be used to create and edit all rule shapes with a pointing device.

2. *Design Canvas.* The current shape (or the axiom, when no rules have been applied) appears at the right side of the canvas.

3. *Basic SketchUp Toolbar.* This toolbar gathers some basic SketchUp commands that can be used to edit shapes and also to change the canvas view (zoom in/out, rotate, etc.).

4. *Edition Toolbar.* Provides specific commands in order to perform shape grammar editing tasks like adding or removing rules, managing shapes and labels, and also visualizing the design in three dimensions.

5. *File Toolbar.* This toolbar provides commands that save/load several ShaDe entities (shapes, grammars, projects or scripts).

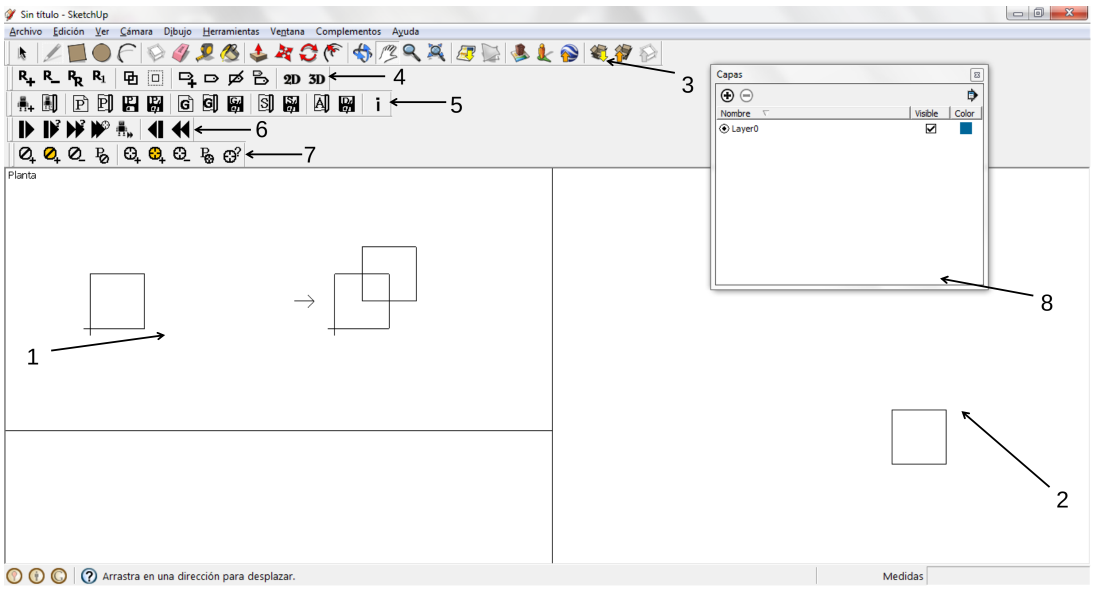

Sin título - SketchUp
Archivo Edición Ver Cámara Dibujo Herramientas Ventana Complementos Ayuda
Capas
Nombre Visible Color
Layer0
Planta
Arrastra en una dirección para desplazar.
Medidas
1
2
3
4
5
6
7
8

Figure 4.8: Screenshot of ShaDe

6. *Execution Toolbar.* These commands allow the user to execute the shape grammar with some of the following protocols: (a) applying certain rule once, (b) applying random rules a number of times, (c) applying random rules until goals are met, (d) executing a design script.

7. *Predicates Toolbar.* Provides commands for managing constraints and goals (mainly activate/deactivate them, defining new predicates and checking if goals are met).

8. *Layers Window.* Allows layer management (choosing the current layer, which layers are displayed, etc.).

4.3.4 Predicate Generation Module

The Ruby scripts for some logic predicates that we consider of common use are predefined in ShaDe. They are gathered in Table 4.3. The user can activate these at any time, so that they are taken into account in rule application.

However, in many cases the user will probably need additional logic predicates. For this reason, ShaDe offers the possibility of defining personalized constraints and goals. This can be done in two ways:

1. Graphically, thanks to a predicate generation module that asks for information about the logic predicate

Predicate Name	Type	Description
Produce distinct shape	Constraint	`satisfied? = true` iff the new shape is distinct from the previous one
No scales	Constraint	`satisfied? = true` iff the transformations used to apply the rule do not involve scales
Inside contour	Constraint	`satisfied? = true` iff the current shape is inside certain contour
No labels	Goal	`satisfied? = true` iff the current shape has no labels
All rules applied	Goal	`satisfied? = true` iff all the rules of the grammar have been applied at least once
No more rules can be applied	Goal	`satisfied? = true` iff no more rules of the grammar can be applied, either because of negative subshape detection or because some constraint is violated

Table 4.3: Predefined constraints and goals provided with ShaDe

2. Directly coding the logic predicate in Ruby, inside a built-in code editor

When the graphic option is chosen, the *Predicate Generation* (PG) module gathers information about the predicate by means of a set of graphic dialogues, and then automatically generates the corresponding Ruby code. We have tried to develop a general and user-friendly graphic interface, covering a wide range of possibilities by means of dialogues and forms that the designer has to fill in. Basically, the predicates can refer to five different entities:

1. Rule IDs. Example: *the applied rule is the one with ID 3*; or *the first applied rule is the one with ID 1.*

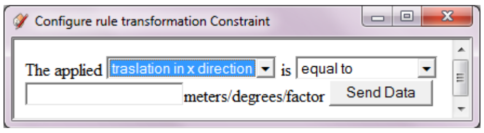
2. Applied transformations. Example: *the applied scale factor is bigger than 2.*

3. Segments of the generated shape. Example: *the shape must not contain segments larger than one meter.*

4. Points of the generated shape. Example: *there are more blue labels than yellow ones inside the shape*, or *there are at least two yellow labels at a distance of 3.6 meters of a blue label.*

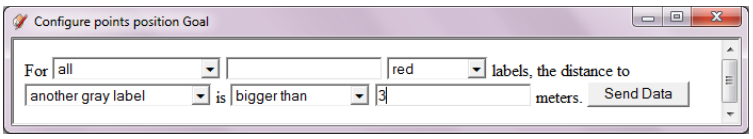
5. The whole generated shape. Example: *the shape is inside a contour*, or *the shape is a sub-shape of another given one.*

In figures 4.9 and 4.10 we can see some graphic dialogues for configuring predicates related to the applied transformations and the labels of the generated shape, respectively.



The applied traslation in x direction is equal to meters/degrees/factor

Figure 4.9: GUI for predicate configuration: applied transformation



For all red labels, the distance to another gray label is bigger than 3 meters.

Figure 4.10: GUI for predicate configuration: current shape labels

4.4 Example 1: Designing with the *Hele* Module

In this section we describe a design script used to generate housing unit schemes. This design script is based on some ideas of Rafael Leoz, a Spanish architect whose most productive period fell between 1950 and 1970.

The goal of this example is to focus on the conceptual design facet of ShaDe. More precisely, we show that our system can be used to generate many varied solutions that, although low-detailed, meet an essential set of design requirements and thus can be used as starting points for complete design projects. In addition, we describe in detail all the shape grammars, constraints and goals used in the example. A video displaying the most relevant steps of this example is available on the Web[3].

4.4.1 The *Hele Module* and Predefined Rooms

One of the main motivations behind Leoz's work was the improvement of social housing. Leoz studied how to build affordable houses by means of mass production. The basic idea is to start from a small number of simple shapes that can be easily mass-produced, and aggregate them in order to obtain a vast number of more complex shapes. The most important contribution in this sense is the so-called *Hele Module* or just *Leoz Module* (Leoz, 1969). Leoz chose this shape because it offers many harmonic combination possibilities, since it does not have symmetries and its perimeter is configured according to the Fibonacci numbers (see Figure 4.11).

We have used pairs of Hele modules in order to produce samples of single-family housing units. Each Hele module is made up of four cells of 3.60 $\times$ 3.60 meters. Two modules can be combined in many different ways. We can see some examples of such combinations in Figure 4.12.

[3]`http://youtu.be/1SwkKOinJ8A`

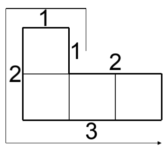


Figure 4.11: Hele Module (reproduced from (Leoz, 1978))

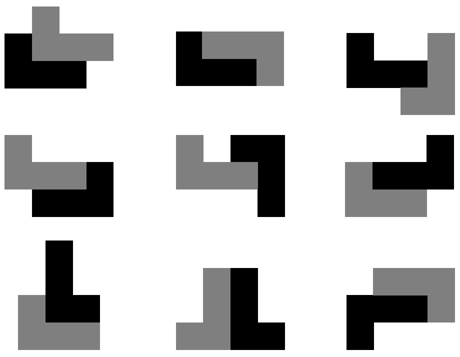


Figure 4.12: Some Hele Module combinations (adapted from (Leoz, 1978))

Leoz usually employed a set of predefined rooms in order to easily configure the interior space of the houses (Leoz, 1978). Since we are using two Hele modules for each housing unit, we have eight cells available to be filled with a set of rooms. The chosen set of rooms, inspired by those used by Leoz in several projects (Leoz, 1978), is depicted in Figure 4.13, with six single-cell rooms and a double one.

4.4.2 The Leoz Design Script

We developed a design script in Shade to produce several distinct 2D housing units schemes, using pairs of Hele modules and the set of predefined rooms depicted in Figure 4.13.

Leoz usually worked with grids that divided the plane in a regular manner. From these grids, he extracted simple shapes that were further aggregated in order to produce more complex shapes (Leoz, 1978). The Hele module comes from a rectangular grid. This grid can be used as axiom for our first production step. The subshape recognition algorithm used by the shape grammar interpreter will recognize the Hele modules inside this grid. We can now establish the layers that will reduce the computational cost of grammar execution. They will also help us to represent designs in three dimensions and to visualize different aspects of the design separately:

1. The underlying grid

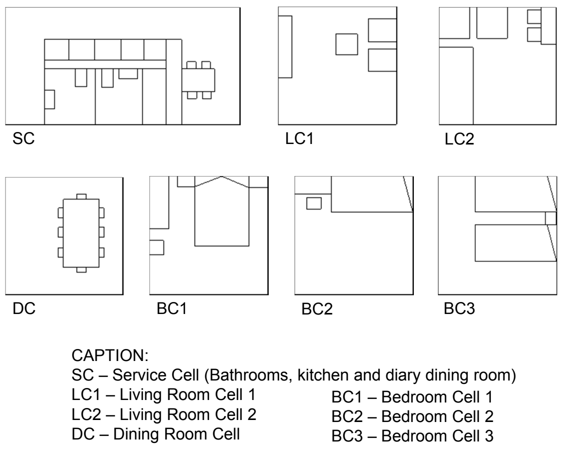


Figure 4.13: Set of predefined rooms

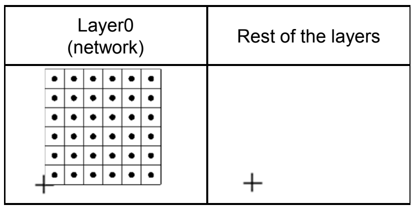


Figure 4.14: The axiom for the Leoz script

2. The Hele modules yielded by the grid
3. The walls of the rooms
4. Elements with table-height
5. Elements with seat-height
6. Elements with wardrobe-height

Figure 4.15 depicts the layers with the help of an example. All the shapes and rules of the grammars need to be defined for the six layers.

The Leoz design script comprises three design projects, each one with its own shape grammar, constraints and goals (see Table 4.4). In the following we describe the aim of each project:

- Project 1: *First Hele Module.* The first production step is to add the first Hele module from the axiom (see Figure 4.14). Figure 4.16 displays the single rule for this project. Labels are black in the grid layer, blue in the modules layer and white in the rooms layer. No constraints nor goals are needed.

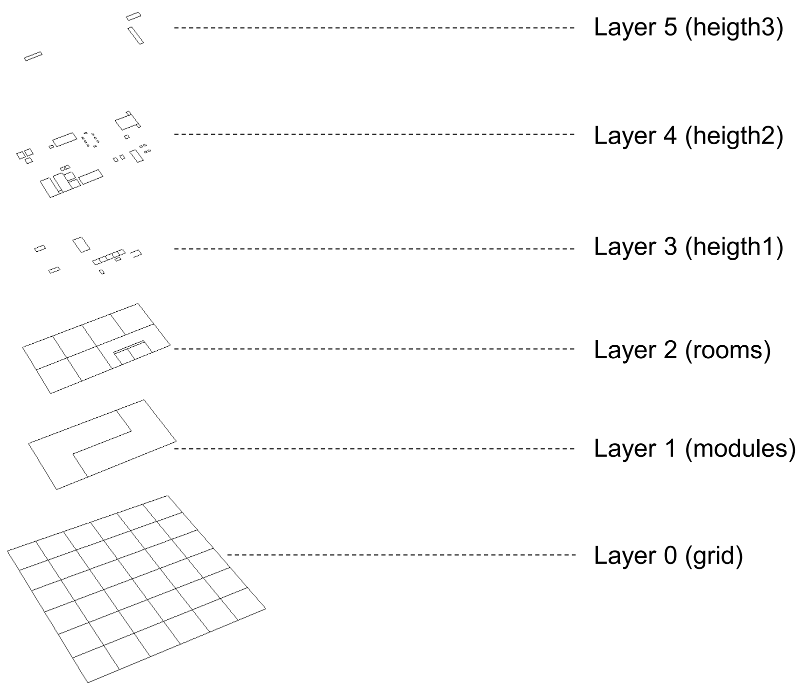


Figure 4.15: The layers of Leoz script for ShaDe

- Project 2: *Second Hele Module*. Starting from the shape produced by Project 1, Project 2 adds the second Hele module. In Figure 4.17 we can see the single rule for this project. Labels are black in the grid layer, yellow in the modules layer and white in the rooms layer. A constraint is needed to ensure that both modules are in contact. We have established that two labels of the second Hele module (coloured in yellow) must be at a distance of 3.6 meters of a label from the first Hele module (coloured in blue). In Figure 4.12 we can see a small set of Hele module combinations. Thanks to the emergence properties of shape grammars, ShaDe is able to automatically generate all the combinations of two Hele modules.

- Project 3: *Rooms*. This project adds different predefined rooms that are depicted in Figure 4.13 (the pieces of furniture are schematic), starting from the shape produced by Project 2. It has seven rules, one for each kind of room (Figure 4.18). Before the application of these rules, labels are white in the rooms layer. When rules are applied, their colors change depending on the room that is added. We need four constraints and one goal for the execution of this project:

  – Constraint 1: The transformations involved in subshape recognition must

Project	**Constraints**	**Goals**	**Execution mode**
1. First Hele module	-	-	1 rule
2. Second Hele module	(1) Hele modules in contact	-	1 rule
3. Rooms	(1) No scaling (2) First rule applied: service room (3) Rules are applied in order (4) Distance(service, dining) $< 5m$	(1) All rules have been applied	Goals

Table 4.4: Leoz script summary

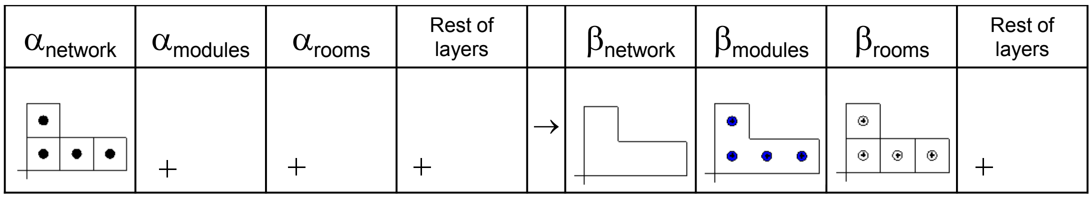


Figure 4.16: Rule for Project 1 (Leoz example)

not involve scaling, that is, the rooms are to be added in their original size. This constraint comes predefined with ShaDe (see Table 4.3).

– Constraint 2: The first rule applied must be the one of the service cell, since it is the one that needs more space.

– Constraint 3: The rules are applied in order, to make sure that every room is present.

– Constraint 4: The dining room must be placed near the service room; these two rooms cannot be separated by more than 5 meters. This constraint can be configured using the coloured labels that the rules add to the *rooms* layer.

– Goal: All the rules of the grammar have been applied. This goal comes predefined in ShaDe (see Table 4.3).

This architectural example here clearly shows that some design criteria are easier to specify by means of shape rules, and other adapt better to predicates. For instance, the use of L-shaped modules is easier to elicit by means of geometric rules rather than by some kind of explicit programming. However, the fact that the two Hele modules have to be in contact is better expressed as a logic predicate: if we tried to express it by means of shape rules, we would need to configure one distinct rule for every possible combination of two modules.

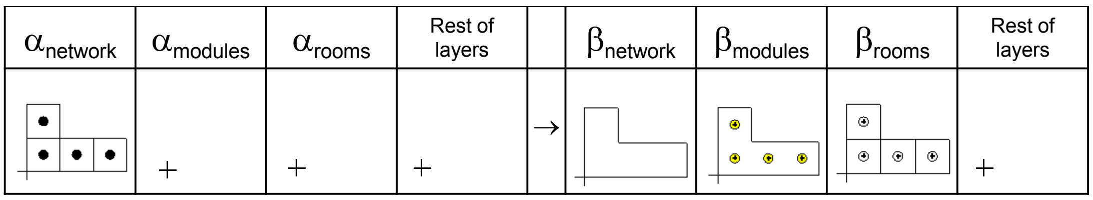


Figure 4.17: Rule for Project 2 (Leoz example)

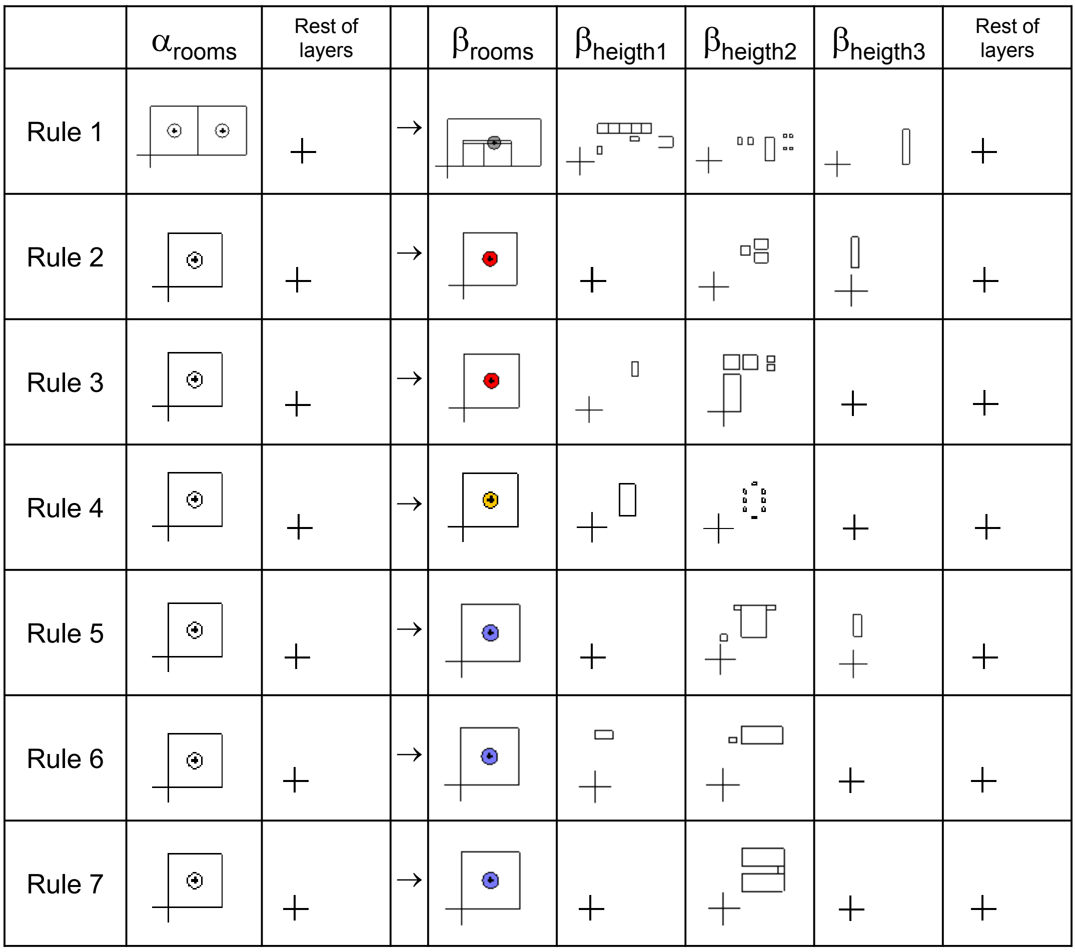


Figure 4.18: Rules for Project 3 (Leoz example)

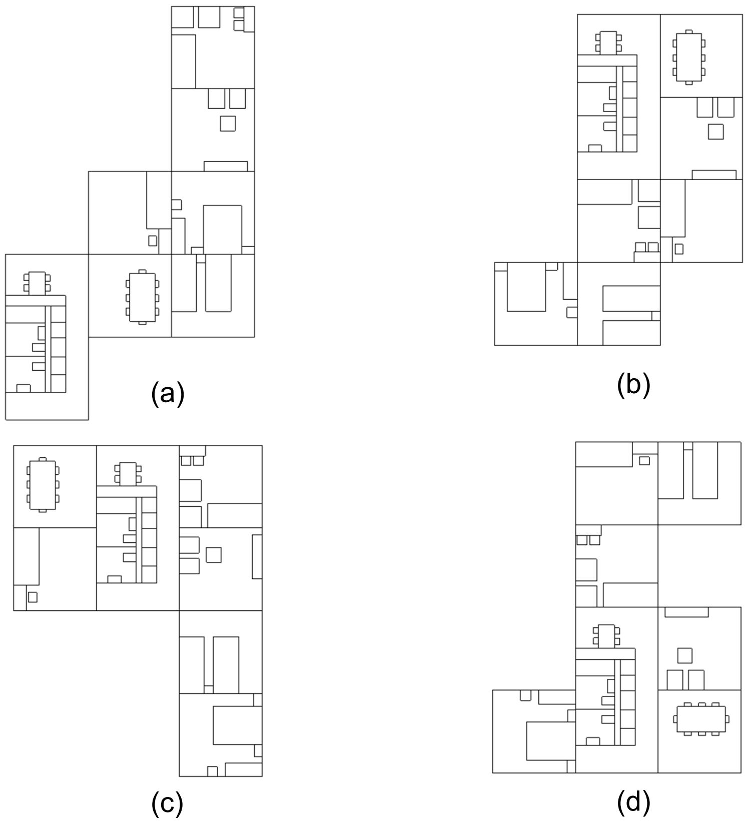


Figure 4.19: Four designs produced by the Leoz design script

4.4.3 Results

We asked the tool to generate ten designs using our design script (although there is no limit to the number of designs we can ask ShaDe to generate). We show four of them in Figure 4.19. The designs satisfy the constraints and goals gathered in Table 4.4, and also present considerable variability. By specifying heights for each layer, the results can also be visualized in three dimensions. We show the housing unit of Figure 4.19(a), once the height information for every layer has been determined, in Figure 4.20. Some tools of SketchUp have been used in order to add textures and visualize it in a friendly way.

4.5 Example 2: Virtual Restaurants

In this section we describe a second example that generates virtual restaurants to be used in the context of restaurant simulation video games[4]. This example aims to

[4]The script containing the shape grammars, layers, constraints and goals of this example can be downloaded from `http://www.lcc.uma.es/~perez/ntidapa/restaurant-script.zip`

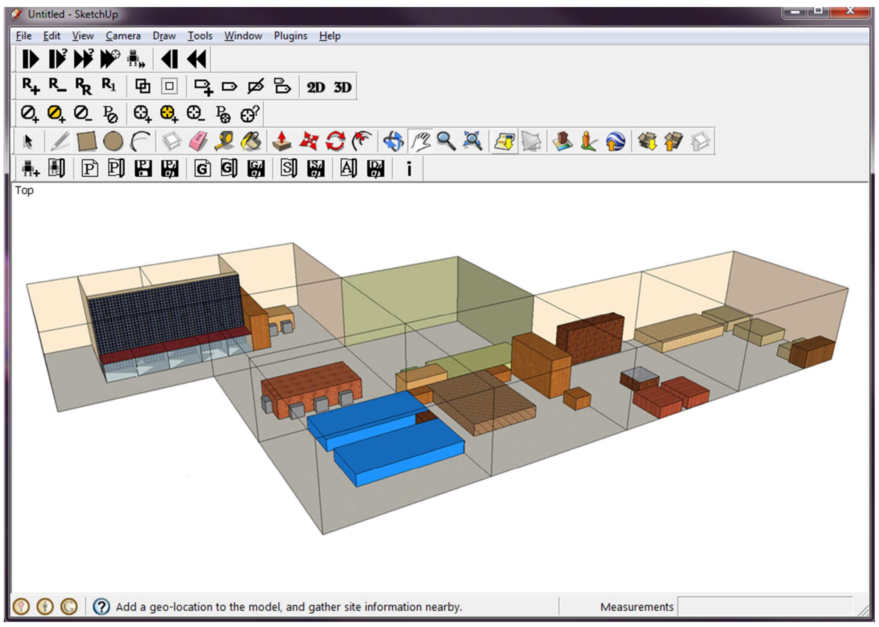

Untitled - SketchUp
File Edit View Camera Draw Tools Window Plugins Help
2D 3D
Top
Add a geo-location to the model, and gather site information nearby.
Measurements

Figure 4.20: A housing unit with height information and textures

produce more detailed solutions from the computer graphics point of view, and it also takes into account a set of requirements that are implicitly present in typical restaurant simulation video games, therefore it is oriented to provide gamers with complete (not conceptual) building solutions. Nevertheless, the produced designs can also be used as starting points in the construction process typical of this kind of video games.

4.5.1 Strategy/simulation Video Games

In many strategy/simulation video games the player has to build her own world (for example *The Sims*, *Restaurant Empire*, *Hotel Giant*, etc.). For some users, part of the appeal comes from this construction process, whereas for others the main source of entertainment is the simulation, where they just take a predefined virtual world and start specifying the behaviour of the elements in the game. In this context, a system that automatically generates virtual worlds provides a twofold advantage: on one hand it can be used by construction-oriented gamers to obtain different starting points for their creations, and, on the other hand, it provides simulation-oriented gamers with an unlimited number of different starting points.

We have developed an example related to restaurant simulation video games. Some examples of this genre are the PC saga *Restaurant Empire* or the Android application *Restaurant Story* (see Figure 4.21).

Figure 4.21: Screenshots of the games *Restaurant Empire* (a) and *Restaurant Story* (b)

4.5.2 The Restaurant Design Script

In our example we consider the following elements of a restaurant building: contour, entrance door, windows, kitchen, tables, chairs and bathroom. Each element is decomposed in several layers, summing up a total of 27 layers. These will significantly reduce the computation time of the involved shape grammars, as well as allow a simple three-dimensional representation with high level of detail, and a comfortable visualization of the different elements. More concretely, the following details have been included:

- Walls have thickness
- The buildings have main doors and windows
- The furniture pieces are not box-shaped anymore. For example, the chairs have legs, seats and backs
- Further additional details have been provided: columns and stairs for the main door, frames for the windows, burners for the kitchen, etc.

These details are added by six shape grammars whose rules are decomposed in 27 layers. The rules are depicted in Figures 4.23 to 4.28. The design process is divided into six projects with 9 constraints and 5 goals, establishing a strong interaction between the elements so as to produce restaurants according to the implicit design rules that we can find inside a typical restaurant video game. The first project starts applying rules from the axiom depicted in Figure 4.22, and the rest of projects start from the shape produced by the previous project. Note that, for space reasons, the empty layers do not appear in the figures for the axiom and projects of the restaurant script.

Descriptions, constraints and goals for the six projects are gathered in table 4.5. The script generates restaurants of $50m^2$ with five tables and at least ten chairs,

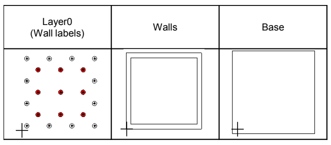


Figure 4.22: The axiom for the restaurant script

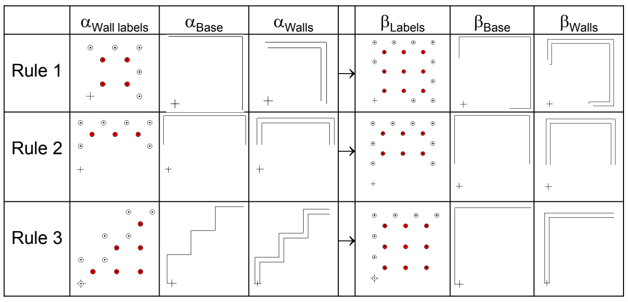


Figure 4.23: Rules for Project 1 (restaurant example)

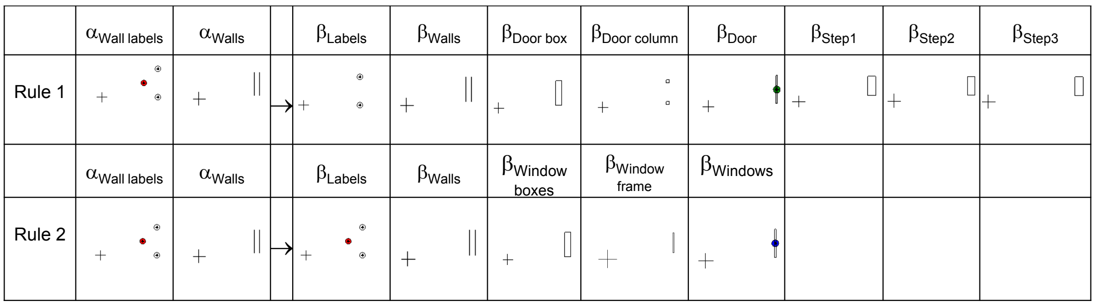


Figure 4.24: Rules for Project 2 (restaurant example)

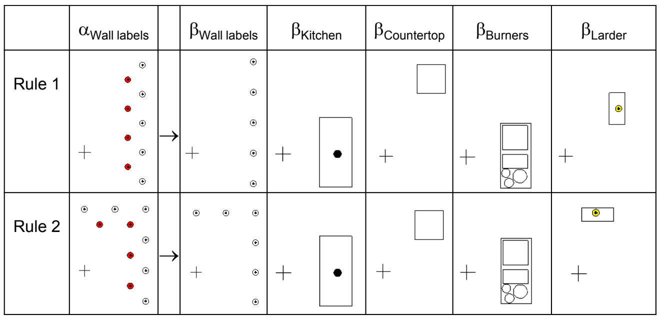

$\alpha_{\text{Wall labels}}$
$\beta_{\text{Wall labels}}$
β_{Kitchen}
$\beta_{\text{Countertop}}$
β_{Burners}
β_{Larder}
Rule 1
Rule 2

Figure 4.25: Rules for Project 3 (restaurant example)

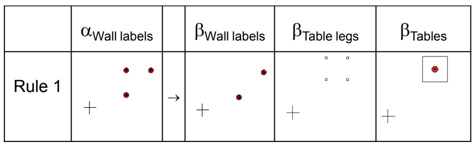

$\alpha_{\text{Wall labels}}$
$\beta_{\text{Wall labels}}$
$\beta_{\text{Table legs}}$
β_{Tables}
Rule 1

Figure 4.26: Rule for Project 4 (restaurant example)

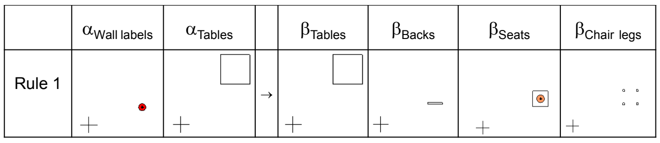


Figure 4.27: Rule for Project 5 (restaurant example)

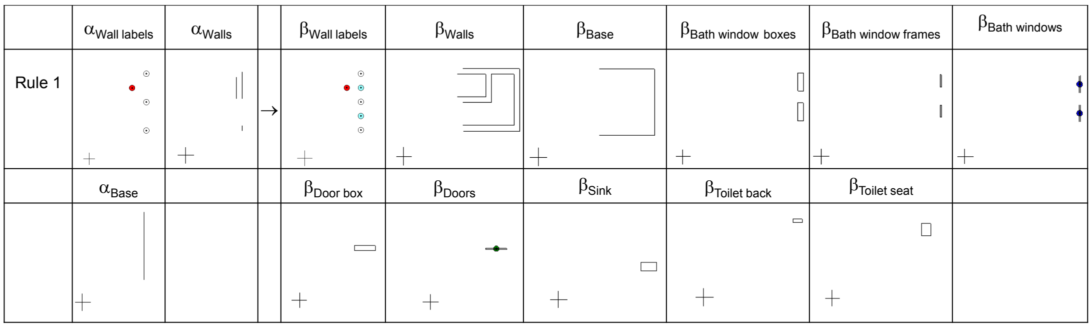

$\alpha_{\text{Wall labels}}$
α_{Walls}
$\beta_{\text{Wall labels}}$
β_{Walls}
β_{Base}
$\beta_{\text{Bath window boxes}}$
$\beta_{\text{Bath window frames}}$
$\beta_{\text{Bath windows}}$
Rule 1
→
α_{Base}
$\beta_{\text{Door box}}$
β_{Doors}
β_{Sink}
$\beta_{\text{Toilet back}}$
$\beta_{\text{Toilet seat}}$

Figure 4.28: Rule for Project 6 (restaurant example)

which are reasonable as starting points for this kind of games. Results can be further developed or improved by the players according to their preferences or game objectives.

4.5.3 Results

As in the previous example, we asked the tool to generate ten designs using our design script. We show four of them in Figure 4.29. The designs satisfy the constraints and goals gathered in Table 4.5, and also present considerable variability. We show a restaurant in three dimensions, once the height information for every layer has been determined, in Figure 4.30. We can use the tools of SketchUp to add some textures and visualize it in a friendly way (in Figure 4.31(a) we can see another restaurant with textures). In Figure 4.31(b) we can see the same restaurant of Figure 4.31(a) with an automatically generated roof, thanks to the Sketchup plugin *Roof Maker* (Roof Maker, 2015).

4.6 Evaluation of Time Performance

4.6.1 Theoretical Analysis

The original Krishnamurti's algorithm for subshape detection (Krishnamurti, 1981) (that is, for computing all the transformations t such $t(\alpha) \leq \gamma$) considers all the triplets inside shape γ. When γ has n points and all of the points inside α and γ are of the same type, the number of triplets to be considered in γ is $P(n,3) = (n)(n-1)(n-2)$

Project	**Constraints**	**Goals**	**Execution mode**
1. Contour	(1) No scaling	(1) Total Area $\geq 50m^2$	Goals
2. Entrance Door and Windows	(1) No scaling (2) Maximum doors: 1 (3) Maximum windows: 70% of the perimeter	(1) Minimum doors: 1 (2) Minimum windows: 20% of the perimeter	Goals
3. Kitchen	(1) No scaling (2) Distance(kitchen, door)$\geq 3m$ (3) Distance(larder, windows)$\geq 0,6m$	-	1 rule
4. Tables	(1) No scaling (2) Distance(table, tables)$\geq 1,5m$ (3) Distance(table, kitchen)$\geq 3m$	-	5 rules
5. Chairs	(1) No scaling	(1) Minimum chairs: 10 (2) Minimum chairs per table: 1	Goals
6. Bathroom	(1) No scaling (2) Distance(bathroom, windows)$\geq 1m$ (3) Distance(bathroom, door)$> 0m$	-	1 rule

Table 4.5: Restaurant script summary

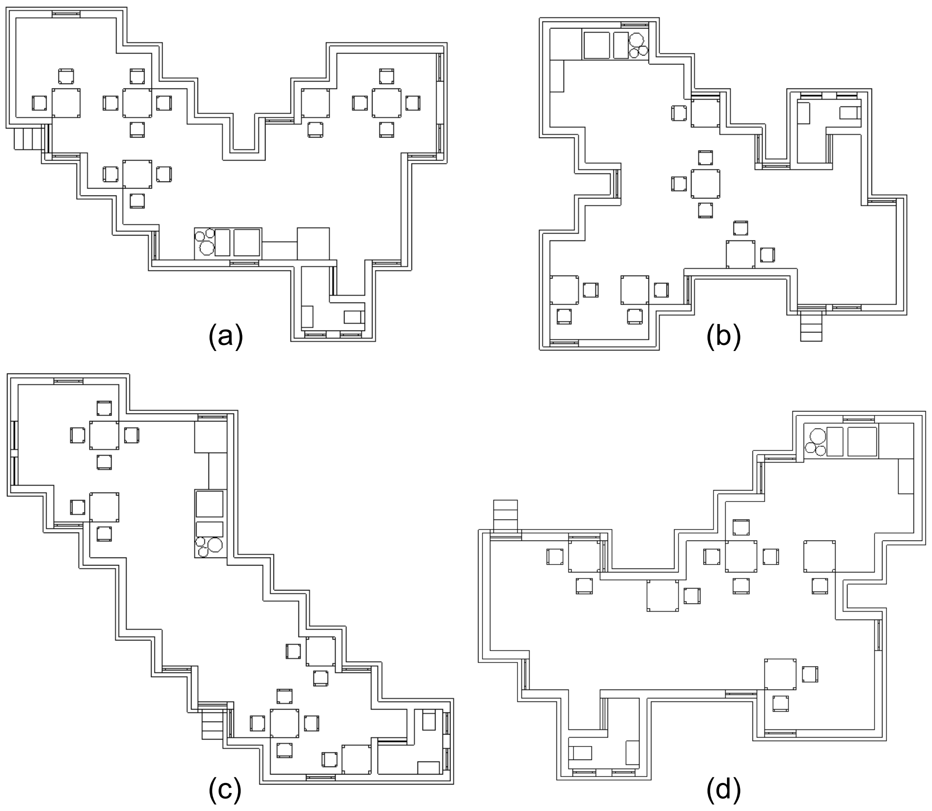


Figure 4.29: Four designs produced by the restaurant design script

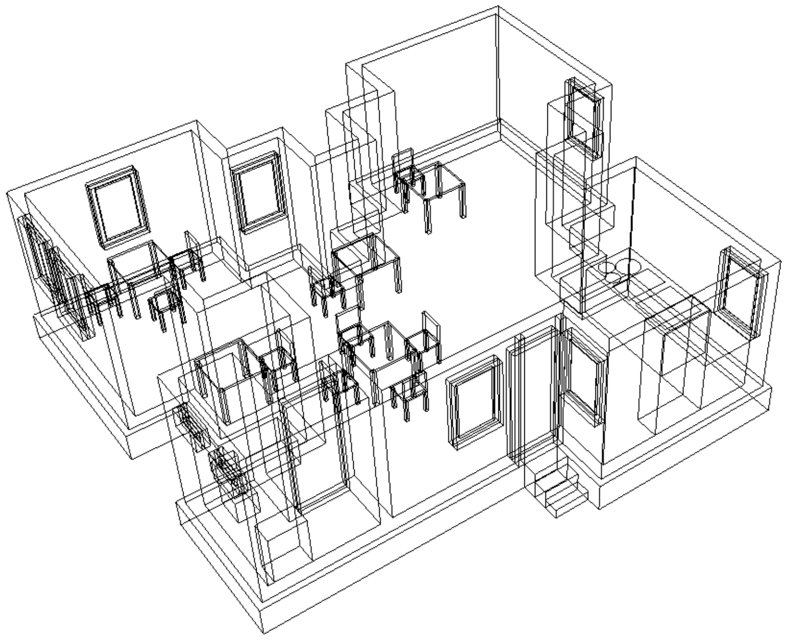


Figure 4.30: A restaurant in three dimensions

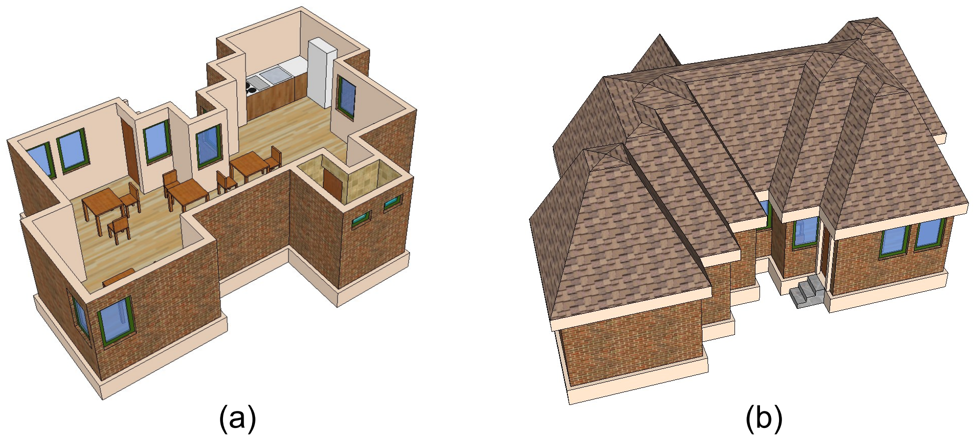


Figure 4.31: A restaurant with textures (a) and with roof (b)

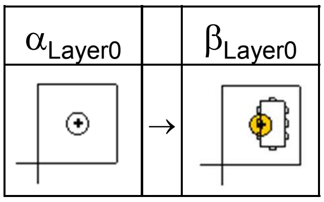


Figure 4.32: Rule 4 of the third Leoz Project (flattened)

(that is, the number of ordered 3-permutations of n elements). Since it tries to compute the associated transformation for each triplet (if it exists), the original algorithm has a complexity of $\Theta(n^3)$. However, a latter implementation of the algorithm (Krishnamurti & Giraud, 1986) reduces this bound to $\Theta(n^2)$. This is achieved considering only two distinguishable points with two possible candidates for the third point.

Now suppose we use k layers to reformulate the shape grammar. In the average case, a shape γ has $\frac{n}{k}$ points inside each layer, that is, there are $\frac{n}{k}(\frac{n}{k}-1)(\frac{n}{k}-2)$ triplets in each layer. Since our algorithm for layered subshape detection (Section 4.2.2) only focuses in one layer to compute the transformations (the one with less triplets), now the performance is $\Theta((\frac{n}{k})^3)$ (or $\Theta((\frac{n}{k})^2)$ for the two-point version). For example, if we use 10 layers, then in the average case the performance is divided by 10^3 (10^2 for the two-point version). In the worst case, all the points are inside a single layer and thus we can only use that layer in order to determine the transformations, so the performance is the same as with single-layered grammars. In the best case, there is a layer that only has one triplet, so determining the unique associated transformation (if it exists) would have a complexity of $\Theta(1)$.

4.6.2 Empirical Analysis

Here we present the differences between the execution of a standard shape grammar and a layered one. All tests reported here have been run on an Intel Core i7 860 @2.80 GHz processor with 8GB RAM and Windows 7 (64 bits).

Tests were run over the shape grammar of the third Leoz Project, that is, the one that adds the room cells inside the Hele modules. An example of one of the rules is depicted in Figure 4.18. There are seven similar rules in total, one for each type of room cell. Two versions of this shape grammar have been used: the layered and the flattened one. The second one has all its shapes in one single layer. For example, the flattened rule equivalent to rule 4 in Figure 4.18 is depicted in Figure 4.32.

Both shape grammars have been executed with the same protocol: the seven rules are applied in order, step by step, using as axiom an outcome (layered or flattened as appropriate) of the second Leoz project. Our aim here is not to produce valid housing

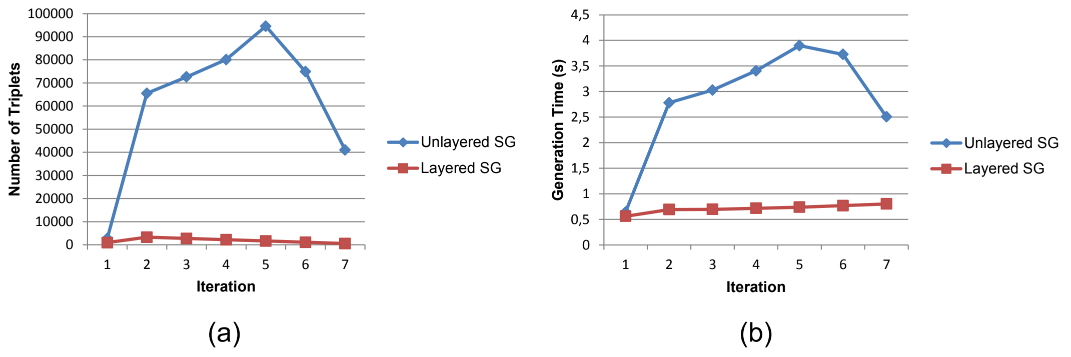


Figure 4.33: Performance measures: (a) number of triplets detected and (b) generation time in layered and unlayered approaches

	Unlayered	**Layered**
Maximum	130,73s	32,52s
Minimum	112,54s	16,45s
Average	119,215s	21,06s

Table 4.6: Performance measures: maximum, minimum and average generation times of Leoz design script for layered and unlayered approaches

units, but to measure performance, so a priori we do not need design constraints nor goals. Nevertheless, we have activated the constraint that prevents scaling, so both shape grammars produce the same outcomes.

Figure 4.33(a) shows the number of triplets that the subshape detection algorithm considers in both cases. Our proposal always considers less than 4000 triplets, while the unlayered approach considers many more, reaching almost 95000 triplets in the fifth iteration.

Figure 4.33(b) shows execution time for both shape grammars. Measures include the time involved in the subshape detection algorithms, as well as the time needed for shape arithmetic operations. We can see that in the layered case we need less than one second for every rule application, while in the unlayered case we need more time, reaching almost four seconds in the fifth iteration.

In conclusion we can say that, if we can define layers for a shape grammar without sacrificing desired outcomes of its application, our layered approach can lead to a considerable improvement in time performance. This improvement is even more evident in the execution of the whole Leoz design script, where the design requirements are more demanding and thus the depth-first search algorithm backtracks more times. Table 4.6 gathers the maximum, minimum and average generation times of ten executions of the Leoz design script, for the layered and unlayered approaches.

Chapter 5

Computational Techniques for Learning and Detecting Criteria in Design Problems

What constraints?
The sum of all constraints. Here is one of the few effective keys to the design problem–the ability of the designer to recognize as many constraints as possible–his willingness and enthusiasm for working within these constraints–the constraints of price, of size, of strength, balance, of surface, of time, etc.: each problem has its own peculiar list

Question: Madame L'Amic. Answer: Charles Eames

In this chapter we describe in detail the approach advanced in Section 3.2, based on learning a heuristic that determines how to apply naive generative rules. In this approach, the amount of design requirements encoded inside the rules is minimized, so the creation and modification of such rules is very easy. However, a blind execution of these naive rules lead to infeasible solutions. We propose to combine naive shape grammars with reinforcement learning processes that codify design criteria as rewards, and learn heuristics that tell how to apply the rules so as to maximize those rewards and hence produce feasible solutions (Ruiz-Montiel et al., 2011a,b, 2013a, 2015). Some previous works in the literature combine naive rules with techniques that discover good rule application sequences in order to generate feasible designs, like shape annealing (Cagan & Mitchell, 1993; Shea & Cagan, 1997, 1999b,a) or genetic algorithms (Gero et al., 1994; Jin & Li, 2007; Caldas, 2008; Wu et al., 2008). As far as we are concerned, the system presented here is the first one that combines reinforcement learning and naive rules for design.

The rest of this chapter is structured as follows: in Section 5.1 we explain the general mechanism in which naive rules can be combined with an heuristic, and how reinforcement learning techniques can be used to learn such heuristics. According to the approach used in order to deal with the multiple objectives usually present in design problems, we propose two methodologies: the first one involves scalarized MORL and the second one involves the so-called Pareto MORL. They will be described in sections 5.2 and 5.3, respectively.

5.1 Naive Shape Grammars and Heuristics

As we explained in Section 3.2, the level of design criteria hard-coded inside the rules can be minimized, allowing for an easy creation and modification of the shape grammars. One way of achieving this is by using just atomic geometric elements that will be aggregated to make up the solutions.

At this point, it is useful to introduce a small geometric problem that can be employed to illustrate the different elements of the proposed methodologies described in this chapter. Concretely, now we will use it in order to describe a possible naive shape grammar and explain why we need to *learn* an heuristic that determines how to apply its rules.

Let us suppose that we want to generate bi-dimensional shapes of a certain area, with a high compactness factor. This compactness can be formally defined as $area/perimeter^2$. The most compact bi-dimensional shape is a circle. However, imagine that we want to generate *rectilinear* polygons, which are polygons whose edges meet at right angles. This can be due because, for example, an architect plans to use that shapes as possible layouts for a house, and s/he wants to avoid curved designs and non-right angles.

In Section 3.2 we provided an example of a naive grammar that adds atomic, squared tiles in order to make up tessellated shapes (Figure 3.4). We can use this rule in order to generate rectilinear polygons. For simplicity, we will not allow rule transformations that involve scaling shapes, as we want all our tiles to be the same size.

As we explained before, if we allow curved shapes, then the solution is unique (namely a circle); but if we use squared tiles, then we can have several possible solutions depending on the number of tiles of the desired outcomes. For example, if we seek shapes with eight tiles, then we have two possible solutions (discarding symmetries), depicted in Figure 5.1.

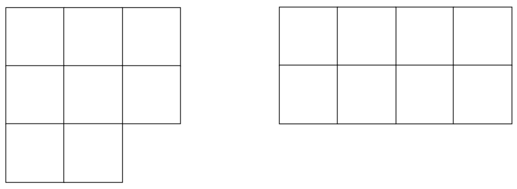


Figure 5.1: Possible solutions for the problem of generating 8-tile, compact shapes

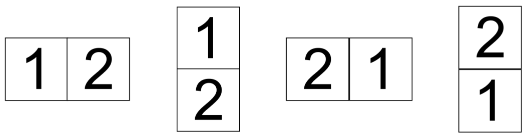


Figure 5.2: Possible intermediate 2-tile shapes

It is clear that if we just apply the rule randomly until we get shapes with the desired number of tiles, then the obtained solutions are not guaranteed to be compact. We need a heuristic that, at each step, determines which is/are the better rule/s to be applied (if there is more than one good rule, then we would choose one of them randomly). One may argue that we can just use the compactness factor as heuristic to apply the rules, in the following manner: *at each step, apply the rule that yields the most compact intermediate shape. If there is a draw, then choose the rule randomly.* For example, if we are at the first step of the derivation process (so that we only have one tile), then this heuristic could yield any of the intermediate shapes in Figure 5.2, which are topologically equivalent and thus have the same compactness factor. Imagine that we choose the first one of these shapes. To apply the second rule, we follow the same heuristic, and thus any of the shapes in Figure 5.3 can be generated, as they are also equally compact. Following this system, a shape with four tiles setted in a row could be generated. When applying the following rule to this shape, then the outcome is analogous to the previous ones: we can either yield a L or T-shaped configuration of tiles, or a line of 5 tiles (Figure 5.4). It becomes clear that from a shape of 5 tiles in a row we cannot reach any of the optimal solutions in Figure 5.1, so we need a more informed heuristic that avoids these situations.

In fact, the approach of using the set of design criteria itself as a heuristic for applying rules is somewhat equivalent to the approach described in Chapter 4, in which design criteria are directly managed through a backtracking search process. Certainly, if we undid the rule application that yielded a shape with 5 tiles in a row, we could still get an optimal solution by following another path. However, the constraints for this problem would be hard to define, as we would need to consider the particular

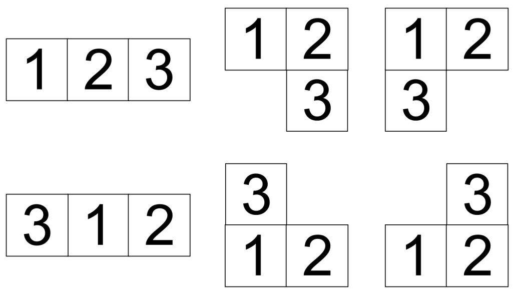


Figure 5.3: Possible intermediate 3-tile shapes

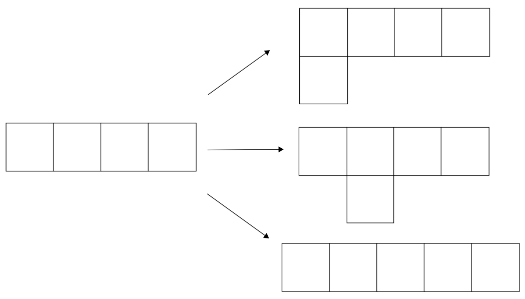


Figure 5.4: Possible intermediate 5-tile shapes

configurations of tiles that cannot lead to good solutions. Defining the goal would also be tricky, as we would need to know a priori the optimal compactness factor. Moreover, a backtracking algorithm used for managing criteria in a problem with many rule applications (imagine, for example, that we want to generate shapes with forty tiles) would have huge time needs, so it would be desirable to learn a more informed heuristic to be used in a non-backtracking process. These issues suggest that the compactness factor of a shape is not a suitable design criteria to be *managed* in the search process proposed in Chapter 4 (at least when the shape grammar used to generate the design space is naive) and instead we should use it in a more indirect manner in order to *learn* a heuristic for applying the rules.

One way of achieving this is by using the involved design criteria in order to define long-term rewards and then employing a reinforcement learning algorithm that learns how to take actions in order to maximize them. In the following section we explain how reinforcement learning can be applied to the framework of shape grammars, as well as the issues that arise if multiple criteria are present.

5.1.1 Reinforcement Learning and Naive Shape Grammars

In this section we consider the issues related to the application of a reinforcement learning technique to a shape grammar derivation process.

Given a shape grammar and an initial shape (axiom), we want to obtain a heuristic that can determine which sequences of rules lead to a final shape that complies with a certain set of design criteria. In the context of reinforcement learning, this heuristic is called a policy, and is obtained by associating each possible rule application with a long-term value, learned through repeated interaction with the environment, which in this case consists of applying rules and observing the reward of the resulting shapes.

Hence, the mechanism used to apply the rules is different to the one considered in Section 4, since in that particular case we did not associate rule applications to a long-term value: we performed a traditional backtracking search algorithm based on design constraints and goals.

By using reinforcement learning methods we would only need to run once the learning algorithm and then use the obtained policy to apply the rules until we reach a final state. A large state space would certainly increase the time needed for the algorithm to obtain a good policy, but once it is learned, generating a solution by means of this policy would be straightforward. Moreover, as we will see in Section 5.2.1.2, we can use generalization techniques to deal with such large state spaces and thus decrease the learning time.

5.1.1.1 Shape, Rules and Design Criteria as States, Actions and Rewards

Design problems usually involve more than one criteria that the solutions need to fulfil. This implies that the reward is no longer a scalar number but a vector with one component per objective. As we explained in Section 2.3.4, there are two main approaches for applying reinforcement learning techniques in such situations: the *utility-based* approach and the *axiomatic* approach. The first one relies on a scalarization function in order to treat the problem as a scalar one. The second one aims at learning the Pareto front of optimal policies.

If we rely on scalarized reinforcement learning techniques, then the obtained reward is a scalar number and the design problem can be formalized as a single-objective MDP. Given a shape grammar $\langle Sh, L, Ru, I\rangle$ and a set of design criteria, the considered MDP is defined by:

- S: the set of labelled shapes that can be produced by any rule sequence of the shape grammar
- $A = Ru \times T$: the actions are the set of possible pairs $(rule, \tau)$ where $\tau \in T$ is a

geometric transformation

- $T : S \times A \rightarrow S$: the deterministic transition function that determines which labelled shape is obtained when applying an action to a given labelled shape
- $R : S \times A \rightarrow \mathbb{R}$: the expected immediate reward when applying an action to a given labelled shape, defined from the set of considered design criteria

A policy is a function that selects a pair $(rule, \tau)$ at each step of the derivation of a shape grammar, and it is applied from the initial state s_0, which in this case is the axiom of the shape grammar. In this chapter we will use some variants of the algorithm Q-learning, described in Section 2.3.

On the contrary, if we seek to discover the whole Pareto front of policies, then the design problem must be formalized as a multi-objective MDP, which differs from the previous one in the definition of the expected immediate reward: $R : S \times A \rightarrow \mathbb{R}^n$, where now the feedback provided by the environment is a vector instead of a scalar number.

In the system proposed in Section 5.2 we follow the first approach, using a linear scalarization function for the rewards. In Section 5.3 follow the second approach, for what we describe a new algorithm, PQ-learning, which is a multi-objective extension for the temporal-difference algorithm Q-learning. PQ-learning learns the whole Pareto front of non-stationary policies for a multi-objective problem.

5.2 Scalarized MORL for Design

Here we propose a methodology that involves naive shape grammars as a generative system for the design space, and scalarized multi-objective reinforcement learning as a method for discovering a rule application heuristic.

This methodology follows the *utility-based* approach proposed by Roijers et al. (2013), in which we rely on a scalarization function in order to tackle a multi-objective problem. Concretely, we use linear scalarization, and we start from the premise that the preferences over different objectives are known a priori. In the taxonomy proposed by Roijers et al. (2013), this implies discovering a single policy (see (1) in Figure 2.7).

In the following we describe the proposed methodology (Section 5.2.1) and expose two application cases (sections 5.2.2 and 5.2.3).

5.2.1 The Methodology

Once the naive grammars are defined (following, for example, an atomic approach like the one used in 5.1), the proposed methodology consists mainly on two tasks:

definition of rewards and definition of features. Here we illustrate them by means of the compactness problem described earlier in this chapter (sections 5.2.1.1 and 5.2.1.2). We also study some aspects related to the performance of the methodology applied to the compactness problem in Section 5.2.1.3.

5.2.1.1 Definition of Rewards

In order to use a reinforcement learning process to discover a rule application policy, we need to develop a software routine that provides a quantitative evaluation of a potential design solution, according to the design criteria that define the problem. These routines will be used as rewards in the reinforcement learning process.

In the small example introduced before we seek to obtain compact shapes. As we explained, this design criterion can be formalized with the expression $area/perimeter^2$, which can be used to define the reward routine. Let us now suppose that we want shapes with nine tiles. The reward would be determined by means of the following expression:

$$r(s,a) = \begin{cases} area(s')/perimeter(s')^2 & \text{if } s' \text{ has nine tiles} \\ 0 & \text{otherwise} \end{cases} \tag{5.1}$$

Where s' is the state that represents the shape generated by applying the action a over the state s. For the intermediate shapes with less than eight tiles, the reward is defined as zero, as the compactness factor of these states is not relevant. Nonetheless, a good policy will avoid the generation of shapes many tiles in a row, not because they are not compact enough, but because they cannot lead to good solutions.

If more than one criteria is sought, we need a different routine for every design requirement. The reward in this case is thus a vector instead of a scalar number, so we are facing a multi-objective reinforcement learning problem. As we mentioned before, in this methodology we tackle the problem from an *utility-based* approach, and thus assume that we can use a scalarization function in order to aggregate the outcomes of the different routines and hence obtain a scalar reward. Here we propose the use of a linear scalarization function in order to do this, so we need a weight for every design requirement.

For example, let us suppose that we also want to maximize the perimeter of the generated shapes, which leads to a situation with two utterly conflicting goals, since if we want shapes with a high compactness factor we are actually searching to minimize their perimeter. The reward could be determined by the following expression:

$$r(s,a) = \begin{cases} w_1 \times (area(s')/perimeter(s')^2) + w_2 \times perimeter & \text{if } s' \text{ has nine tiles} \\ 0 & \text{otherwise} \end{cases} \tag{5.2}$$

Where w_1 and w_2 represent the preferences over the two objectives. If more importance is given to the first goal, then $w_1 > w_2$, and vice versa. Here we suppose that we know these preferences a priori, so we can just run a single reinforcement learning process with its reward defined with the help of the corresponding weights, and obtain a single policy as solution.

5.2.1.2 Generalization

As we have already mentioned, working with shape grammars introduces the challenge of dealing with a vast state-action space. In these cases, using a table to store the values associated to (state, action) pairs can lead to some problems:

- The large memory space requirements that such a big table would need
- The large time requirements that the learning algorithm would need to learn every position of the table

Generalization techniques can help to overcome these issues. First, the use of a function instead of a table reduces the memory space requirements in a drastic manner. For example, if we use a linear function, we just need to store the linear coefficients. Second, as many different shapes can share the same set of features, when we update the value for a given set of features, we are learning (generalizing) over all similar shapes. Thus, if an adequate set of features is selected, proper Q-values can be learnt even for shapes that were never visited during the learning process, representing them by means of the following expression:

$$Q(s,a) = \theta_1 \times f_1(s,a) + \theta_2 \times f_2(s,a) + \cdots + \theta_n \times f_n(s,a)$$

Where the coefficients θ_i are learned through the reinforcement learning process.

There is another issue, specific to the problem of generating design solutions, that generalization techniques can help with. This is related to the desirability of obtaining diverse design solutions for a given set of criteria. If we use a table to store the Q-values, the associated policy will produce solutions that are likely to be identical. This is due to the fact that the values of different positions of the table are probably different between them. This lack of diversity would only disappear if there were draws in some positions of the table, e.g., if $Q(s, a_1) = Q(s, a_2)$, that could be resolved by a random choice between the action a_1 or the action a_2. However, although the real optimal Q-values were identical, it would probably take a long time for the learning algorithm to learn the Q-values with the accuracy needed for this sake. We could establish some threshold for comparing Q-values, but the use of generalization eliminates this problem: as many different shapes share the same set of features, their associated Q-values would also be identical, yielding the draws that we need in order to produce a diversity of solutions.

In the system proposed here we will use the linear version of Watkins's $Q(\lambda)$ algorithm illustrated in Figure 2.6. As we explained in Section 2.3.3, this TD(λ) technique is more suitable to be used with function approximation than a pure TD one, given the divergence issues that arise when combining generalization methods with bootstrapping algorithms like Q-learning.

Although the features of the algorithm described in Figure 2.6 are binary, in some cases our features will be real numbers ranging between 0 and 1. The expression $\forall i \in F_a : e(i) \leftarrow e(i) + 1$ will then be replaced with $\forall i, 1 \leq i \leq n : e(i) \leftarrow e(i) + f_i(s, a)$, where $f_i(s, a)$ is the i-th feature of the state-action pair (s,a), and the expression $Q_a \leftarrow \sum_{i \in F_a} \theta_i$ is replaced with $Q_a \leftarrow \sum_{i=1}^{n} f_i(s, a)\theta_i$.

Let us use our small example in order to illustrate the process. The design requirement in this case is a high compactness factor (or, equivalently, a high compactness factor and a high perimeter with weights of 1 and 0, respectively). The general intuition behind the definition of the features is that they have to be related via a linear function in order to maximize the final reward. In this case, the use of a linear function of the number of modules with 1, 2, 3 and 4 neighbours makes sense, as compact shapes will probably have a certain combination of the following features:

- $f_1(s, a)$ = number of modules of 1 m^2 with 1 neighbour in s'
- $f_2(s, a), f_3(s, a), f_4(s, a)$ analogously to $f_1(s, a)$ with 2, 3 and 4 neighbours respectively

As these features are continuous, they will be normalized between 0 and 1, by dividing by the total number M of modules of 1 m^2 that are going to exist at the end of one learning episode. In the following section we illustrate the learning process and the results obtained by applying scalar reinforcement learning to our small problem.

5.2.1.3 Performance

In this section we discuss the performance of the described methodology applied to the compactness problem, regarding issues related to the learning process and the execution of a learned policy.

For our geometric problem of maximizing the compactness factor of shapes with 9 tiles, the algorithm learns how much the defined features relate to each other, by discovering the θ_i coefficients associated with each feature f_i. We have used the following configuration of parameters: $\alpha = 0.1, \epsilon = 0.3, \lambda = 0.5$ and $\gamma = 0.8$. After a particular execution of the algorithm with these parameters, the obtained coefficients are the following: $\theta_{1,1} = 0.0123$, $\theta_{1,2} = 0.0221$, $\theta_{1,3} = 0.0606$, $\theta_{1,4} = 0.1135$. For shapes with 9 tiles, the optimal accumulated reward is the one associated with a squared shape with a perimeter of 12, yielding a compactness factor of $9/12^2 = 0.0625$. As we are using a discount factor $\gamma < 1$, then the maximum reward is discounted in a factor of $\gamma^{(n_{steps}-1)}$, where $n_{steps} = 8$, as we have to apply 8 rules in order to get shapes of 9 tiles from the axiom. This gives us an optimal discounted accumulated reward of 0.0131.

In order to evaluate the learning process, we follow a *train/test perspective* (Kaelbling et al., 1996), suitable when the obtained policy is to be used in an off-line setting and thus the most important factor to measure is the final quality of the learned solution, as in our case, as we do not care about the accumulated rewards obtained during the learning process as far as, eventually, the learned policy is a good one. This train/test performance is measured by executing the policies learned at certain intermediate moments of the learning process, without exploration, and registering the accumulated rewards of the obtained shapes. In our case, we take this measurement every 100 episodes. As different applications of the same policy can potentially yield different shapes due to draws between features, we execute each intermediate policy 10 times and average the obtained rewards. However, in this case the optimal policy should always produce the same shape, that is, a square of 3×3 tiles. This measuring has been repeated over 10 runs of the learning process. In Figure 5.5 we illustrate the learn-test performance of the learning process.

Our particular choice of parameters ($\alpha = 0.1, \epsilon = 0.3, \lambda = 0.5$ and $\gamma = 0.8$) is supported by a sensitivity analysis that was performed in order to study the robustness of the technique regarding variations of each parameter. The technique turned out to be robust for different standard values of α and ϵ. Algorithm $Q(\lambda)$ also proved to be robust regarding parameter λ in this domain, returning good designs for all values. A trade-off value of $\lambda = 0.5$ was chosen to take advantage of the speed provided by bootstrapping methods and the safety of Monte-Carlo techniques (see Sutton & Barto (1998), Section 8.6). The algorithm was only sensitive to variations in the discount rate γ. Concretely, the algorithm did not perform well for $\gamma = 1$ (see 5.6(b)), maybe because the interactions of this rate with the convergence of the learning process, pointed out by Thrun & Schwartz (1993). The performance for $\gamma = 0.9$ (see 5.6(a)) was very similar to the one for $\gamma = 0.8$ (5.5). Notice that for $\gamma = 0.9$, the optimal

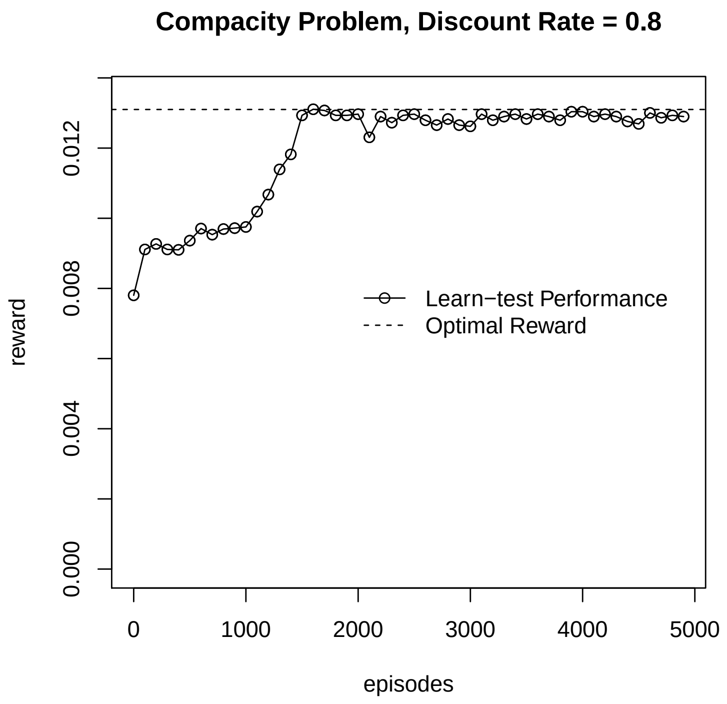


Figure 5.5: Train/test performance for the compactness problem

accumulated discounted reward is 0.0299, and for $\gamma = 1$ is 0.0625.

In order to clarify how the learnt policy guides the rule derivation process, we will go through one step of the guided application of the rule in Figure 3.4. Let us suppose that the derivation has yielded the shape in Figure 5.7.

There are many possible transformation alternatives for the next application of the rule. Each transformation would yield a different shape, but many of these shapes share the same features. In order to represent all the shapes that share the same set of features, we use patterns according to the following method: we mark with a cross all the positions where one application of rule 1 could add the extra black label of its right part (see Figure 5.11). For example, in Figure 5.8 we can see the pattern for two shapes that share the following features:

- $f_{1,1}(s, A) = 1/M$
- $f_{1,2}(s, A) = 5/M$
- $f_{1,3}(s, A) = 1/M$
- $f_{1,4}(s, A) = 2/M$

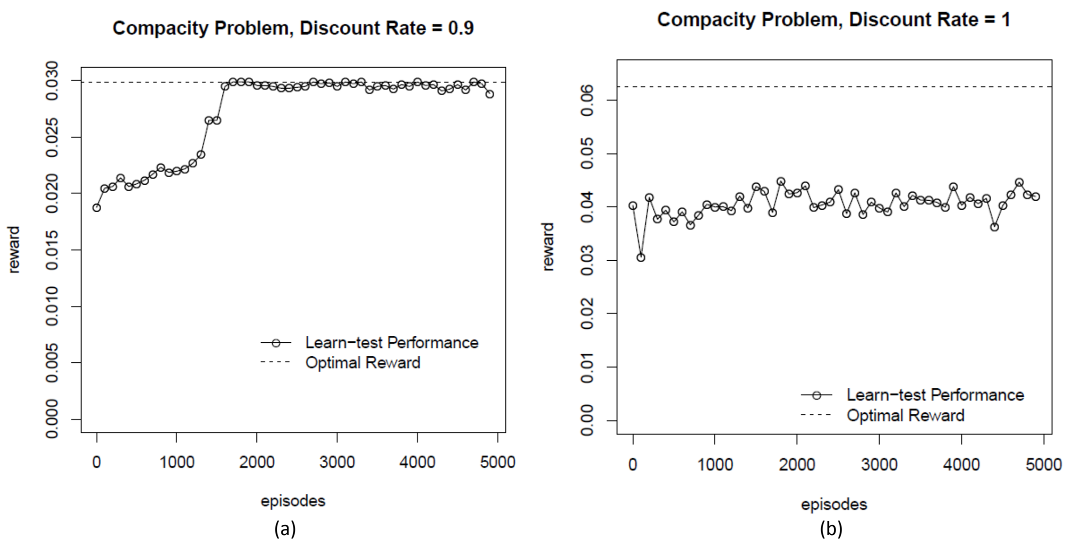


Figure 5.6: Train/test performance for the compactness problem, for different discount rates

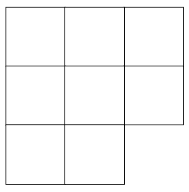


Figure 5.7: Shape with 8 tiles

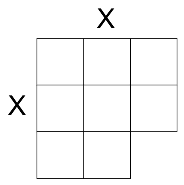


Figure 5.8: Pattern that represents two shapes obtained from the shape of Figure 5.7 sharing the same features

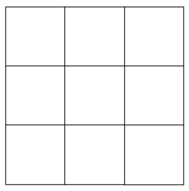


Figure 5.9: Shape represented by the best pattern obtained from the shape of Figure 5.7

Where s is the shape depicted in Figure 5.7 and A is the set of transformations of the rule that yield some of the shapes included in the pattern of Figure 5.8 (in this case $|A| = 2$). We can now compute the value for these features by means of the learned coefficients:

$$Q_1(s, A) = \theta_1 f_1(s, A) + \cdots + \theta_4 f_4(s, A) = 0,1843/M$$

The application of the rule to the shape in Figure 5.7 yields four different combinations of features. In Table 5.1 we gather the four corresponding patterns along with their features and the computed value.

In light of the computed values gathered in Table 5.1, the chosen transformation is one of those that yields the pattern in the first row of Table 5.1 (that is, we randomly pick one action of the set A_1), because it produces the higher value. This pattern represents the two shapes depicted in Figure 5.9.

5.2.2 Application Case 1: Generation of Habitable Housing Unit Designs

In this section we describe a real-world application for the design of single-family housing units (Ruiz-Montiel et al., 2011a, 2013a), using the previously described system. In the first phases of this design process, where the architect faces the *blank page syndrome* but also has to consider many design requirements, a system that proposes many distinct, feasible and even unexpected starting points can be of great help, serv-

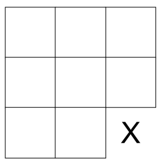
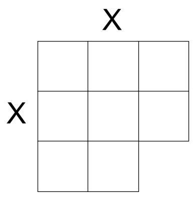
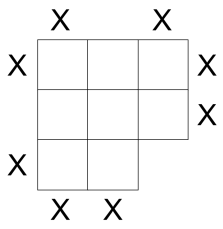
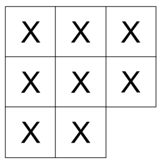
Pattern	**Features**	**Value**
	$f_1(s, A_1) = 0$ $f_2(s, A_1) = 4/M$ $f_3(s, A_1) = 4/M$ $f_4(s, A_1) = 1/M$	$0,2152/M$
	$f_1(s, A_2) = 1/M$ $f_2(s, A_2) = 5/M$ $f_3(s, A_2) = 1/M$ $f_4(s, A_2) = 2/M$	$0,1843/M$
	$f_1(s, A_3) = 1/M$ $f_2(s, A_3) = 4/M$ $f_3(s, A_3) = 3/M$ $f_4(s, A_3) = 1/M$	$0,1851/M$
	$f_1(s, A_4) = 0/M$ $f_2(s, A_4) = 5/M$ $f_3(s, A_4) = 2/M$ $f_4(s, A_4) = 1/M$	$0,152/M$

Table 5.1: Patterns yielded by one application of rule 1 to the shape in Figure 5.7

ing as inspiration sources. The expression “inspiration source" describes the conscious use of different resources or even previous designs, as references for the solution to a problem (Eckert et al., 2010). Our methodology can provide such set of starting points or inspiration sources, that can be considered as triggers for new ideas and can speed up and facilitate the design process.

5.2.2.1 Housing Unit Design

The design of housing units, either single or multi-family, is a strongly constrained process. The nature of such constraints ranges from issues such as the area of each space, to adjacency relationships between them and, of course, more detailed constraints particular to each different space. In this work we have started from the housing program proposed by the studio Montaner & Muxí (Montaner Muxí arquitectes, 2008) for the regional government of Andalusia (Spain). This program details the criteria that a basic house must fulfil depending on the number of inhabitants. By a *basic house* we understand a house that, besides satisfying some minimum habitability conditions, also offers some adaptability, that is, its spatial composition may be modified if the number of inhabitants rises. The final outcome of our experiments will be two-dimensional floor distribution schemes (in the following, *schemes*) of basic, two-person housing units. The produced schemes are distributed in just one floor.

Albeit working with floor plans does not guarantee a global approach to architectural design, it allows the establishment of relationships between spaces, the generation of circulation spaces and the definition of static places. It also allows a dimensional quantification regarding area and longitudinal measures, and therefore also concerning the house scale, making it compatible with constructive and structural dimensions that approach reality.

In the housing proposal of Montaner, several kinds of spaces that must be present in a basic house are established. Among these we have considered three main categories: 1) specialized spaces (which need specific installations), 2) non-specialized spaces (do not need specific installations, and their use is determined by the inhabitants: dining-room, living-room, bedroom) and 3) complementary spaces (such as the *distribution hall*, that allows circulation between spaces).

Table 5.2 summarizes the imposed explicit constraints. The nature of the house contour is not constrained by our guidelines (Montaner Muxí arquitectes, 2008), but in the context of a computational system we must set some additional criteria in order to avoid the generation of unfeasible contours. In our experiments we have favoured *compact* contours, (i.e. contours with a high area/perimeter ratio) over scattered ones, because the latter might make the process of establishing the internal layout of the scheme very difficult.

Global requirements
R1: Total area must be of at least 46 m^2
R2: The contour must be compact
Kitchen requirements
R3: Minimum linear space must be of at least of 6 modules 60×60 cm
R4: Minimum distance between modules and walls: 1,10 m
R5: Minimum distance between modules: 1,10 m
Bathroom requirements
R6: At least two modules of 90×180 cm must exist
Non-specialized spaces requirements
R7: The area of each non-specialized space must be bigger than 9 m^2
R8: A 2,8 m-diameter circle must be inscribed inside each non-specialized space
Complementary spaces requirements
R9: A support space that allows the circulation between spaces must exist

Table 5.2: Requirement set for a single-family basic house (adapted from Montaner Muxí arquitectes (2008))

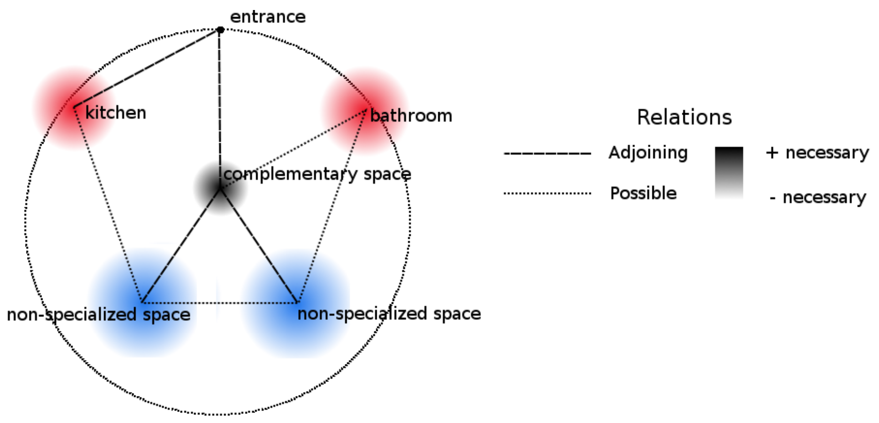


Figure 5.10: Proximity relationships in a single-family house (adapted from Montaner Muxí arquitectes (2008))

In Figure 5.10 proximity relationships between spaces are represented graphically. The circle represents the external wall of the unit. Dotted lines establish *possible* relations, that differ from the striped line (*adjoining* relation) in the level of importance (possible relations are less necessary than adjoining ones). Our system also considers an additional constraint that is not explicitly mentioned in the housing proposal, but it is desirable to take it into account in the design of a housing unit: the entrance to the house must be near both the kitchen and the distribution hall, and not very close to the bathroom.

Working with schemes of housing units for two people gives rise to some considerations about intimacy and privacy that have implications in the existence of internal partition walls. Sharing a two-person housing unit involves a high level of intimacy between the inhabitants, and thus less privacy is needed than in the case of a higher number of people living in the house, so it is possible to relax the existence of internal partitions. The most necessary walls are those that isolate the bathroom from the rest of the house. Even walls isolating the kitchen are not essential. The walls that delimit non-specialized spaces such as living rooms or rooms are optional, and demarcation can be achieved by means of proper pieces of furniture.

Here we have exposed the criteria that will guide the housing unit generation process in our system. Knowledge in Table 5.2 and in Figure 5.10 will be formalized in the form of naive rules and, much more intensively, rewards, that are numerical expressions that represent the quantities to be maximized by the learning algorithm.

5.2.2.2 Naive Grammars and Phases of Generation

We have defined several design phases that establish an ordering in the placement of the different architectural elements. Each phase can be described as follows:

- Phase 1: Generation of a contour.
- Phase 2: Labelling the distribution hall.
- Phase 3: Placing the kitchen modules.
- Phase 4: Placing the bathroom modules.
- Phase 5: Labelling non-specialized spaces.
- Phase 6: Labelling the entrance.

This division in phases allows us to simplify the problem, since each phase needs to care only about the relevant set of requirements.

Figure 5.11 illustrates the grammars for phases 1-6 (there is a maximum of two rules per phase). The input to the first phase is a simple axiom: a suitably labelled module of one square meter (1 m^2). The shape generated in the n-th phase is the input for the $(n+1)$-th phase.

In the following we explain how each rule works. Rule 1 is in charge of creating the contour. White labels are used to mark the added walls, and black labels mark the interior of the housing unit. In Figure 2.1 we can see an example of how this rule enlarges the small contour delimited by the axiom. Some residual walls can arise from the application of rule 1, thus we use rule 2 in order to delete them. As we can see in Figure 5.11, the left-hand side of rule 2 ensures that it will only erase inner walls, keeping the walls delimiting the contour untouched. Rule 3 puts the label that marks the location of the distribution hall somewhere in the interior of the housing unit, which is marked with black labels. Rules 4 and 5 are in charge of placing the kitchen modules (that is, the pieces of furniture that will delimit the kitchen space).

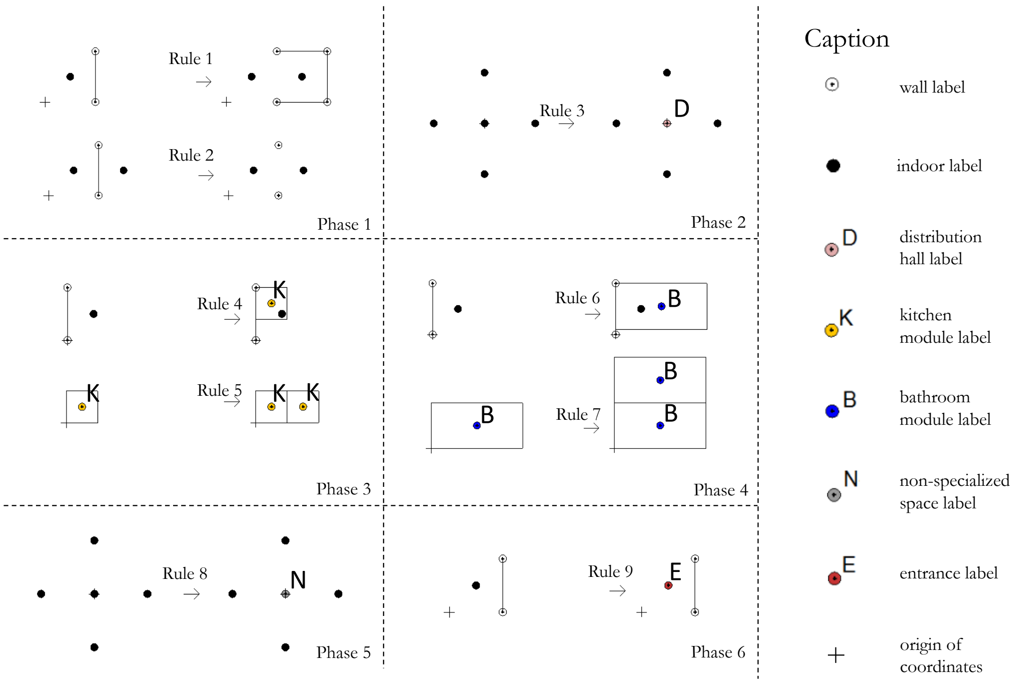


Figure 5.11: Naive grammars for phases 1-6

In particular, rule 4 adds the first module, which is placed right next to a contour wall. Rule 5 adds the rest of the modules, just adding one module next to an existing one. Rules 6 and 7 are similar to rules 4 and 5, but they refer to bathroom modules. Notice that the dimensions of these modules (both for the kitchen and the bathroom) are directly extracted from the housing program of Montaner (see Table 5.2). Rule 8 marks a non-specialized space, and it works exactly as rule 3, except for the kind of label in its right-hand side. Finally, rule 9 marks the entrance to the housing unit, placed right next to a contour wall.

Rules are executed in order (the ordering is given by the rule numbers) and repeatedly applied as long as possible, or until a final state test has been satisfied. This final state test depends on each rule. Tests for every rule are gathered in Table 5.3.

If this set of rules is used without further guidance to generate schemes, then when executing each rule, all possible transformations are computed and one of them is selected at random. Two examples of schemes generated by means of this random process are depicted in Figure 5.13. As we will discuss in Section 5.2.2.4, the schemes are not actually admissible, highlighting the need of extra guiding mechanisms in order to comply with housing unit requirements.

Rule	Final state test
1	Total area = 46 m^2
2	Execution continues until the rule cannot be applied any longer
3	A distribution hall label exists
4	The first kitchen module is placed
5	Six kitchen modules are placed
6	The first bathroom module is placed
7	Two bathroom modules are placed
8	Two specialized labels exist
9	The entrance to the house is placed

Table 5.3: Final state tests for every rule

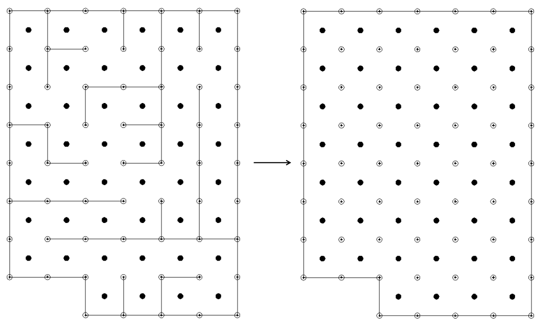


Figure 5.12: Effect of the application of rule 2

5.2.2.3 Learning Processes for Naive Shape Rules

In order to generate valid designs, we must use an heuristic (policy) that determines how to apply the rules. These policies will select the most suitable transformation to be applied at each step in order to produce feasible solutions.

Nevertheless, not every rule in Figure 5.11 needs the use of a policy. For example, rule 2 cannot perform badly, since it only erases residual walls that appear from the repeated execution of rule 1 (see Figure 5.12).

Rule 3, which is in charge of placing the label for the distribution hall, can also perform well randomly, since when it is applied, none of the elements of the scheme has been placed yet. So this label can be placed at any point inside the scheme, and then the rest of the elements will be placed considering its situation.

Rules 1, 4, 5, 6, 7, 8 and 9 need to be guided by a policy in order to perform properly, because they have to meet requirements that have not been considered inside the rules due to their naive nature.

These policies will be learnt through reinforcement learning. The rewards that govern the learning process are determined from the conditions described in Section 5.2.2.1, concretely the ones in Table 5.2 and Figure 5.10. We also need to define features, because we deal with a vast state space and thus storing the value of each state in a table is not practical; therefore, we will learn a function of the features that returns the values of states.

The first phase is equivalent to the geometric problem we have used previously in order to illustrate the proposed methodology, in which we seek to obtain shapes with a high compactness factor. The reward for this phase is thus this compactness measure, and the features are the same that have already been described, that is, number of tiles with 1, 2, 3 and 4 neighbours.

The rewards for the learning process of rules 4-9 are defined as follows: the more requirements a state complies with at a certain step, the bigger reward it gets. An advantage of dividing the process in distinct phases is that we only have to consider subsets of requirements.

For example, let us consider rule number 5, which is in charge of placing the kitchen modules. It is depicted in figure 5.11. As we can see, the rule simply states that an additional module can be placed next to an existing kitchen module. In this case, we have considered six requirements that are evaluated as 1 or 0 depending on whether they are fulfilled or not in the shape corresponding to a given state. The first one is a geometric constraint: every module must lie inside the contour. The others are direct translations of constraints imposed by the guideline Montaner Muxí arquitectes (2008):

- $r_{5,1}(s) = 1$ if every module is inside the contour.
- $r_{5,2}(s) = 1$ if every module is accessible.
- $r_{5,3}(s) = 1$ if the distance between modules and walls is bigger than 1,1 m.
- $r_{5,4}(s) = 1$ if the distance between non-contiguous modules is larger than 1,1 m.
- $r_{5,5}(s) = 1$ if the modules are at a proper distance from the distribution hall (more than 1,2 m and less than 6).
- $r_{5,6}(s) = 1$ if there are at least six modules.

The scalarized reward $r(s)$ of a state (a shape) s in the context of rule 5 is computed as the following sum:

$$r_5(s) = 3 * r_{5,1}(s) + r_{5,2}(s) + r_{5,3}(s) + r_{5,4}(s) + r_{5,5}(s) + r_{5,6}(s)$$

As we can see, some preferences (weights) have been used to define this reward: we have given more importance to the fulfilment of the first requirement (the one that determines if the modules are inside the contour).

To define the features for this phase, we have decided to identify a feature with every single requirement, so six binary features were considered for the shapes generated by this rule. Each feature is computed over the shape s' produced by applying action a to the shape of state s:

$$f_{5,i}(s,a) = r_{5,i}(s'), 1 \leq i \leq 6$$

So the learned value $Q_5(s,a)$ is determined by the function:

$$Q_5(s,a) = \theta_{5,1} \times f_{5,1}(s,a) + \cdots + \theta_{5,6} \times f_{5,6}(s,a)$$

Using this kind of features implies that states that correspond to different shapes can have the same features and thus the learning agent does not distinguish between them. The way the features are defined, sometimes a rule application may not result in a change of the features and it may seem that the agent cannot observe the immediate consequences of applying the rule. However, this just means that the rule application does not increase neither decrease the quality of the shape. As the rules that we are using are additive, the underlying shapes do change with each rule application and, in the end, a final state (which, in the previous example, is identified by the sixth feature) will be reached.

The reward is defined for both intermediate and final states, and an action that does not result in a change any of the features of an intermediate state still receives a reward. Although it may seem that the quality of intermediate steps is not important, if we also reward high-quality intermediate steps (even when there are no difference in their features), then the final shapes can be better in architectural terms: it can happen that the final shape of a given execution does not comply with some criteria, but if we have favoured high quality intermediate states, then this final shape is probably nearer to a good solution than if we had not favoured these good intermediate shapes. This can be easily illustrated in the light of the previous example. Let us imagine that every kitchen module is accessible until the fifth rule application, but the last action spoils the accessibility. The second feature for the last shape is thus zero. Now imagine another rule application in which the modules were accessible only during the third first shapes in the sequence, so the second feature for the last shape is also zero. However, the first sequence of actions is better because it leads to a final shape that is nearer to our quality standards and can be easily modified to be a good one. The agent can distinguish these situations thanks to the rewards of the intermediate states and thanks to the fact that positive rewards are received even when the applied action does not result in a change of the features.

In the end, the algorithm will learn the adequate coefficient θ for each feature f

(that is, the policy), so as to determine which features must be pursued first in order to maximize the accumulated reward obtained at the end of the rule application.

The complete set of requirements that define rewards for rules 1,4-9 is gathered in Table 5.4, and the scalarized reward expressions used for each rule are gathered in Table 5.5. All the requirements are binary, i.e one or zero when the associated predicate is satisfied or not respectively.

Features for rules 4-9 have been defined according to the expression: $f_{i,j}(s,a) = r_{i,j}(s')$, as we have explained for rule 5. As we explained before, other approaches for features definition are possible (apart from directly translating design requirements). This is the case of rule 1, for which the design requirement is a high compactness factor, and we have not directly defined the features from this requirement, as in rules 4-9.

Except for the first rule, for which the axiom (initial state in the learning process) is always the same (one module of 1 m^2), for the rest of the rules the initial states can be quite different. For example, in the case of rule 6, the initial state is the house contour with the label of the distribution hall and the kitchen modules already placed. The distribution hall and the kitchen modules could be in many different places, so it would be illogical to always start the learning process with the distribution hall and the modules exactly in the same place. It is better to use a set of distinct initial states for every learning process. These initial states are generated by means of the previous rules, using the already learnt policies when necessary.

The coefficients for each policy were initialized to arbitrary values (in our case, they were set to zero) before the learning process. After running the reinforcement learning algorithm, the values for these coefficients are learnt, with the objective of maximizing the total accumulated reward. The learnt coefficients are gathered in table 5.6.

For rule 1, the learnt coefficients establish a combination of the number of tiles with 1, 2, 3 or 4 neighbours. For rules 4-9, the learnt coefficients give importance to each feature regarding the maximization of the accumulated reward. Possibly not all requirements are going to be fulfilled at the end (that is, not every feature is going to be 1-valued), so the learnt coefficients give insight to determine which requirements must be pursued first in order to maximize the final total reward.

When all the policies have been learnt, we can automatically produce designs with the rules guided by their policies when these are present (that is, in rules 1, 4, 5, 6, 7, 8 and 9). In Section 5.2.2.4 we show some obtained results.

Rule 1 (generating the contour)
$r_{1,1}(s) \rightarrow$ the compactness factor has to be maximized
Rule 4 (placing the first kitchen module)
$r_{4,1}(s) = 1$ if the module is at a proper distance from the distribution hall (more than 2 m and less than 4)
$r_{4,2}(s) = 1$ if the module is separated from the walls by at least 1,1 m
Rule 5 (placing the rest of the kitchen modules)
$r_{5,1}(s) = 1$ if every module is inside the contour
$r_{5,2}(s) = 1$ if every module is accessible
$r_{5,3}(s) = 1$ if the distance between modules and walls is bigger than 1,1 m
$r_{5,4}(s) = 1$ if the distance between non-contiguous modules is larger than 1,1 m
$r_{5,5}(s) = 1$ if the modules are at a proper distance from the distribution hall (more than 1,2 m and less than 6)
$r_{5,6}(s) = 1$ if there are at least six modules
Rule 6 (placing the first bath module)
$r_{6,1}(s) = 1$ if the module does not overlap with kitchen modules
$r_{6,2}(s) = 1$ if the module is at a proper distance from the distribution hall (more than 2 m and less than 4)
$r_{6,3}(s) = 1$ if the module is separated enough from the walls
Rule 7 (placing the rest of the bath modules)
$r_{7,1}(s) = 1$ if the modules do not overlap with any kitchen module
$r_{7,2}(s) = 1$ if the modules are at a proper distance from the distribution hall (more than 2 meters and less than 4)
$r_{7,3}(s) = 1$ if the modules are separated enough from the walls
$r_{7,4}(s) = 1$ if there are at least two modules
Rule 8 (labelling non-specialized spaces)
$r_{8,1}(s) = 1$ if in each label a 3 meter-diameter circle can be centred, without overlapping with walls or modules
$r_{8,2}(s) = 1$ if there is a label separated from the bath by less than 4,5 m
$r_{8,3}(s) = 1$ if there is a label separated from the kitchen by less than 4,5 m
$r_{8,4}(s) = 1$ if there are 2 non-specialized labels separated at least by 3 m
$r_{8,5}(s) = 1$ if the distance from each non-specialized label to the distribution hall label is larger than 2 m and shorter than 6
$r_{8,6}(s) = 1$ if when there is one single non-specialized label, then there is enough space for another one
Rule 9 (labelling the entrance)
$r_{9,1}(s) = 1$ if the entrance is separated by at least 1 m from every kitchen module
$r_{9,2}(s) = 1$ if entrance is less than 2 m from some kitchen module
$r_{9,3}(s) = 1$ if the entrance is separated by at least 4 m from every bathroom module
$r_{9,4}(s) = 1$ if the distance from the entrance to the distribution hall label is shorter than 4 m

Table 5.4: Requirements for rules 1-9

Rule	**Reward**
1	area/perimeter^2
4	$r_{4,1}(s) + r_{4,2}(s)$
5	$3 * r_{5,1}(s) + r_{5,2}(s) + r_{5,3}(s) + r_{5,4}(s) + r_{5,5}(s) + r_{5,6}(s)$
6	$3 * r_{6,1}(s) + r_{6,2}(s) + r_{6,3}(s)$
7	$3 * r_{7,1}(s) + r_{7,2}(s) + r_{7,3}(s) + r_{7,4}(s)$
8	$3 * r_{8,1}(s) + r_{8,2}(s) + r_{8,3}(s) + r_{8,4}(s) + r_{8,5}(s) + r_{8,6}(s)$
9	$r_{9,1}(s) + r_{9,2}(s) + r_{9,3}(s) + r_{9,4}(s)$

Table 5.5: Rewards for every learning process

Rule 1
$\theta_{1,1} = 0,0037$; $\theta_{1,2} = 0,0044$; $\theta_{1,3} = 0,0338$; $\theta_{1,4} = 0,0624$
Rule 4
$\theta_{4,1} = 9,6338$; $\theta_{4,2} = 10,9915$
Rule 5
$\theta_{5,1} = 11,3647$; $\theta_{5,2} = 5,0703$; $\theta_{5,3} = 6,3998$; $\theta_{5,4} = 6,9184$; $\theta_{5,5} = 7,581$; $\theta_{5,6} = 1,7631$
Rule 6
$\theta_{6,1} = 13,9956$; $\theta_{6,2} = 10,9998$; $\theta_{6,3} = 10,7779$
Rule 7
$\theta_{7,1} = 6,7972$; $\theta_{7,2} = 7,2621$; $\theta_{7,3} = 6,8873$; $\theta_{7,4} = 6,8873$
Rule 8
$\theta_{8,1} = 6,3159$; $\theta_{8,2} = 2,9472$; $\theta_{8,3} = 7,2953$; $\theta_{8,4} = 5,1918$; $\theta_{8,5} = 7,4059$; $\theta_{8,6} = 8,3831$
Rule 9
$\theta_{9,1} = 7,4246$; $\theta_{9,2} = 2,0591$; $\theta_{9,3} = 5,7704$; $\theta_{9,4} = 1,5242$

Table 5.6: Policies learnt for every rule

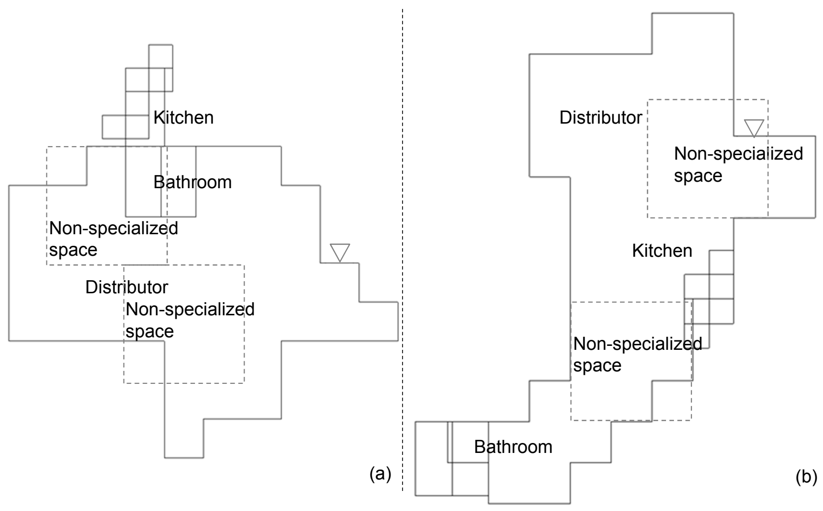


Figure 5.13: Two schemes obtained with naive grammars

5.2.2.4 Results

All tests of this application case were run on an Intel Core i7 860 @2.80 GHz processor with 8GB RAM and Windows 7 (64 bits).

5.2.2.4.1 Naive grammars without policies. A number of schemes were generated with the naive set of rules depicted in Figure 5.11, without benefiting from any process of learning. Namely, the system was used to generate 100 schemes (two of them are shown in Figure 5.13). For the final presentation of the schemes we have (1) replaced labels for the kitchen, bath, distributor hall and non-specialized spaces with suitable text, (2) replaced the entrance label with an arrow, and (3) surrounded non-specialized spaced with dotted, 3×3 m. squares.

Computation was fast. The minimum generation time for a scheme was 30,42 s and the maximum 41,88 s. The mean time was 34,87 s.

All the schemes were different, but all of them violate several requirements of the housing program described in Section 5.2.2.1. In particular, those in Figure 5.13 violate the following requirements of Table 5.2:

- R2. The contours are scattered
- R3. The kitchen modules do not form a lineal space
- R4. The distance between kitchen modules and walls is not higher than 1,1 m
- R8. A 2,8 m-diameter circle cannot be inscribed inside each non-specialized space
- R9. There is not a support space that allows the circulation between spaces.

Regarding the relevant relationships gathered in Figure 5.10, these schemes do not respect them mainly because a circulation cannot be established between the different spaces in the housing unit.

These results could be expected given the use of naive, non-expert shape grammars without further guidance. Grammar rules do not enforce the whole set of the involved requirements, and thus arbitrary execution is not likely to lead to feasible designs.

5.2.2.4.2 Naive grammars with policies. A hundred designs were produced by means of generation processes guided by policies when necessary (that is, for rules 1 and 4-9 in Figure 5.11). Only 11 out of these 100 schemes violated some of the constraints in Table 5.2. Therefore, compared to the ones generated by the naive grammars alone, these are closer to fulfil the housing program described in Section 5.2.2.1.

We arbitrarily chose 12 schemes out of the valid ones. The generated schemes are depicted in Figures 5.14 and 5.15. Computation time was fast; the minimum generation time for a scheme was 33,94 s and the maximum 44,35 s. The mean time was 38,46 s. The slight time increase with respect to the random execution of the naive grammars is due to the calculus of the feature values, that has to be performed in this policy-driven approach, but not in the random one. Nevertheless, the number of iterations (understood as rule derivations) is the same for both approaches, as the final step tests (shown in Table 5.3) are shared.

The schemes show also great design diversity. A more in-depth discussion of the quality of the results from an quantitative and architectural point of view can be found in Section 5.2.2.5.

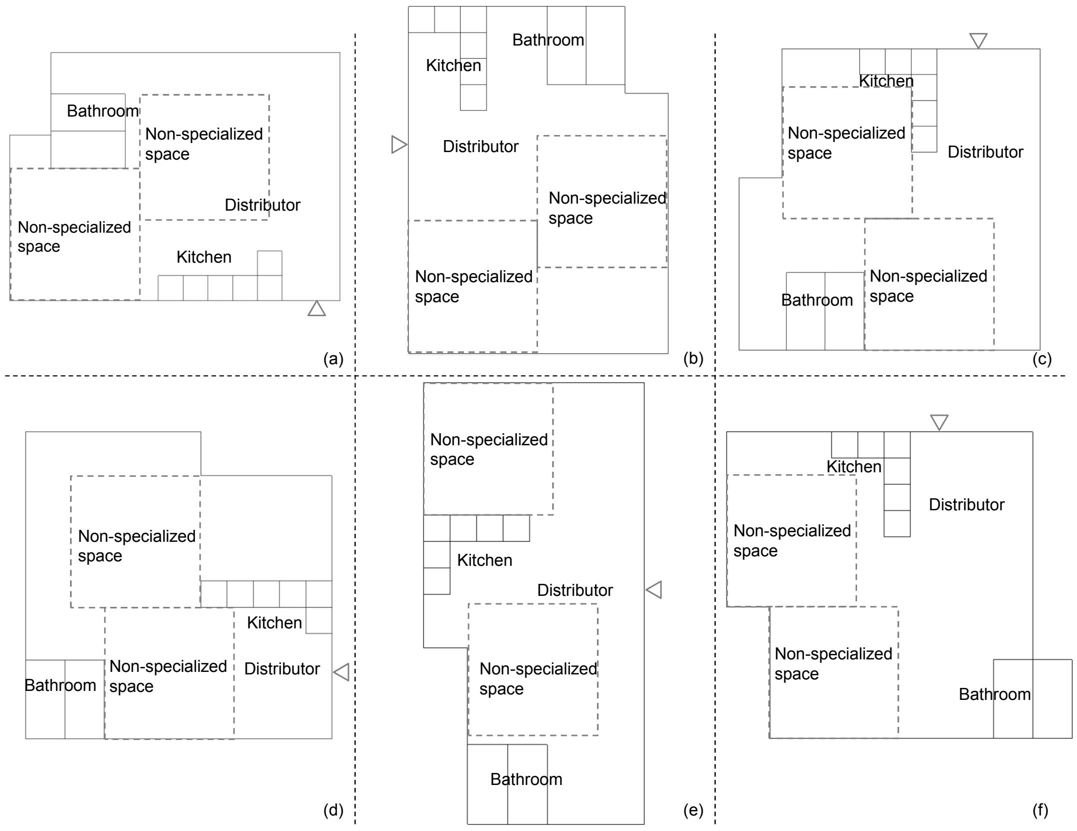


Figure 5.14: Some generated designs (results a-f)

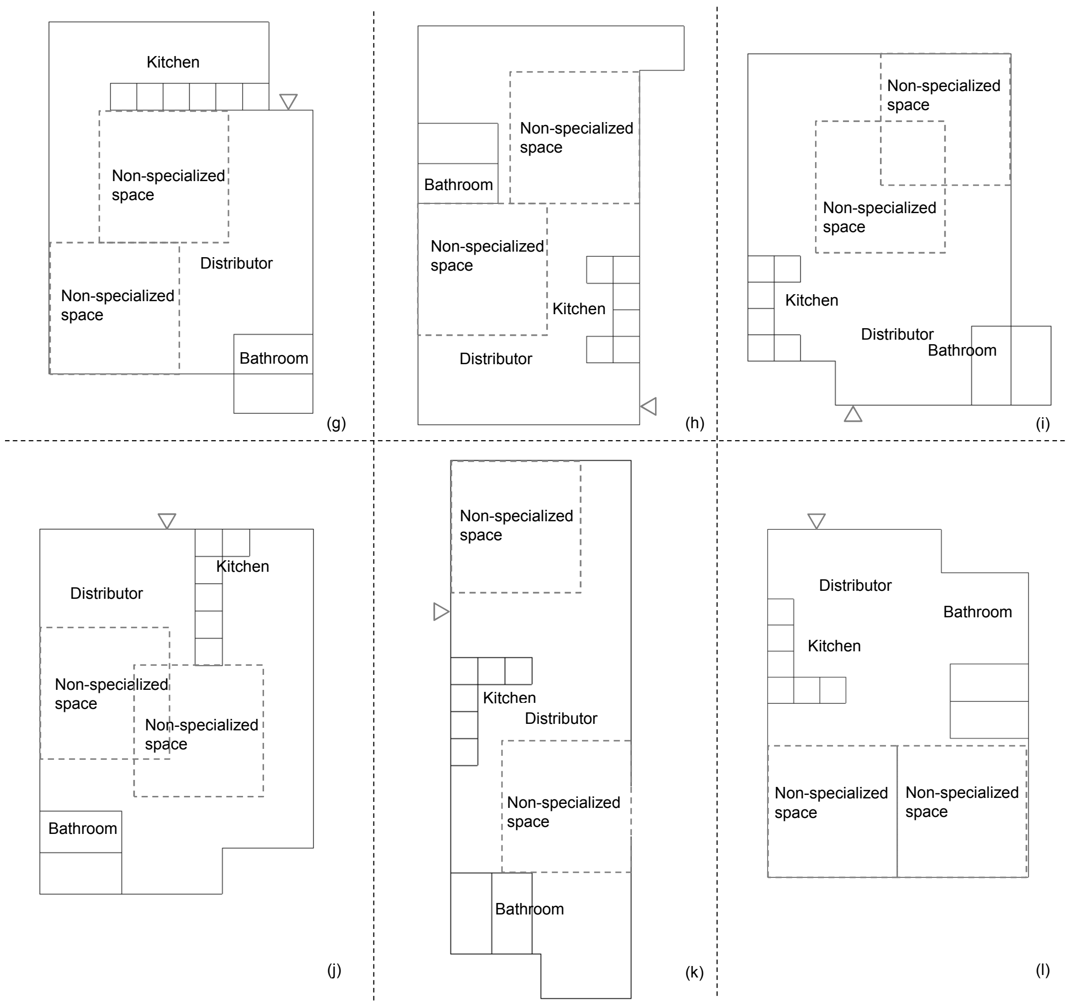


Figure 5.15: Some generated designs (results g-l)

Rule	Reward Improvement
1	2.452
4	2,2222
5	2.1667
6	1.1904
7	1.2174
8	1.6
9	1.8182

Table 5.7: Reward improvements gained thanks to the learned policies for every rule

5.2.2.5 Discussion

The results shown in section 5.2.2.4 can be discussed from several points of view. We will focus on the following issues:

1. Admissibility of generated schemes
2. Architectural evaluation of the generated schemes
3. Variability of generated schemes and generation times

5.2.2.5.1 Admissibility of generated schemes. The first goal of the system should be to generate admissible designs according to the Montaner program (that is, regarding the items in Table 5.2 and the relationships depicted in Figure 5.10). Obviously, a naive grammar generates inadmissible schemes: it is difficult for a scheme generated at random to satisfy all the requirements. It was illustrated in Section 5.2.2.4.1.

However, when policies are learnt the rules generate better designs, as the rewards considering Montaner criteria are maximized. Regarding conditions of Table 5.2 and Figure 5.10, we found that in the set of the 100 generated housing units, the requirements were violated only 22 times.

In terms of the obtained rewards, we have measured the average reward improvement that is gained thanks to the learned policies. For each rule i that can be combined with a policy, we denote $\overline{R}_i$ as the average accumulated reward obtained when the rule is applied randomly, and $\overline{R}_i^*$ as the average accumulated reward obtained when the learned policy is used to guide the rule application. The reward improvement for the rule i is then represented by the expression $\overline{R}_i^*/\overline{R}_i$. The reward improvements for every rule in this application case are gathered in Table 5.7.

5.2.2.5.2 Architectural evaluation of the generated schemes. The results in Figures 5.14 and 5.15 were studied by a team of architects. From the depicted set of schemes, two were chosen by the architects as the best alternatives, and other two ones as the less adequate. Additionally, some improvements that can be easily performed in order to obtain more plausible solutions are detailed.

Schemes (a) and (b) can perform well, since non-specialized spaces are each one combined with a specialized space (kitchen or bathroom). In (a) there is an efficient combination of entrance, kitchen, distribution hall and non-specialized space that releases space and optimize the circulation. Moreover, when we enter the house we find a wide space that gives the feeling of more spaciousness. The scheme (b) aligns the kitchen and the bathroom, which is desirable given the possibility of sharing piping installations.

On the other hand, schemes (g) and (h) are not feasible from an architectural point of view. The kitchen and the bathroom have a diagonal disposition that strangulates non-specialized spaces and makes the circulation difficult.

Schemes (e) and (k) can be easily improved by changing the location of the bathroom, moving it from the bottom to the top of the scheme. Scheme (d) can be improved by moving the kitchen to fit a corner (for example, the top-right one), and thus form a more delimited space. In scheme (j) we could change the entrance position to one more centred, in order to improve the circulation inside the house.

Interestingly, in (a) we can observe an unexpected fact concerning the entrance. Normally, this kind of housing units have the entrance more or less centred in order to make the circulation easier. Nevertheless, the entrance of scheme (a) performs very well placed in a lateral position, given the high compactness of the solution. Also, the contours of the housing units are to a great extent very unusual and inspire *grouping* or *clustering* possibilities between units. These are examples of how we can find unexpected and inspiring solutions using the shape grammar framework.

In general, except for schemes (g) and (h), all the solutions in Figures 5.14 and 5.15 are feasible with minor changes or even just as they have been generated, as is the case of (a) and (b).

It is worth questioning why the set of criteria used by the team of architects has not been considered in the assessment of schemes during the learning process. Firstly, we emphasize that this architectural evaluation is part of the intended work flow inside which our system makes sense, that is: (1) generating many feasible and varied schematic designs effortlessly, (2) studying and evaluating the proposals from the point of view of the designer, and (3) using the best generated solution(s) as seed(s) for complete architectural projects. Secondly, the criteria considered by our team of architects do not belong to any architectural guideline (unlike the requirements that have been used in the learning process, that stem from an architectural guideline), and can be described as subjective and even elusive in some cases.

5.2.2.5.3 Variability of generated schemes and generation times. An analysis of generation time and variability of solutions is interesting in the context of an interactive system that is to be used by designers who are willing to obtain a big number of different design alternatives. These alternatives will be evaluated, and in case they do not meet design criteria or they just do not satisfy the designer, he or she may wish to generate another big set of solutions.

As mentioned in section 5.2.2.4 and shown in Figures 5.14 and 5.15, the system generates different schemes for the given problem. In fact, when applying the policy, we choose the action that yields the maximum value. In order to produce a potentially distinct shape every time the policy is applied, we need more than a single action a to yield the maximum possible value so that we can pick one action out of the set of draws. As features aim at generalizing shapes, this behaviour is likely to occur: several actions may yield shapes with the same feature values (this fact is described in section 2.3, see Table 5.1). Generated designs thus turn out to be substantially different in our case.

Once the policies are learnt, the system must run in a reasonable amount of time, suitable for an interactive environment. This goal is also achieved by the rules with the learnt policies (generation times of about half a minute per scheme, see Section 5.2.2.4). Notice that generation times are very similar to those for the naive grammars alone, so the effect of incorporating learnt policies to the rules is not relevant. Most time was employed in learning processes, but this is only done once for each rule.

The knowledge intensive domain of housing unit design will benefit from this approach, as designers will be able to consider simultaneously a huge number of automatically generated starting points (possibly unexpected) that will generally meet the considered set of design criteria.

5.2.2.6 Validation

The described system has been validated in an academic context. The goal of the experiments described here is to check whether the system based on shape grammars and scalar reinforcement learning is able to assist architecture students in the early stages of design, producing schemes that provide creative insight (Millán et al., 2014).

In any design task it is not easy to start a new project, so architecture students usually struggle when they need to start a new design and face nothing but a blank sheet of paper. To this end, designers often use inspiration sources, which can act as triggers for the generation of new ideas, and even accelerate or facilitate the design process, especially in the case of novice designers. For that reason, computer-aided design tools or design assistants can be very useful in these early stages, allowing the exploration of different alternatives and providing feasible starting points.

To this end, we have developed BH-ShaDe (Basic House Shape Design), a software tool that can produce schemes by means of the system of shape grammars and scalar reinforcement learning described here. In this way, architecture students are provided with many feasible and varied starting points (basic house schemes) which have been obtained effortlessly. In order to analyse the usefulness of such starting points in the process of developing residential projects, we performed an experiment with a group of 78 architecture students of the University of Málaga.

BH-ShaDe allows students to automatically obtain housing units schemes according to Montaner and Muxi´s guideline. These schemes are automatically generated by the computer, using the shape grammar shown in Figure 5.11. Students just need to specify the desired number of schemes and then the tool will generate them.

In Figure 5.16 we can see the different modules that interact in order to provide students with the aforementioned features. The architecture of BH-ShaDe is based on the architecture of ShaDe, described in Chapter 4 (see Figure 4.7). The main difference with the architecture of ShaDe is that here we have an Automatic Generation Module that generates habitable housing units schemes. This module applies the shape grammars in Figure 5.11 by means of an ad-hoc subshape detection algorithm that only works with these shape grammars, but is much faster than the general algorithm. With the result of this algorithm, the Greedy Algorithm chooses the best rule application with the help of the policies generated by the reinforcement learning processes.

The Shape Grammars Module encapsulates the shapes, the rules and the arithmetic for manipulating shapes. These elements are used by the previously described Automatic Generation Module and also by the Interactive Generation Module, which is a limited version of the Design Generation Module of ShaDe. As we will explain later, this module is mainly intended to help the users to learn the concept of shape grammars. It also allows the interactive execution of a shape grammar that can be applied over the schemes produced by the Automatic Generation Module. This grammar is intended to delimit the non-specialized spaces with leaders, in case the users want to visualize them as closed spaces.

On the top, we can find BH-ShaDe interface, that mainly interacts with the two principal modules of BH-ShaDe (the automatic one and the interactive one). The rules of the shape grammars and the results of successive applications of rules (according to the user's choices or to the automatic generation module) are shown in the interface, as well as some tutorials and the typical file managing tasks.

The evaluation was performed with three groups of students enrolled in the subject Architectural Projects VII, which is taught in the seventh semester of the Architecture Degree of the University of Málaga. In total, 78 students participated in the

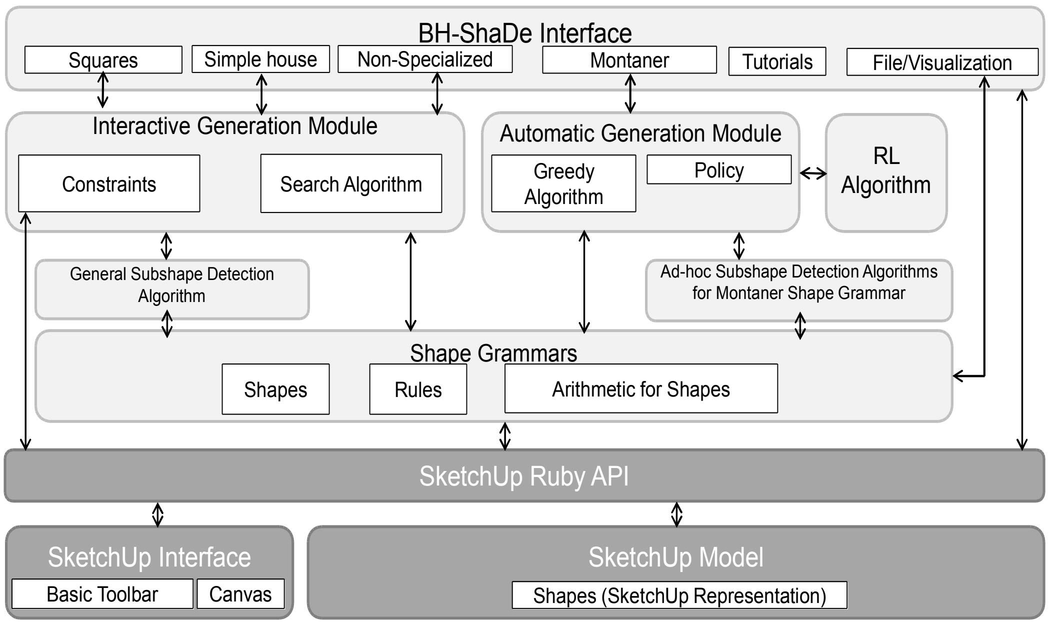


Figure 5.16: Architecture of BH-ShaDe

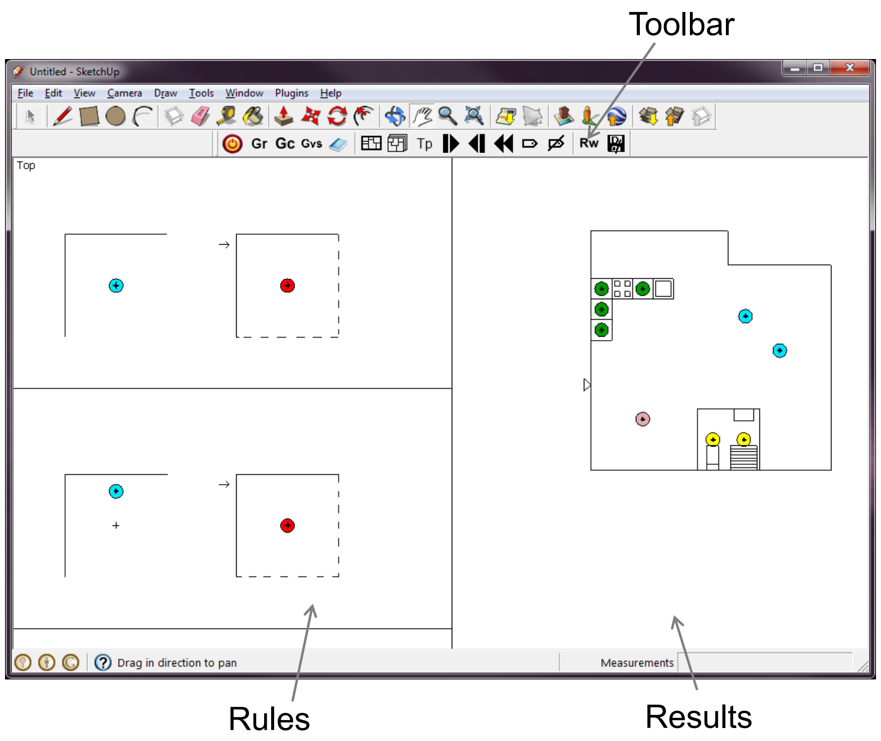


Figure 5.17: Screenshot of BH-ShaDe interface

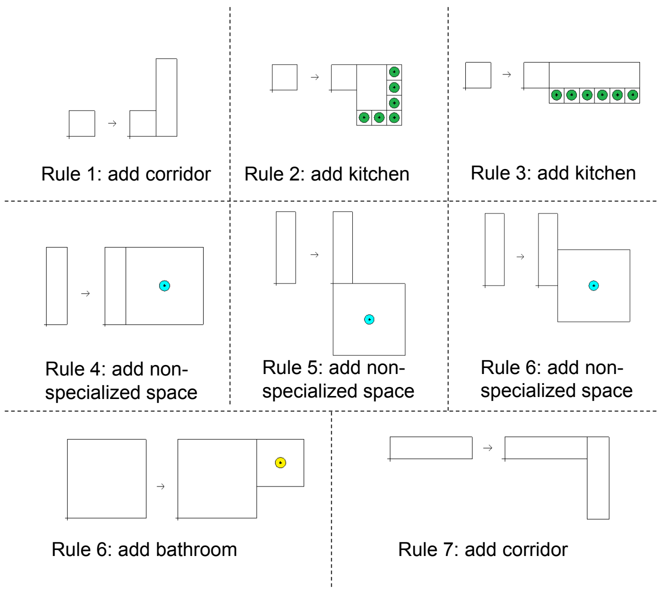

Rule 1: add corridor
Rule 2: add kitchen
Rule 3: add kitchen
Rule 4: add non-specialized space
Rule 5: add non-specialized space
Rule 6: add non-specialized space
Rule 6: add bathroom
Rule 7: add corridor

Figure 5.18: Shape grammar for the generation of a simple housing unit

experiment. Next we will describe the task they had to perform.

In order to get the students familiarized with the concept of shape grammar and their performance, the Interactive Generation Module of BH-ShaDe allows an interactive execution of two fixed sets of shape grammars. The first one is a toy example that only adds squares. The second one is a small shape grammar (Figure 5.18) that can generate very simple housing units (Figure 5.19).

Once the students had learnt the concept of shape grammars, they proceeded to execute the automatic generation module. First, each student executed the grammar in Figure 5.11 with the learned policies to automatically generate a basic housing unit scheme. The students were then divided in groups of three members. Each group had thus 3 schemes. This procedure could have continued so students had a large enough collection of basic houses to use in their projects. However, the generation of each scheme consumed some time that was not useful in our overall goal of generating starting points that supported students. Therefore, we decided to use the automatic generation module of BH-ShaDe to automatically generate a further 24 schemes for each group, resulting in a total of 27 schemes per group. To promote discussion and reflection among the students, these 3-member groups were joined in larger groups of nine students. Each one of these larger groups had then 81 schemes that they had to classify according to the following criteria: A (optimal), B (adequate), C (some modifications needed), D (problematic) and E (absurd). Once the schemes had been classified, they had to select the most suitable for different kinds of groupings: row-houses, apartment blocks, galleries, single-family houses. Based on these selections

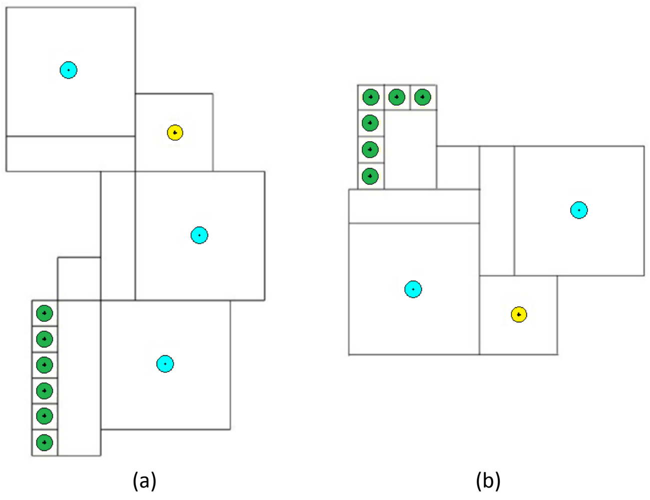


Figure 5.19: Housing units schemes generated with the shape grammar in Figure 5.18

they had to develop a complete architectural project, proposing single-family housing solutions, both in 3D and 2D. The students presented their final projects in the classroom, and the teachers commented and evaluated them.

In this section we will present the results obtained in the experiment. They obtained results have been organized into different subsections: student's survey, teacher's opinion and student's final projects.

5.2.2.6.1 Student Survey. The survey was designed as 11 Likert items, relative to four different topics. The students had to evaluate their degree of agreement with each sentence (from 1 -low- to 6 -high-). We have used the methodology in Jamieson (2004) to analyse nominal Likert items: mode, median, inter-quartile range and nominal levels of disagree (degree of agreement of 1, 2 and 3) vs. agree (degree of agreement of 4, 5 and 6). Additionally, the survey included two free-text items where the students could identify the strong and weak points of the software. We have analysed these free-text items according to the constant comparative method (Corbin & Strauss, 2008), a methodology based on grounded theory (Glaser & Strauss, 1967). In the first step, each student's response is decomposed in the ideas it expresses (answers). Then these answers are divided into categories. In the phenomenological reduction phase, the categories are grouped by subject (themes). Finally, in the triangulation phase, examples of supporting quotes are provided.

As aforementioned, 78 students answered the survey. Results are summarized in Table 5.8, while Table 5.9 and 5.10 summarize the results of the two free text items (using the constant comparative method).

	1	**2**	**3**	**4**	**5**	**6**	MEDIAN	MODE	(Q_1, Q_3)	Don't agree	Agree
About the software tool...											
1. I quickly learned how to use the tool	0	3	4	3	**41**	27	5	5	(5,6)	8, 97%	91, 03%
2. It was easy for me to use the tool	0	0	2	10	32	**34**	5	6	(5,6)	2, 56%	97, 44%
3. The user interface is intuitive	1	3	12	**30**	20	12	4	4	(4,5)	20, 51%	79, 49%
4. The tool worked quick enough	1	4	17	**24**	22	10	4	4	(3,5)	28, 21%	71, 79%
5. The tutorial was easy to follow and useful	1	3	8	11	**36**	19	5	5	(4,5)	15, 38%	84, 62%
About the schemas proposed by the tool...											
6. They can be helpful in the design process	4	12	17	**27**	14	4	4	4	(3,4)	42, 31%	57, 69%
7. They can provide good starting points	2	9	8	25	**29**	5	4	5	(4,5)	24, 36%	75, 64%
8. The schemas were interesting	3	7	19	**24**	21	4	4	4	(3,5)	37, 18%	62, 82%
9. The diversity of the schemas generated was sufficient	5	17	13	**18**	17	8	4	4	(2,5)	44, 87%	55, 13%
10.The schemas were innovative	3	18	**25**	17	12	3	3	3	(2,4)	58, 97%	41, 03%
11.The schemas were reasonable from an architectural point of view	6	19	**28**	19	6	0	3	3	(2,4)	67, 95%	32, 05%
Global Evaluation											
12.It was easy for me to create a dwelling in which I would like to live	5	15	17	**24**	15	2	4	4	(3,4)	47, 44%	52, 56%
13.Without the tool, this practice would have been more difficult	6	17	17	14	**18**	6	3	5	(2,5)	51, 28%	48, 72%
14.I would like to know more about this kind of tools	3	3	24	**29**	8	11	4	4	(3,4)	38, 46%	61, 54%
15.I would like to be able to define my own shape grammars	0	19	**21**	19	11	8	3	3	(3,4)	51, 28%	48, 72%
16.I would like to use this tool in the future	4	17	**22**	21	9	5	3	3	(2,4)	55, 13%	44, 87%
17.I would recommend the use of this tool to my classmates	6	16	**24**	16	10	6	3	3	(2,4)	58, 97%	41, 03%
About this practice											
18.All in one, it was interesting	3	5	7	27	**29**	7	4	5	(4,5)	19, 23%	80, 77%
19.I think that the methodology used was suitable	1	3	15	**26**	25	8	4	4	(4,5)	24, 36%	75, 64%
20.It was rewarding to work in groups	1	3	12	13	**26**	23	5	5	(4,6)	20, 51%	79, 49%

Table 5.8: Results of Likert items

	CATEGORIES			**THEMES**	**EXAMPLES OF SUPPORTING QUOTES**
	Name	**№answers**	**% answers**		
135 different answers	Diversity	26	19, 26%	ASPECTS RELATIVE TO QUALITY OF SOLUTIONS (25, 18%)	*"Great variety of alternative solutions"* *"What I liked most about this practice is that I could develop a feasible project"* *"The tool provides a great variety of cells, some of them present little annex spaces that could be grouped generating unexpected solutions"*
	Validity	6	4, 44%		
	Versatility for groupings	2	1, 48%		
	Usability	3	2, 22%	ASPECTS RELATIVE TO THE SOFTWARE TOOL (11, 85%)	*"The tool was easy to use"* *"The tool generated the schemas quickly and I could take good advantage of some of them"* *"Being able to use new computational methods for architectural design based on randomness"*
	Efficiency	10	7, 41%		
	Possibility of using software tools in the design process	3	2, 22%		
	Possibility of working in groups	10	7, 41%	TEAMWORK (14, 82%)	*"Working in groups and new relationships with other students"* *"Debates in the group about what is desirable or not in architectural design"*
	Processes of selection/reflection carried out in the working groups	10	7, 41%		
	Randomness	9	6, 67%	CREATIVITY (48, 15%)	*"The tool provides a degree of randomness that would be otherwise difficult to include in a project"* *"Little annex spaces that appear accidentally can be used to generate diverse groupings"* *"Being able to have an starting point instead of a blank page"* *"The tool generates schemas that you would not think of"*
	Happy accidents or bugs	3	2, 22%		
	Provides starting points	41	30, 37%		
	Overcoming preconceived solutions	12	8, 89%		

Table 5.9: Categories, themes and supporting quotes for positive aspects

	CATEGORIES			THEMES	EXAMPLES OF SUPPORTING QUOTES
	Name	№answers	% answers		
102 different answers	Software capabilities for edition	9	8, 82%	ASPECTS RELATIVE TO THE SOFTWARE TOOL (12, 74%)	*"The user should be able to modify some aspects of the schemas generated"* *"A more intuitive interface"*
	Usability (interface)	4	3, 92%		
	Shape grammar should include additional criteria	20	19, 61%	ASPECTS RELATIVE TO SHAPE GRAMMARS (21, 56%)	*"Shape grammars should include additional architectonic criteria"* *"To fully exploit the tool, the user should be able to define his/her own shape grammars"*
	Possibility to include user-defined shape grammars	2	1, 96%		
	Overlapping of non-specialized spaces	11	10, 78%	ASPECTS RELATIVE TO DISTRIBUTION OF SPACES (17, 64%)	*"Overlapping of non-specialized spaces should be avoided"* *"Some kitchen modules are inaccessible"* *"The tool should not generate residual spaces"* *"The location of the entrance door near to the kitchen constraints the variety of the solutions"* *"Better distribution of spaces"* *"Wet zones should be contiguous"*
	Bad distribution of kitchen furniture	3	2, 94%		
	Residual spaces	10	9, 8%		
	Better location of doors	12	11, 76%		
	Better distribution of spaces	3	2, 94%		
	Better placement of wet zones	10	9, 80%		
	Excess of randomness	9	8, 82%	ASPECTS RELATIVE TO THE SOLUTIONS (48, 03%)	*"The randomness of the tool should be controlled"* *"Greater variety of schemas generated"* *"Other parameters should be considered (environment, social aspects, etc.) "*
	Poor variety of solutions	7	6, 86%		
	Additional criteria should be considered (not only architectonic)	2	1, 96%		

Table 5.10: Categories, themes and supporting quotes for aspects to be improved

Now we will use the presented results to analyse to what extent the starting points provided by BH-Shade have been useful for the students in the early stages of their design projects. With respect to this question, probably the more useful Likert items are items number 6, 7 y 8. In particular, item number 7 is very relevant, and we can see that 75,64% of the students agree that "The schemes provided by the tool can provide good starting points". In addition, the students seem to agree that "The schemes were interesting" (62,82%), and that "The schemes can be helpful in the design process" (57,69%). This same conclusion can be reached from the analysis of the positive aspects of the first free-text item. Indeed, the theme "Starting Points" spontaneously emerged from student answers, being mentioned by 48,15% of the students, and specifically 30,37% of them mentioned that "The tool provides good starting points". In this theme, other aspect mentioned by 8,89% of the students was the possibility of overcoming preconceived solutions. Teacher's feedback also seems to support this conclusion, because they said that "However, the most interesting projects have emerged from accidental elements like annex spaces or errors". In fact, and according to their experience, "the designs of the clusters of schemes obtained using traditional methods are usually more rigid and less creative that the ones generated with the tool". Continuing with the positive aspects of the tool, the next more frequently mentioned theme in the survey was "Aspects relative to the quality of solutions" (25,28%), and in particular, the category “Diversity” (19,26%). As for the teachers, they declared that “the program generates such high variety of schemes that accidents occurred randomly, giving birth to what at first sight could be considered as undesirable forms, but finally generate the most interesting projects”. With respect to the tool, the students emphasized (7,41%) its “Efficiency” (also the teachers said “The tool expedited the design”). All in one we think that, according to both the teachers and the students, the stronger point of BH-ShaDe is its capability to generate an unlimited number of diverse, feasible, random and suggestive starting points for novice designers. With respect to aspects to be improved (and focusing our discussion in the quality of the starting points provided), the most frequently mentioned theme was “Aspects relative to distribution of spaces” (48,03%), and specifically the categories “Better location of doors” (11,76%), “Overlapping of non-specialized spaces” (10,78%), or “Better placement of wet zones” (9,8%). Next more frequently mentioned theme was “Aspects relative to Shape grammars” (21,56%), in particular, the category “Shape grammars should include additional criteria” (19,61%).

5.2.2.6.2 Teacher’s opinion. Three architecture teachers participated in the experiment. One of them was a member of our research team, while the other two did not have previous knowledge of shape grammars or about the tool. We developed a small survey for these two teachers, which had four open questions. For space reasons we do not include their complete answer, but a brief summary of the more relevant information. It seems that what the teachers liked most of this experience was the possibility to use this kind of tool and learn about shape grammars, together with the interdisciplinary work carried out by the research team and the fact that the tool can provide an unlimited number of schemes, and therefore expedite the design process. Overall, they said that the students had done a great job, and that the more interesting projects emerged from the accidental elements, like the little annex spaces generated in some of the automatic solutions presented by the software tool. More concretely, they said that “The tool expedited the design process. The most interesting projects have emerged from accidental elements, like annex spaces or errors. Initially, they seemed not to have any practical use, buy finally they have served to encourage the clustering of the dwellings, and have provided support so the students could freely use their imagination. We do believe the tool has accelerated this kind of discoveries". They also pointed out that the use of tool has provided an excellent exercise of analysis and reflection. The students have learned in a practical way that their preconceived ideas are not always the best ones.“Having 81 housing plan floors automatically generated by the tool, so they could be discussed and selected by the groups of students, has been an excellent exercise about analysis/reflection, which is

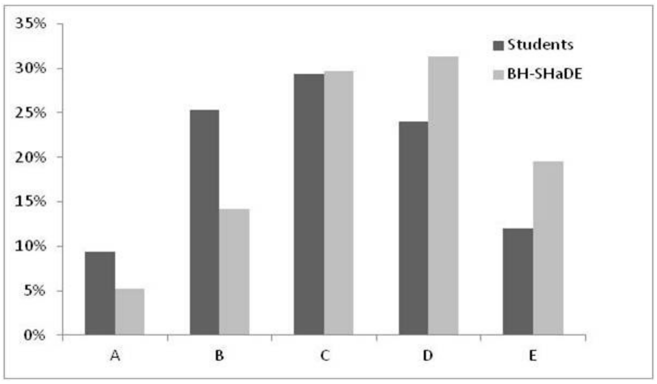


Figure 5.20: Distribution of the total percentage of schemes of each type (A to E)

usually easier to carry out in other people's work than in our own designs. At the same time, the program generates such a wide variety of schemes that accidents occurred randomly, giving birth to what at first sight could be considered as undesirable forms, which finally generated the most interesting projects. Students have therefore learned in a practical way that their preconceived ideas are not always the more appropriate solutions". As for possible improvements, they mentioned that it would be useful to include more architectural criteria in the shape grammars defined. “In relation to the tool and its use in this particular activity, we think that it would be desirable to extend the number of variables to be taken into account in the shape grammar".

5.2.2.6.3 Students' Final Projects. In the last session of the experiment, groups of students presented their final projects. Each group presented A1 sheets with the 81 schemes, evaluated from A (no changes needed) to E (the scheme is absurd). In total, there were 9 groups of 81 schemes. The distribution of the percentage of schemes classified according to the different grades (both among those produced by the students and among those generated automatically by BH-ShaDe) is shown in Figure 5.20.

In their presentations, the students stated that they had held very productive discussions to agree about criteria to classify schemes (recall that teachers also thought that this discussion/reflection process had been very productive). To this respect, some groups had established more demanding criteria than others. The teachers pointed out that some of them had discarded useful schemes for irrelevant reasons, such as a poor positioning of some elements (door, kitchen/bath furniture) or the superposition of non-specialized spaces. The high number of schemes classified with D and E can be explained by the fact that the students considered criteria (circulation, ventilation, light distribution or grouping of wet zones) not accounted in the guideline used by the tool.

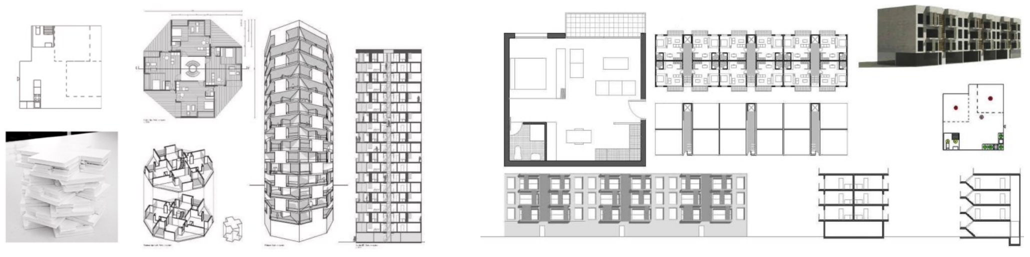


Figure 5.21: Examples of student's project: (a) Octagonal tower and (b) Gallery

From the schemes classified with A or B, each group selected four or five as the basis to create more complex structures. They explored different kinds of groupings: single-family houses, apartment blocks, galleries, etc.

There were many interesting projects, for illustration purposes we will show two of them (selected by the teachers as illustrative for different criteria) in Figure 5.21. The teachers selected the first project as representative of those than emerged from starting points that contained accidental elements (and finally gave birth to creative solutions). This group used one of the schemes generated by the computer, even when it had two small corridors next to non-specialized spaces. The students decided to use these two small corridors as terraces to generate an octahedron tower. The second project was chosen because it illustrated a nearly feasible solution (in teacher?s words, it seemed nearly pre-conceived). The students selected this scheme among those generated by the computer because it has a good distribution of the so-called wet zones (kitchen and bathroom). In this way, once the schemes are grouped, the wet zones occupy the central part of the building.

Overall, it seems that both the teachers and the students think that the inclusion of additional architectural criteria (circulation, ventilation, illumination, grouping of wet zones...) could improve the quality of the solutions provided. Finally, and in relation with the overall quality of the starting points, Figure 5.20 shows that the students gave higher scores to their own designs than to those generated automatically by the tool.

5.2.3 Application Case 2: Generation of Energy-Efficient Housing Unit Designs

In this section we extend the previous application case by adding design requirements related to energy consumption. Now we seek to generate schemes of habitable and energy efficient housing units (Ruiz-Montiel et al., 2015; Hidalgo et al., 2015). Some previous works also rely on shape grammars to produce energy efficient design solutions: Caldas (2008) uses generative rules along with evolutionary algorithms to generate energy-efficient architecture solutions, and Granadeiro et al. (2013) converts a shape grammar into a parametric design system able to yield envelope shapes that can be evaluated by means of an integrated energy simulation software.

Certainly, research on energy efficient buildings is currently on demand, starting with designs that consume the least amount of natural resources and produce a minimum of residual materials along their life cycle.

Behling distinguishes between *active systems*, *passive systems* and the *architectonic form* itself as central elements in the thermodynamics of a building (Ábalos, 2008, 2012). The first ones control the energy interchange in the building by means of mechanic devices with a high energy consumption (such as ventilation, heating and air conditioning systems). On the contrary, passive systems are elements added a posteriori, in top of the architectonic form, that do not need from a mechanic action (like covers, projections, fronts or sunshades). Finally, the architectonic form favours an adequate energy interchange in a passive way, but it is projected a priori.

The traditional methodology consists on emphasizing the technology of active energy-saving systems. This technology is usually superimposed to passive systems, that are consequence of a choice of architectural forms where sustainability has been disregarded. Behling (2012) proposes to invert the relative importance of these systems, so active ones have a residual importance and thus the process of selecting shapes guarantees the sustainability of the project.

5.2.3.1 Energy Efficient Housing Unit Design

In order to generate energy-efficient housing units we have considered a set of good design practices that can be found in several guides and research works (Enerbuilding, 2008; Passivhaus, 2011; Pacheco et al., 2012). These criteria, defined in the climatic context of Spain, can be classified into three categories:

1. Contour-related criteria: according to Pacheco et al. (2012), passive design techniques usually imply the maximization of both the compactness of the contour and the portion of the façade oriented to the south

2. Inner spaces criteria: according to the Passivhaus Standard (Passivhaus, 2011), the orientation of the different inner spaces of the house can affect to its energetic efficiency. For example, the living room might works better oriented to the south due to the adequacy of this orientation throughout the whole day, during both summer and winter; and the bedroom can benefit from an east orientation, as it allows to get advantage of the natural light of the sunrise in order to wake up

Global requirements
R1: Total area must be of at least 48 m^2
Habitability requirements
R2: There must exist two non-specialized spaces, a kitchen, a bathroom, a distributor space and an access to the housing unit
R3: Spaces must not overlap
R4: Distance requirements extracted from Figure 5.10, where two spaces are here considered to be adjoining if the distance between them is not greater than 4 meters
Energy efficiency requirements
R5: The contour must be compact
R6: The proportion of the façade oriented to the south (south-oriented perimeter/-total perimeter) has to be maximized
R7: Non-specialized spaces must be south or east-oriented
R8: The total glazed surface must represent a minimum percentage of the total façade. Such percentage depends on the compactness factor of the contour: if the perimeter is greater than 40 meters, and hence we have a low compactness factor, the percentage will be 25%. Otherwise, this percentage will be 30%
R9: Every space must have an associated window
R10: The glazed surface oriented to the south must represent at least the 40% of the total south façade
R11: The glazed surface oriented to the south must represent at least the 40% of the total glazed surface
R12: The glazed surface oriented to the west must represent at most the 20% of the total glazed surface

Table 5.11: Requirement set for a energy efficient single-family basic house

3. Window-related criteria: according to the Enerbuilding guide (Enerbuilding, 2008), the amount of glazed surface oriented to the south must represent a significant percentage of the total façade

Regarding the design criteria related to the habitability of the housing units, here we have considered a similar set of requirements to those considered in the previous example, but simplified. The whole set of considered criteria are gathered in Table 5.11. Here we consider some additional requirements that are not mentioned in the guidelines, but have a considerable architectonic relevance: non-specialized spaces must have an associated window, and the amount of glazed surface oriented to the west has been limited.

5.2.3.2 Naive Grammars and Phases of Generation

The generation process for producing energy-efficient, habitable housing units has been divided into four phases:

1. Generation of the contour
2. Placement of the different spaces in the housing unit
3. Placement of the windows

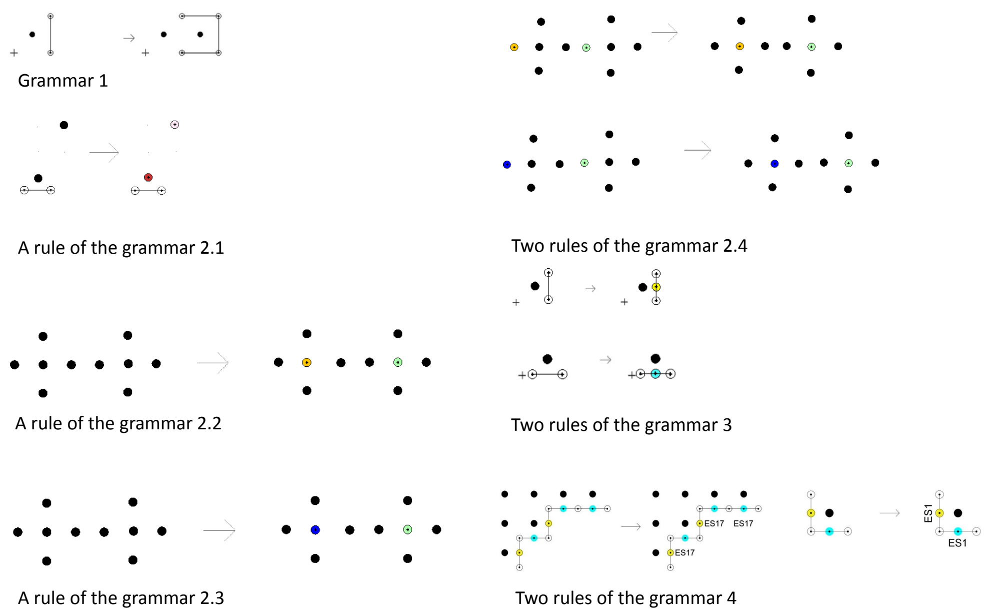

Grammar 1
A rule of the grammar 2.1
A rule of the grammar 2.2
A rule of the grammar 2.3
Two rules of the grammar 2.4
Two rules of the grammar 3
ES17
ES17
ES17
ES1
ES1
Two rules of the grammar 4

Figure 5.22: Some rules of the involved shape grammars

4. Grouping of contiguous windows

As in the previous application case, each phase is associated to a particular shape grammar and a particular set of design criteria.

In Figure 5.22 we illustrate the some rules of shape grammars used in this case. The complete set of rules can be found in the technical report that describes in detail this application case (Hidalgo et al., 2015). For each grammar, a termination condition is established, in order to decide when the application of the rules must stop. In the following we detail the necessary steps for synthesizing a housing unit scheme:

1. The production of a scheme starts with an axiom defined by a rectangle of 6×4 meters. Starting from this shape, the rule of the grammar 1 is applied until reaching a contour of $48m^2$

2. The second phase consists of four grammars: the first three (grammars 2.1, 2.2 and 2.3) place the necessary spaces of the house, by adding different labels. Each one of these grammars is applied once, by picking up one of their rules, over the shape synthesized by grammar 1. Grammar 2.4 is used over the shape produced by the grammar 2.3, and contains rules that move the placed spaces. These rules

are applied until the design criteria related to the second phase are met or until a limit of 50 rule applications

3. The third phase places the windows applying the grammar 3 over the shape synthesized by the grammar 2.4. Rules of grammar 3 are applied until the total glazed surface is greater or equal to a given percentage of the total façade. Such percentage depends on the compactness factor of the contour. We work with three types of windows: south-oriented, north-oriented and east/west-oriented, each one with different protection elements

4. The fourth phase regroups the windows transforming them into large windows of porches, applying grammar 4 randomly over the shape produced by grammar 3

In Figure 5.23 we show two schemes generated by means of this sequence of steps, without using any rule heuristic that tells which rules are better to apply. That is, the rule that is applied at each step is picked up randomly. As we will discuss in Section 5.2.3.4, these schemes are not actually admissible, highlighting the need for rule application heuristics in order to comply with the considered housing unit requirements.

5.2.3.3 Learning Processes for Naive Shape Rules

In order to get good solutions we need to learn policies for phases 1, 2 and 3. Concretely, the grammars that will benefit from this learned policies will be grammars 1, 2.4 and 3. These policies will decide which rule and transformations are better to apply at each step, with the goal of synthesizing valid designs regarding the design criteria gathered in Table 5.12. The fourth phase does not need such a policy, because we consider that every generated group of windows is admissible.

In Table 5.12 we describe in detail the whole set of considered design criteria (related to both energetic efficiency and habitability) that are sought in this example, classified according to the grammar for which they will be considered.

The rewards for the learning processes of each grammar are defined in terms of the requirements gathered in Table 5.12. They are described in Table 5.13.

The features for every grammar are detailed in Table 5.14. Once the policies are learned, they will be used to guide the generative processes by choosing between different combinations of features. For example, for grammar 2.4, the features are based on the distance between pairs of spaces. What a policy for this phase can learn is which ranges of distances for each pair are more suitable in order to attain the most habitability and efficiency requirements.

Requirements Grammar 1
R1-1: The total surface of the housing unit must be at least of 48 m^2
R1-2: The compactness factor of the contour (area/perimeter^2) has to be maximized
R1-3: The proportion of the façade oriented to the south (south-oriented perimeter/-total perimeter) has to be maximized
Requirements Grammar 2.4
R2-1: There must exist two non-specialized spaces, a kitchen, a bathroom, a distributor space and an access to the housing unit
R2-2: Spaces must not overlap
R2-3: Distance requirements extracted from Figure 5.10, where two spaces are here considered to be adjoining if the distance between them is not greater than 4 meters
R2-4: Non-specialized spaces must be south or east-oriented
Requirements Grammar 3
R3-1: The total glazed surface must represent a minimum percentage of the total façade. Such percentage depends on the compactness factor of the contour: if the perimeter is greater than 40 meters, and hence we have a low compactness factor, the percentage will be 25%. Otherwise, this percentage will be 30%
R3-2: Every space must have an associated window
R3-3: The glazed surface oriented to the south must represent at least the 40% of the total south façade
R3-4: The glazed surface oriented to the south must represent at least the 40% of the total glazed surface
R3-5: The glazed surface oriented to the west must represent at most the 20% of the total glazes surgace

Table 5.12: Requirement set for a single-family basic house, regarding habitability conditions and energetic efficiency

Grammar	**Reward**
1	The reward is a weighted sum of requirements R1-2 and R1-3, normalized between 0 and 1. In case this sum is greater than 0.8, the reward is truncated to 1, in order to favour variability
2.4	Each individual requirement extracted from R2-2, R2-3 and R2-4 is evaluated to 1 or -1 depending on whether it is fulfilled or not, respectively. The reward is a weighted sum of these evaluations
3	Each individual requirement extracted from R3-2, R3-3, R3-4 and R3-5 is evaluated to 1 or -1 depending on whether it is fulfilled or not, respectively. The reward is a weighted sum of these evaluations

Table 5.13: Rewards for every learning process

Features for Grammar 1 (normalized between 0 and 1)
FG1-1: Number of modules of $1m^2$ with one neighbour
FG1-2: Number of modules of $1m^2$ with two neighbour
FG1-3: Number of modules of $1m^2$ with three neighbour
FG1-4: Number of modules of $1m^2$ with four neighbour
FG1-5: Length of the south projection of the contour
Features for Grammar 2.4
In the learning process associated to this grammar, we work with 48 binary features that characterize the final shapes by means of the distances between pairs of spaces. For example, there are three features for the kitchen-bathroom pair, where the first feature is evaluated to 1 if the distance between them is not greater than 2 meters, and to 0 otherwise. Similarly, the second and third features are evaluated to 1 when the distance is comprised between 2 and 4 meters, and to 0 when is greater than 4 meters, respectively
Features for Grammar 3
In the learning process associated to this grammar, we work with 12 binary features that characterize the final shapes by means of the percentages of glazed surface on each orientation. For example, there are three features for the south-oriented windows, where the first feature is evaluated to 1 if the number of windows located in this orientation represents less than the 35% of this façade, and to 0 otherwise. Similarly, the second and third features are evaluated to 1 when the percentage is comprised between 35% and 70% and when is greater than 70%, respectively

Table 5.14: Rewards and features for the learning processes

In the following section we show the obtained results.

5.2.3.4 Results

In this section we show some obtained results and evaluate them according to their energy consumption. As we explained in previous sections, the rewards for this application case have been chosen according to a set of good design practices aimed at obtaining energy-efficient buildings. However, in order to test that these good practices really lead to energy-efficient designs when used as rewards inside a reinforcement learning process, we perform an additional final step, apart from the generation process, that consists on evaluating the energy consumption of the buildings that correspond to the generated schemes. For this sake, we suppose that the materials and structures for the walls of the housing units are determined by the constructive system Baupanel®(Baupanel, 2015). As for the windows, the chosen materials are aluminium for their structure and double glazing a with low-factor kind of glass. The evaluation of the energetic demand of the housing units has been performed by means of the official tool LIDER-CALENER, developed by the Spanish Government. According to the Spanish normative, the energetic consumptions must not be greater than $15kw/m^2$ per year.

In this section we also measure the generation times for the schemes produced by the naive rules by themselves (that is, without benefiting from the learned heuristics)

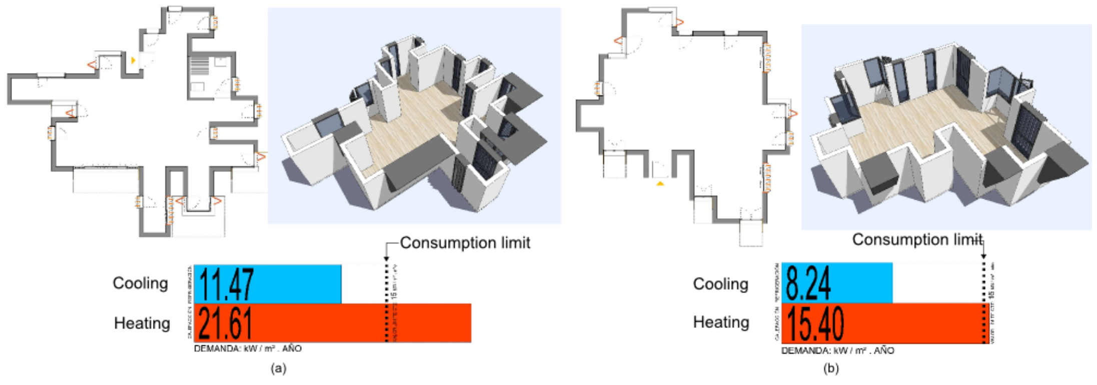


Figure 5.23: Two schemes generated without extra control mechanisms, along with their energetic evaluations

and by the naive rules with policies. All tests of this application case were run on an Intel Core i7 860 @2.80 GHz processor with 8GB RAM and Windows 7 (64 bits).

5.2.3.4.1 Naive grammars without policies. A number of schemes was generated with the set of naive rules illustrated in Figure 5.22, without using the learned policies. Namely, the system was used to generate 100 schemes (two of them are shown in Figure 5.23). For the final presentation of the schemes, labels and segments are substituted with architectonic elements (walls, doors, windows, etcetera), and two and three-dimensional detailed views are provided, as well as the evaluation of energetic demands on cooling and heating.

Computation was even faster than in the previous application case, as less rule applications are needed for generating a final scheme. The minimum generation time for a scheme was 2,38 s and the maximum 2.86s. The mean time was 2.63 s.

The schemes are varied, but they violate several requirements of the housing program described in Section 5.2.3.1. For example, those in Figure 5.23 violate the following requirements of Table 5.11:

- R3. Spaces overlap
- R4. Distance requirements are not fulfilled
- R5. The contour is not compact
- R6. The proportion of the façade oriented to the south has not been maximized
- R10. The glazed surface oriented to the south does not represent at least the 40% of the total south façade

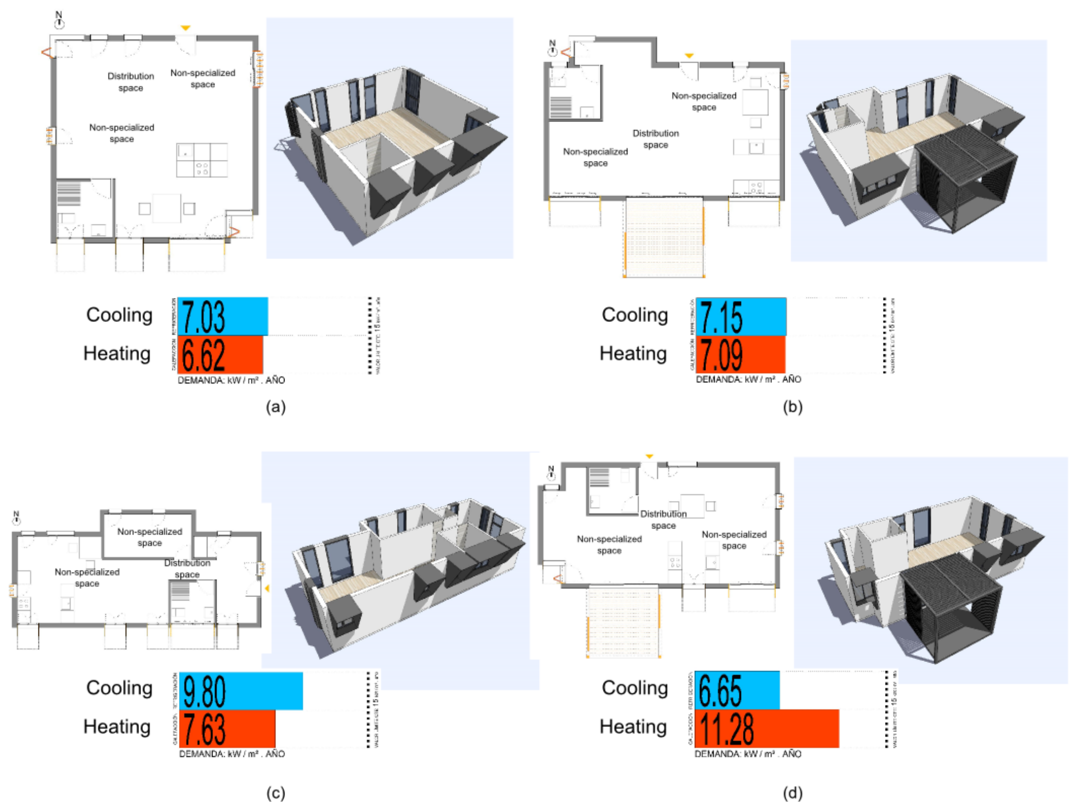


Figure 5.24: Four schemes generated using the learned policies, along with their energetic evaluations

- R11: The glazed surface oriented to the south does not represent at least the 40% of the total glazed surface

Regarding the energetic efficiency, these designs do not comply with the limits imposed by the Spanish government, particularly in the case of the heating consumption.

These results could be expected given the use of naive rules without further guidance, as they do not enforce the whole set of the involved requirements, and thus arbitrary execution is not likely to lead to feasible designs. These deficiencies justify the need for an extra control mechanism that determines which rule/s is/are better to apply at each step.

5.2.3.4.2 Naive grammars with policies. A hundred designs were produced by means of generation processes guided by policies when necessary. In Figure 5.24 we can see four of them. As we can see, the energetic evaluation of these housing units complies with the limits established by the Spanish normative.

Computation time was fast. The minimum generation time for a scheme was 7.23 s and the maximum 8.5s. The mean time was 7.95s. As in the previous application

Grammar	Reward Improvement
1	2.47
2.4	1.09
3	1.39

Table 5.15: Reward improvements gained thanks to the learned policies for every grammar

case, the time increase with respect to the random execution of the naive rules is due to the calculus of the features that has to be carried out in this policy driven approach.

5.2.3.5 Discussion

In this section we discuss the profit obtained thanks to the use of the learned policies. As in the previous application case, here we have also measured the average reward improvement that is gained thanks to the learned policies. For each grammar i that can be combined with a policy, we denote $\overline{R}_i$ as the average accumulated reward obtained when the grammar is applied randomly, and $\overline{R}_i^*$ as the average accumulated reward obtained when the learned policy is used to guide the grammar derivation. The reward improvement for the grammar i is then represented by the expression $\overline{R}_i^*/\overline{R}_i$. The reward improvements for every grammar in this application case are gathered in Table 5.15.

As we can appreciate in the light of tables 5.7 and 5.15, the gains obtained in this application case are significantly more modest than in the previous one. This fact is most evident in the case of grammar 2.4, where the improvement is almost non-existent.

5.3 Pareto MORL for Design

In this section we describe a new methodology that also involves naive shape grammars as a generative system and reinforcement learning as a method for discovering a rule application heuristic. However, as opposed to the previously described system, this methodology follows the *axiomatic* approach described by Roijers et al. (2013), so we start from the premise that the output of the algorithm is a set of policies that yield a Pareto front of optimal accumulated rewards.

Many proposals for multiple policy MORL follow a sequential scheme and compute only a policy at once, corresponding to a certain combination of the objetives (sometimes the combination of objectives is nonlinear (Shelton, 2001; Vamplew et al., 2011; Van Moffaert et al., 2013b)). On the contrary, few algorithms aim to learn the set of policies in a simultaneous manner. That means, for the linearizing approach, that weights are considered simultaneously and implicitly (Hiraoka et al., 2008; Mukai et al., 2012).

Two recent proposals compute the Pareto set simultaneously and without assuming a linearizing procedure or any other combination function (Handa, 2009; Van Moffaert & Nowé, 2014), although they are not applied to design processes.

The work of Handa (2009) employs statistical learning in order to extract a set of policies from a set of generated episodes. At each iteration of the process, the learned set (or a set of uniform policies in the first iteration) is used to generate a number of episodes, and the algorithm selects the best ones according to a Pareto-related measure. The selected episodes are clustered depending on their associated accumulated rewards, and each cluster is employed to extract a new policy by means of statistical learning. The newly extracted policies are used in the next iteration, and the process is repeated until a stopping criterion is reached.

The algorithm developed by Van Moffaert & Nowé (2014) also learns several policies in a single run. At each step of the interaction the algorithm updates the expected immediate reward for the visited state and action pair (s, a). The immediate rewards learned so far and the learned values for the reached state s' are employed to calculate a set of Q values that can be obtained from (s, a). To compute these Q values, the algorithm does not rely on the set of Q values learned previously for (s, a). Instead, the algorithm learns the model on the go, that is, it stores the obtained immediate rewards and approximates the probabilities of going from s to each s' when performing a.

The algorithm here presented also searches simultaneously for all the Pareto-optimal deterministic solutions. Following the philosophy of Q-learning Watkins (1989), the algorithm does not try to learn the model but just stores the Q-values. However, in the MORL case it is also necessary to maintain a set of pointers that justify each value.

In contrast with Handa (2009), the algorithm does not employ statistical learning but temporal differences. It also bootstraps, that is, it relies on other value estimations. In contrast with Van Moffaert & Nowé (2014), the algorithm strictly follows the philosophy of Q-learning and temporal differences: Q-values are updated following each interaction with the environment and taking into account the received rewards and the previous Q-values.

In the following we describe the new algorithm, called PQ-learning (Ruiz-Montiel, 2013; Ruiz-Montiel et al., 2013b). Then we apply the new technique to the geometric problem of maximizing the compactness factor and the perimeter of rectilinear polygons (see Section 5.2.1.1). As PQ-learning is a tabular technique, now the methodology followed to use PQ-learning for a design problem is straightforward, as we only have to define the rewards. It remains future work to extend PQ-learning so it can be used along with generalization techniques.

5.3.1 PQ-learning: a New Multi-Policy MORL Approach

This section describes PQ-learning, an extension of the scalar Q-learning algorithm that learns the set of all nondominated policies for a MOMDP. Since we aim to learn the set of all nondominated policies at once, Q-learning is a promising starting point, since the policy used to interact with the environment is not the same that is learned.

5.3.1.1 Characterization of Optimal Policies

Our problem is to find the policies that maximize the expected vector return of a MOMDP (see Section 2.3.4 for a formalization for the concept of MOMDP and the associated expected vector return). We will consider the *dominance relation* $\succ$ or *Pareto order* in $\mathbb{R}^n$. For every pair of vectors $\vec{v}, \vec{w} \in \mathbb{R}^n$, $\vec{v} \succ \vec{w}$ iff there exists a dimension i such that $v_i > w_i$ and there is no dimension j such that $v_j < w_j$. We write $\vec{v} \succeq \vec{w}$ when $\vec{v} \succ \vec{w}$ or $\vec{v} = \vec{w}$. Given a set $X \subset \mathbb{R}^n$, the *Pareto front* of X, $ND(X)$, is the set of nondominated vectors in X, that is, $ND(X) = \{\vec{x} \in X \mid \nexists \vec{y} \in X \;\; \vec{y} \succ \vec{x}\}$.

The *vector value* of a state s under a policy π, denoted $\vec{v}^{\,\pi}(s)$, is the expected vector return when starting from s and following π. We call $\vec{v}^{\,\pi}(s)$ the *state-vector function* for policy π. The vector value of taking action a in state s under a policy π, denoted $\vec{q}^{\,\pi}(s,a)$ is the expected vector return when starting from s, taking action a, and thereafter following π. We call $\vec{q}^{\,\pi}(s,a)$ the *action-vector function* for policy π.

We will say that a policy π dominates or equals a policy π' iff forall $s \in S$, $\vec{v}^{\,\pi}(s) \succeq \vec{v}^{\,\pi'}(s)$. A policy is non-dominated iff it is not dominated by any other possible policy. We denote a non-dominated policy as π^*, and its non-dominated action-vector function as $\vec{v}^{\,\pi*}(s)$.

We denote the set of all action-vectors of taking action a in state s under any non-dominated policy π^* as $\mathbb{Q}^*(s,a)$.

5.3.1.2 Updating Expression

Just like Q-learning, PQ-learning learns action-vector values through repeated interaction with the environment. However, there are important differences that arise from the fact that rewards are vectors, and that the algorithm does not learn a single policy, but a set of policies at the same time.

The values learned in PQ-learning are sets $\mathbb{Q}(s,a)$, which are used to estimate the optimal $\mathbb{Q}^*(s,a)$ sets. A decision-making agent interacts with the environment through a sequence of steps. In the n^{th} step, the agent observes its current state s_n, selects and performs an action a_n, observes the following state s', receives an immediate reward $\vec{r}_n$ and adjusts the set $\mathbb{Q}(s_n, a_n)$ using a learning factor α_n. Therefore, the essence of PQ-learning is described by an updating expression, in many senses analogous to the one of Q-learning.

The basic idea behind the updating procedure of PQ-learning is that a non-dominated action-vector must result from the combination of non-dominated action-vectors associated to the same policy. Since PQ-learning learns all non-dominated policies at the same time, the $\mathbb{Q}(s, a)$ sets store different vectors estimates. Each estimate arises from the combination of a set of non-dominated state vectors of states s' reached after action a.

The scalar Q-learning algorithm calculates action values $Q(s, a)$ combining the state values $V(s')$ of states s' reached after performing action a in state s. In the case of PQ-learning, we have sets of state-vectors $\mathbb{V}(s')$, defined as the set of nondominated action-vectors for all possible actions in state s'. Each of these state-vectors is associated to a different policy. Each action vector in $\mathbb{Q}(s, a)$ will result from the combination of particular state-vector estimates of reached states, and should always be updated according to them, reflecting thus a particular policy.

Formally, each *vector estimate* in $\mathbb{Q}(s, a)$ will consist of a pair $(\vec{q}, P)$, where $\vec{q}$ is the current value of the vector estimate, and P is a set of indices. Each index $p \in P$ is a pair (s', i) where s' is an identifier of the accessed state and i stands for the i-th vector in $\mathbb{V}(s')$, precisely the one that is used to update $\vec{q}$. Notice that the dimension of $\vec{q}$, given by the number of objectives, is fixed and the same for every state. However, the size of P can be different for different states and grows during the execution of the algorithm from 0 to the number of states reachable from s after performing action a.

We will say that a pair (s', i) is new for a set $\mathbb{Q}(s, a)$ when there is no pair $(\vec{q}, P) \in \mathbb{Q}(s, a)$ such that $(s', i) \in P$, and write $(s', i) \not\sqsubset \mathbb{Q}(s, a)$. Otherwise we will say that (s', i) is not new in $\mathbb{Q}(s, a)$ and write $(s', i) \sqsubset \mathbb{Q}(s, a)$. We will say that a state s' is new for a set $\mathbb{Q}(s, a)$ when there is no index i such that $(s', i) \sqsubset \mathbb{Q}(s, a)$. We will write $s' \not\sqsubset \mathbb{Q}(s, a)$. Otherwise we will say that s' is not new in $\mathbb{Q}(s, a)$ and write $s' \sqsubset \mathbb{Q}(s, a)$. We also define the set $P \setminus s'$ as the result of removing all pairs (s', i) from P, that is, $P \setminus \{(s', i)\}$ (whatever i is).

We will formally define the sets $\mathbb{V}(s)$ in function of the vector estimates (this definition is similar to the one used by White (1982) and Wiering & de Jong (2007)):

$$\mathbb{V}(s) = ND \bigcup_{a \in A} \{\vec{q} \mid (\vec{q}, P) \in \mathbb{Q}(s, a)\} \tag{5.3}$$

We can identify two different kinds of operations in the updating process of $\mathbb{Q}(s, a)$,

1. Creating new vector estimates every time a state s' is reached for the first time after performing action a in state s.

2. Updating $\mathbb{Q}(s,a)$ after an already traversed transition leading to some node s'. This includes updating, creating, and deleting particular vector estimates.

We assume that for all $s \in S$ and for all $a \in A$, $\mathbb{Q}_0(s,a) = \{(\vec{0}, \varnothing)\}$. The updating expression for PQ-learning is,

$$\mathbb{Q}_n(s,a) = \begin{cases} \mathbb{N}_{n-1}(s,a) \cup \mathbb{U}_{n-1}(s,a) \cup \mathbb{E}_{n-1}(s,a) & \text{if } s = s_n \wedge a = a_n \\ \mathbb{Q}_{n-1}(s,a) & \text{otherwise} \end{cases} \tag{5.4}$$

In the following we define the sets $\mathbb{N}, \mathbb{U}$ and $\mathbb{E}$ depending on the kind of operation that is being performed.

- **Case 1**. When a state s' is reached for the first time from state s through action a, every vector estimate in $\mathbb{Q}_{n-1}(s,a)$ is updated with every vector estimate inside $\mathbb{V}_{n-1}(s')$, performing a pairwise update. If $\mathbb{V}_{n-1}(s')$ only has one vector estimate, then $|\mathbb{Q}_n(s,a)| = |\mathbb{Q}_{n-1}(s,a)|$. In general, when s' is reached for the first time, $|\mathbb{Q}_n(s,a)| = m|\mathbb{Q}_{n-1}(s,a)|$, where m is the number of vector estimates inside $\mathbb{V}_{n-1}(s')$. More formally,

$$\begin{aligned} \mathbb{N}_{n-1}(s,a) = \{((1-\alpha_n)\vec{q} + \alpha_n[\vec{r}_n + \gamma \vec{v}_j], P \cup \{(s', j)\} \mid \\ (\vec{q}, P) \in \mathbb{Q}_{n-1}(s,a) \wedge \vec{v}_j \in \mathbb{V}_{n-1}(s') \wedge s' \not\subset \mathbb{Q}_{n-1}(s,a)\} \end{aligned} \tag{5.5}$$

  In this case, $\mathbb{U}, \mathbb{E} = \varnothing$.

- **Case 2**. When the reached state s' was already previously reached from state s through action a, the previous pairwise update defined by the set $\mathbb{N}$ has already been performed in some previous step corresponding to the first time that state s' was reached, thus now $\mathbb{N} = \varnothing$. For correctly updating $\mathbb{Q}(s,a)$, we have to distinguish those vector estimates inside $\mathbb{V}(s')$ that have previously updated some vector of $\mathbb{Q}(s,a)$ from those which not:

  – Set $\mathbb{U}$ deals with the first kind of vectors in $\mathbb{Q}(s,a)$, which are updated with their associated vectors in $\mathbb{V}(s')$. The formula is entirely analogous to the one in the single objective case:

$$\mathbb{U}_{n-1}(s,a) = \{((1-\alpha_n)\vec{q} + \alpha_n[\vec{r}_n + \gamma\vec{v}_j], P) \mid (\vec{q}, P) \in \mathbb{Q}_{n-1}(s,a) \wedge (s',j) \in P \wedge \vec{v}_j \in \mathbb{V}_{n-1}(s')\} \tag{5.6}$$

If any vector inside $\mathbb{V}(s')$ has updated a vector in $\mathbb{Q}(s,a)$, then $\mathbb{U} = \varnothing$.

– Set $\mathbb{E}$ deals with the second kind of vectors, arising when a previously unknown vector appears in $\mathbb{V}(s')$ (we call this an *extra* vector). These vectors have not been used to update any vector of $\mathbb{Q}(s,a)$. The expression $s' \sqsubset \mathbb{Q}_{n-1}(s,a) \wedge (s',j) \not\sqsubset \mathbb{Q}_{n-1}(s,a)$ characterizes this situation: as this is not the first time s' is reached, $s' \sqsubset \mathbb{Q}_{n-1}(s,a)$, but as $\vec{v}_j$ is new, $(s',j) \not\sqsubset \mathbb{Q}_{n-1}(s,a)$. In this case, we cannot make use of vectors estimates inside $\mathbb{Q}_{n-1}(s,a)$ to determine the value of the new vector, because all of them have already been supported by some vector estimate inside previous instances of $\mathbb{V}(s')$, different from the extra vector $\vec{v}_j$. So in this situation we have to set from scratch the value of the vector in the new pair that is being inserted into $\mathbb{Q}_{n-1}(s,a)$. However, we can establish the new set of indices with the available information of the pairs inside $\mathbb{Q}_{n-1}(s,a)$. Given an extra vector $\vec{v}_j$, we will insert a new vector estimate in $\mathbb{Q}_n(s,a)$ for each vector estimate inside $\mathbb{Q}_{n-1}(s,a)$, with the set of indices that arises of removing the pair of the form (s',k) (whatever k is) from the set of indices of the vector estimate, and then adding the pair (s',j):

$$\mathbb{E}_{n-1}(s,a) = \{(\alpha_n[\vec{r}_n + \gamma\vec{v}_j], (P \setminus s') \cup \{(s',j)\}) \mid \vec{v}_j \in \mathbb{V}_{n-1}(s') \ \wedge\ s' \sqsubset \mathbb{Q}_{n-1}(s,a) \ \wedge\ (s',j) \not\sqsubset \mathbb{Q}_{n-1}(s,a) \ \wedge\ \exists\vec{q}\ (\vec{q}, P) \in \mathbb{Q}_{n-1}(s,a)\} \tag{5.7}$$

If there are no new vectors appearing inside $\mathbb{V}(s')$, then $\mathbb{E} = \varnothing$.

When a vector estimate that was present in previous instances of $\mathbb{V}(s')$ (and thus there is an associated vector estimate inside $\mathbb{Q}_{n-1}(s,a)$) has been removed, it is because now it represents a dominated policy. In that situation, the vector estimate inside $\mathbb{Q}_{n-1}(s,a)$ supported by the vector estimate removed from $\mathbb{V}_{n-1}(s')$ is not inside $\mathbb{Q}_n(s,a)$.

Notice that PQ-learning keeps a vector in $\mathbb{Q}(s,a)$ for each possible combination of non-dominated vectors from the state-vector sets of reachable states. However, by application of Bellman's optimality principle, only the subset of non-dominated state vectors of s, $\mathbb{V}(s)$, are needed to support action vectors in other Q sets.

5.3.1.3 Action Selection Mechanism

As Q-learning, PQ-learning is an off-policy technique, meaning that the policy that is followed during learning is not the same as the one learned. Indeed, this fact is even more evident in the multi-objective setting, as we are learning several policies at once, but the agent is only following one path.

This off-policy essence allows the agent to follow any policy during the learning process, as long as every state-action pair (s, a) is visited a potentially infinite number of times. Nevertheless, devising a proper action-selection strategy can lead to a better online performance.

The usual intuition behind an action-selection strategy is that, when exploiting the already acquired knowledge, we choose one of the actions that yield a maximum value. In a scalar setting we only have one maximum value, and if there is a draw, then we choose one of the values randomly and take the action that yields it. The spirit of the mechanism presented here is similar, as we can consider that all the vector estimates inside $\mathbb{V}(s)$ are maximum values (because they are non-dominated). However, the chances of choosing a particular action are not equally distributed: the chances of choosing an action a, being in state s, are proportional to the number of vector estimates of $\mathbb{Q}(s, a)$ that are also inside $\mathbb{V}(s)$.

$$Pr\{a_n = a | s_n = s\} = \frac{|\{\vec{q} : \vec{q} \in \mathbb{V}_{n-1}(s) \wedge \exists P : (\vec{q}, P) \in \mathbb{Q}_{n-1}(s, a)\}|}{|\mathbb{V}_{n-1}(s)|} \tag{5.8}$$

In our experiments we have used a ϵ-greedy mechanism, that is, with probability ϵ a random action is chosen, and with $(1 - \epsilon)$ an action is chosen according to Expression 5.8. The balance between exploitation and exploration in this framework will be studied in Section 5.3.1.6.

5.3.1.4 Using the Learned Policies

In scalar Q-learning, once the values have been learned, we can derive the optimal (or near-optimal) policy by choosing the action that yields the maximum value. That is, if we are in state s, the chosen action is $argmax_a Q(s, a)$. However, in PQ-learning we cannot use this operator, since we do not have scalar values $Q(s, a)$ any more, but sets $\mathbb{Q}(s, a)$. Moreover, our policies are non-stationary, so they do not condition only in the current state, but also in the current time step.

As our algorithm has been devised to work without the foreknowledge of the preferences or weights for each objective, after the learning phase comes a selection phase, in which we can resort to a particular weight combination or just to a user selection process in which a single solution is directly chosen from the set. Regardless the selection method, the decision has to be made in the time step t_0, that is, at the beginning of the process.

The process for deriving a concrete learned policy in our case starts by computing the V-set of the initial state (see Expression 5.3). Once we have this V-set, we have to select a vector estimate out of it:

$$\vec{q}_{t_0} = \Theta(\mathbb{V}(s_0)) \tag{5.9}$$

Where $\Theta()$ is a selection operator. The action to be taken in state s_0 and time step t_0 is the one whose associated Q-set contains the chosen vector estimate:

$$\pi(s_0, t_0) = a : (\exists P_{t_0} : (\vec{q}_{t_0}, P_{t_0}) \in \mathbb{Q}(s_0, a)) \tag{5.10}$$

In the following time steps the selection operator is no longer needed, because the decision of which policy is going to be followed has been already done. We must resort to the set of indices P associated to the last chosen action, that is, P_{t_0}. In time step t_1, we need to look for the vector estimate inside $\mathbb{V}(s_1)$ indexed in P_{t_0}:

$$\vec{q}_{t_1} = \vec{v}_i : \vec{v}_i \in \mathbb{V}(s_1) \wedge (s_1, i) \in P_{t_0} \tag{5.11}$$

And the chosen action in state s_1 and time step t_1 is:

$$\pi(s_1, t_1) = a : (\exists P_{t_1} : (\vec{q}_{t_1}, P_{t_1}) \in \mathbb{Q}(s_1, a)) \tag{5.12}$$

It could happen that, when $n > 0$, we cannot find the suitable vector estimate $\vec{q}_{t_n}$ inside $\mathbb{V}(s_n)$. This is a symptom that the learning process is not completed, and we still have a vector estimate inside $\mathbb{V}(s_0)$ whose supporting values are no longer non-dominated. In that case we can use the selection operator again in order to choose a vector estimate inside $\mathbb{V}(s_n)$.

Thus, in general, if we are in state s_n at time step t_n, the action to take is given by the expression:

$$\pi(s_n, t_n) = a : (\exists P_{t_n} : (\vec{q}_{t_n}, P_{t_n}) \in \mathbb{Q}(s_n, a)) \tag{5.13}$$

Where

$$\vec{q}_{t_n} = \begin{cases} \Theta(\mathbb{V}(s_0)) & \text{if } n = 0 \\ \vec{v}_i : \vec{v}_i \in \mathbb{V}(s_n) \wedge (s, i) \in P_{t_n - 1} & \text{if } n > 0 \wedge \exists \vec{v}_i \\ \Theta(\mathbb{V}(s_n)) & \text{otherwise} \end{cases} \tag{5.14}$$

5.3.1.5 PQ-learning in Action

Let us assume a state transition diagram for a multi-objective Markov decision process as depicted in figure 5.25. The task is episodic, it always start at state s_1, and and terminates whenever the agent reaches any of the states s_3, s_4 or s_5 (terminal states). Action a_1 has a probabilistic outcome and may lead to states s_2 or s_3. Let us assume that for all terminal states s' we have $\mathbb{V}(s') = \{[\vec{v}_1 = (0, 0), \emptyset]\}$, i.e. each state has a single zero vector that is not supported by any index. Additionally, values for $\mathbb{Q}$ are not defined, since there are no available actions at terminal states.

We will apply PQ-learning to this example assuming $\alpha = 0.1$, and $\gamma = 1$. The evolution of $\mathbb{Q}$ and $\mathbb{V}$ are displayed in table 5.16 for all state-action pairs and non-terminal states, and for each time step. The initial conditions are given for time step $t = 0$. What follows is a description of a possible sequence of transitions that illustrates the application of the PQ-learning rule.

1. A first episode starts at s_1, where a_1 is the only action available. Let us assume the transition leads to s_2 with reward $\vec{r} = (0, 0)$. Since $\mathbb{V}(s_2)$ has two vectors, the application of the PQ-learning rule results in the following,

$$\begin{aligned} \mathbb{N}_1(s_1, a_1) &= \{[(0.9 \times (0,0) + 0.1 \times ((0,0) + (0,0)), \{(s_2, 1)\}] \\ &\quad [(0.9 \times (0,0) + 0.1 \times ((0,0) + (0,0)), \{(s_2, 2)\}]\} \\ \mathbb{U}_1(s_1, a_1) &= \varnothing \\ \mathbb{E}_1(s_1, a_1) &= \varnothing \\ \mathbb{Q}_1(s_1, a_1) &= \{[\vec{v}_1 = (0,0), \{(s_2, 1)\}] \\ &\quad [\vec{v}_2 = (0,0), \{(s_2, 2)\}]\} \end{aligned}$$

2. Assume that at $t = 2$ action a_2 is chosen, leading to state s_4 with reward $\vec{r} = (1000, 2000)$. Notice that after this step, $\mathbb{V}(s_2)$ has only one non-dominated

vector.

$$\begin{aligned}
\mathbb{N}_2(s_2,a_2) &= \{[(0.9\times(0,0)+0.1\times((1000,2000)+(0,0)),\{(s_4,1)\}]\}\\
\mathbb{U}_2(s_2,a_2) &= \varnothing\\
\mathbb{E}_2(s_2,a_2) &= \varnothing\\
\mathbb{Q}_2(s_2,a_2) &= \{[\vec{v}_1=(100,200),\{(s_4,1)\}]\}
\end{aligned}$$

3. At $t=3$ a new episode starts, transitioning again from s_1 to s_2. Now, $\mathbb{V}(s_2)=\{[\vec{v}_1=(100,200)(s_4,1)]\}$, therefore,

$$\begin{aligned}
\mathbb{N}_3(s_1,a_1) &= \varnothing\\
\mathbb{U}_3(s_1,a_1) &= \{[(0.9\times(0,0)+0.1\times((0,0)+(100,200)),\{(s_2,1)\}]\}\\
\mathbb{E}_3(s_1,a_1) &= \varnothing\\
\mathbb{Q}_3(s_1,a_1) &= \{[\vec{v}_1=(10,20),\{(s_2,1)\}]\}
\end{aligned}$$

4. Assume now that at $t=4$ action a_3 is chosen, due to an exploration step. This leads to state s_5 with reward $(2000,1000)$. Since this state is reached for the first time from state-action pair (s_2,a_3), the rule is applied as follows,

$$\begin{aligned}
\mathbb{N}_4(s_2,a_3) &= \{[(0.9\times(0,0)+0.1\times((2000,1000)+(0,0)),\{(s_5,1)\}]\}\\
\mathbb{U}_4(s_2,a_3) &= \varnothing\\
\mathbb{E}_4(s_2,a_3) &= \varnothing\\
\mathbb{Q}_4(s_2,a_3) &= \{[\vec{v}_2=(200,100),\{(s_5,1)\}]\}
\end{aligned}$$

5. At $t=5$ a third episode starts, transitioning once again stochastically from s_1 to s_2. However, now $\mathbb{V}(s_2)$ presents an *extra* vector. Therefore,

$$\begin{aligned}
\mathbb{N}_5(s_1,a_1) &= \varnothing\\
\mathbb{U}_5(s_1,a_1) &= \{[(0.9\times(10,20)+0.1\times((0,0)+(100,200)),\{(s_2,1)\}]\}\\
\mathbb{E}_5(s_1,a_1) &= \{[(0.1\times((0,0)+(200,100)),\{(s_2,2)\}]\}\\
\mathbb{Q}_5(s_1,a_1) &= \{[\vec{v}_1=(19,38),\{(s_5,1)\}]\\
&\quad\ [\vec{v}_3=(20,10),\{(s_2,2)\}]\}
\end{aligned}$$

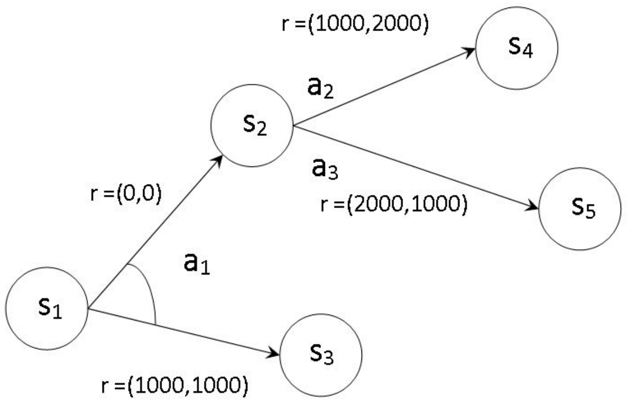


Figure 5.25: Sample state transition diagram.

6. Assume the episode terminates with a new transition from s_2 to s_4. The update rule is applied as follows,

$$\begin{aligned}
\mathbb{N}_6(s_2,a_2) &= \varnothing \\
\mathbb{U}_6(s_2,a_2) &= \{[(0.9\times(100,200)+0.1\times((1000,2000)+(0,0)),\{(s_4,1)\}]\} \\
\mathbb{E}_6(s_2,a_2) &= \varnothing \\
\mathbb{Q}_6(s_2,a_2) &= \{[\vec{v}_1=(190,380),\{(s_4,1)\}]\}
\end{aligned}$$

7. Finally, let us assume the last episode leads from s_1 to s_3. This illustrates the case where each vector in $\mathbb{Q}(s_1,a_1)$ is supported by two vectors, one from s_2, and other from s_3,

$$\begin{aligned}
\mathbb{N}_7(s_1,a_1) &= \{[(0.9\times(19,38)+0.1\times((1000,1000)+(0,0)),\{(s_2,1)(s_3,1)\}] \\
&\quad [(0.9\times(20,10)+0.1\times((1000,1000)+(0,0)),\{(s_2,2)(s_3,1)\}]\} \\
\mathbb{U}_7(s_1,a_1) &= \varnothing \\
\mathbb{E}_7(s_1,a_1) &= \varnothing \\
\mathbb{Q}_7(s_1,a_1) &= \{[\vec{v}_1=(117.1,134.2),\{(s_2,1)(s_3,1)\}] \\
&\quad [\vec{v}_3=(118,109),\{(s_2,2)(s_3,1)\}]\}
\end{aligned}$$

t	s	(s,a)	$\mathbb{Q}_t(s,a)$	$\mathbb{V}_t(s)$
0	s_1	(s_1,a_1)	$\vec{v}_1 = (0,0)\ \emptyset$	$\vec{v}_1 = (0,0)\ \emptyset$
	s_2	(s_2,a_2)	$\vec{v}_1 = (0,0)\ \emptyset$	$\vec{v}_1 = (0,0)\ \emptyset$
		(s_2,a_3)	$\vec{v}_2 = (0,0)\ \emptyset$	$\vec{v}_2 = (0,0)\ \emptyset$
1	s_1	(s_1,a_1)	$\vec{v}_1 = (0,0)\ (s_2,1)$	$\vec{v}_1 = (0,0)\ \ (s_2,1)$
			$\vec{v}_2 = (0,0)\ (s_2,2)$	$\vec{v}_2 = (0,0)\ \ (s_2,2)$
	s_2	(s_2,a_2)	$\vec{v}_1 = (0,0)\ \emptyset$	$\vec{v}_1 = (0,0)\ \emptyset$
		(s_2,a_3)	$\vec{v}_2 = (0,0)\ \emptyset$	$\vec{v}_2 = (0,0)\ \emptyset$
2	s_1	(s_1,a_1)	$\vec{v}_1 = (0,0)\ (s_2,1)$	$\vec{v}_1 = (0,0)\ \ (s_2,1)$
			$\vec{v}_2 = (0,0)\ (s_2,2)$	$\vec{v}_2 = (0,0)\ \ (s_2,2)$
	s_2	(s_2,a_2)	$\vec{v}_1 = (100,200)\ (s_4,1)$	$\vec{v}_1 = (100,200)\ (s_4,1)$
		(s_2,a_3)	$\vec{v}_2 = (0,0)\ \emptyset$	
3	s_1	(s_1,a_1)	$\vec{v}_1 = (10,20)\ (s_2,1)$	$\vec{v}_1 = (10,20)\ \ (s_2,1)$
	s_2	(s_2,a_2)	$\vec{v}_1 = (100,200)\ (s_4,1)$	$\vec{v}_1 = (100,200)\ (s_4,1)$
		(s_2,a_3)	$\vec{v}_2 = (0,0)\ \emptyset$	
4	s_1	(s_1,a_1)	$\vec{v}_1 = (10,20)\ (s_2,1)$	$\vec{v}_1 = (10,20)\ \ (s_2,1)$
	s_2	(s_2,a_2)	$\vec{v}_1 = (100,200)\ (s_4,1)$	$\vec{v}_1 = (100,200)\ (s_4,1)$
		(s_2,a_3)	$\vec{v}_2 = (200,100)\ (s_5,1)$	$\vec{v}_2 = (200,100)\ (s_5,1)$
5	s_1	(s_1,a_1)	$\vec{v}_1 = (19,38)\ (s_2,1)$	$\vec{v}_1 = (19,38)\ \ (s_2,1)$
			$\vec{v}_3 = (20,10)\ (s_2,2)$	$\vec{v}_3 = (20,10)\ \ (s_2,2)$
	s_2	(s_2,a_2)	$\vec{v}_1 = (100,200)\ (s_4,1)$	$\vec{v}_1 = (100,200)\ (s_4,1)$
		(s_2,a_3)	$\vec{v}_2 = (200,100)\ (s_5,1)$	$\vec{v}_2 = (200,100)\ (s_5,1)$
6	s_1	(s_1,a_1)	$\vec{v}_1 = (19,38)\ (s_2,1)$	$\vec{v}_1 = (19,38)\ \ (s_2,1)$
			$\vec{v}_3 = (20,10)\ (s_2,2)$	$\vec{v}_3 = (20,10)\ \ (s_2,2)$
	s_2	(s_2,a_2)	$\vec{v}_1 = (190,380)\ (s_4,1)$	$\vec{v}_1 = (190,380)\ (s_4,1)$
		(s_2,a_3)	$\vec{v}_2 = (200,100)\ (s_5,1)$	$\vec{v}_2 = (200,100)\ (s_5,1)$
7	s_1	(s_1,a_1)	$\vec{v}_1 = (117.1,134.2)\ \ (s_2,1)(s_3,1)$	$\vec{v}_1 = (117.1,134.2)\ \ (s_2,1)(s_3,1)$
			$\vec{v}_3 = (118,109)\ \ (s_2,2)(s_3,1)$	$\vec{v}_3 = (118,109)\ \ (s_2,2)(s_3,1)$
	s_2	(s_2,a_2)	$\vec{v}_1 = (190,380)\ (s_4,1)$	$\vec{v}_1 = (190,380)\ (s_4,1)$
		(s_2,a_3)	$\vec{v}_2 = (200,100)\ (s_5,1)$	$\vec{v}_2 = (200,100)\ (s_5,1)$

Table 5.16: Evolution of estimated values over time with PQ-learning. Thicker horizontal lines indicate division between training episodes.

5.3.1.6 Benchmark Results

Now we analyse the results obtained when applying PQ-learning to two benchmark problems proposed by Vamplew et al. (2011). We have measured the train/test performance, suitable for multiple-policy algorithms, as they are likely to be used in an off-line setting (Vamplew et al., 2011). As several policies are learned at once, every time a set of policies is evaluated we compute the hypervolume of the set of accumulated vectorial rewards obtained by each intermediate learned policy (Vamplew et al., 2011). The hypervolume of a set of vectors $\mathbb{S}$, given a reference point dominated by every vector in $\mathbb{S}$, is the volume of the space that is dominated by the points in $\mathbb{S}$ and that dominates the reference point. The quality of a set of learned policies is measured comparing its hypervolume with the hypervolume of the true front.

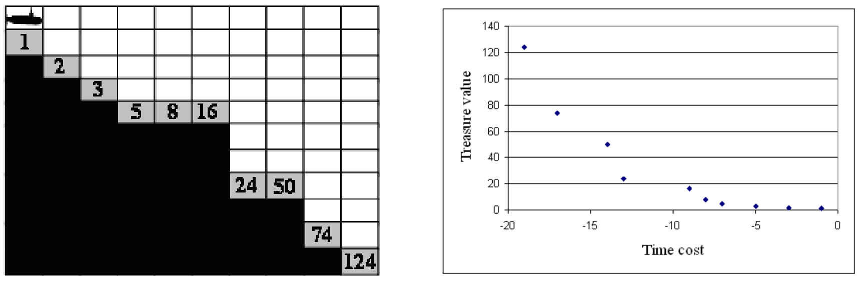
All results in this section have been averaged over ten runs of the algorithm, on a Intel Core i7 CPU @2.80Ghz with 8 GB RAM and Windows 7.

5.3.1.6.1 Deep Sea Treasure. The first problem considered is *Deep Sea Treasure* (DST). This problem can be used to test if a MORL algorithm is able to find all the state vectors for a problem where these define a non-convex frontier in reward space (Vamplew et al. (2008)). DST has two objectives, and its true front of non-dominated solutions contains ten state vectors and has a hypervolume of 535, according to the reference point $(-20, 0)$.

The environment is a grid of 10 rows and 11 columns, as we can see in Figure 5.26(a). The agent controls a submarine that searches for undersea treasures. There are ten treasure locations with different values; the first objective is to minimize the time that the submarine takes to reach the treasure, and the second one is to maximize the value of the achieved treasure. The task is episodic, with each episode starting in the top-left position of the grid and ending when a treasure is reached or after 1000 actions have been taken by the agent. At each step, four actions are available: moving one square to the top, right, bottom or left. If an action would move the submarine outside of the grid, then the position of the submarine remains unchanged.

The reward received at each step is a vector of 2 elements; the first one is a punishment of -1 for the time consumed, and the second one is the value of the achieved treasure, that will be 0 in all steps except when the agent reaches a treasure location (the values are indicated in Figure 5.26(a)). In Figure 5.26(b) we can see the ten non-dominated accumulated rewards associated to the non-dominated policies of this problem.

We need to set three parameters for PQ-learning: the discount rate, the learning rate and the exploration rate. In our experimentations we have found that the process



(a) DST Environment (b) DST Frontier

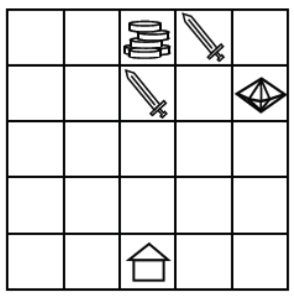
Figure 5.26: Deep Sea Treasure problem: Environment (a) and Frontier (b) (reproduced from Vamplew et al. (2008))

is more sensitive for the last one, that is, for different values of epsilon. We show the train/test performance for $\epsilon \in 0.2, 0.3, 0.4, 0.5$ in Figure 5.28(a), being the higher value the one that yielded the best performance, as the true hypervolume was reached in less episodes. This high exploration rate is a feature particular to the DST problem (it is not a necessary requirement for our algorithm). The discount rate was set to 1 and the learning rate to 0.1.

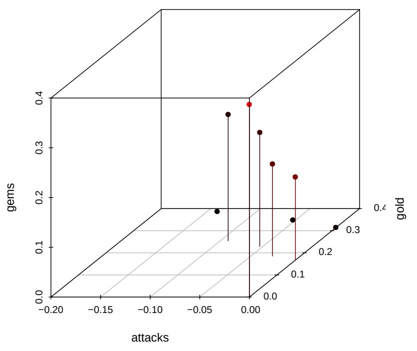
The hypervolume of the accumulated rewards yielded by every set of intermediate learned policies is averaged over 10 runs, along 100000 episodes. The average execution time was 2471.55 seconds. PQ-learning converges to the full set of nondominated solutions, in spite of these defining a non-convex frontier.

5.3.1.6.2 Resource Gathering. The second problem considered is *Resource Gathering* (RG). RG has three objectives, its true front contains eight vectors, and has an hypervolume of 1.8299, according to the reference point $(-1, -1, -1)$.

RG was first proposed by Barrett & Narayanan (2008). The environment is a grid of five rows and five columns, as shown in Figure 5.27(a). The problem is a resource gathering task, as required in many strategy games. An agent has to gather gold and gems while minimizing the attack of enemies, given that the agent is attacked in the enemies position with a probability of 0.1. The first objective is to minimize the punishment for being attacked, the second one is to maximize the gold taken and the third one is to maximize the gems taken. Although the original problem describes a continuing taks, we have reformulated it to be episodic. This allows the exact calculation of the front by means of the Multi-Objective Dynamic Programming method by White (1982) in an affordable time. Each episode starts at the home position, and ends when some of the followings events occurs: (1) The agent returns home with gold or gems, (2) the agent is attacked (in this case, it automatically returns home, losing all the gathered gold and gems, if any) or (3) 1000 actions have been taken by the agent. At each step, four actions are available: moving one square to the top, right, bottom or left. If an action would move the agent outside of the grid, then its position remains unchanged.



(a) RG Environment



(b) RG Frontier

Figure 5.27: Resource Gathering problem: Environment (a) (reproduced from Barrett & Narayanan (2008)) and Frontier (b)

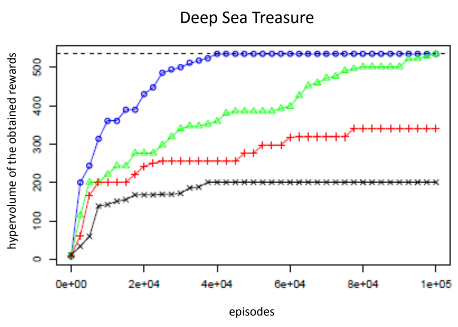
The reward received at each step is a vector of three elements. The first one is the punishment for being attacked, being 0 at every step except when the agent visits an enemy position and is attacked. The second one is the reward for gathering gold, that will be 0 at every step except when the agent is back home with gold, in which case it is 1. The third element is like the second one, but for gems instead of gold. In Figure 5.27(b) we can see the eight non-dominated policies of this problem, given a discount rate $\gamma = 0.9$.

We have applied PQ-learning ten times to RG. The parameters for these experiments are the following:

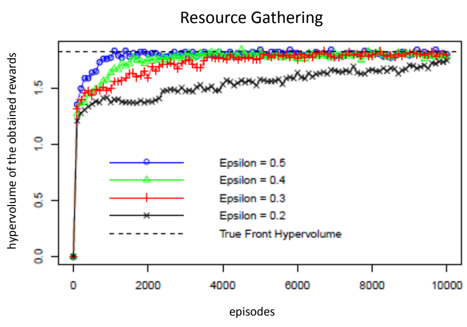
- Discount rate $\gamma = 0.9$ (to favour shorter paths)
- Learning rate $\alpha = 0.1$
- Exploration rate $\epsilon \in 0.2, 0.3, 0.4, 0.5$

Being the performance for $\epsilon = 0.5$ the best one, as the hypervolume was approximated in less episodes.

In Figure 5.28(b) we can see the train/test performance, averaged over 10 runs, along 10000 episodes. The average execution time was $730'07$ seconds. Again, PQ-learning converges to the full set of nondominated vectors in the true front.



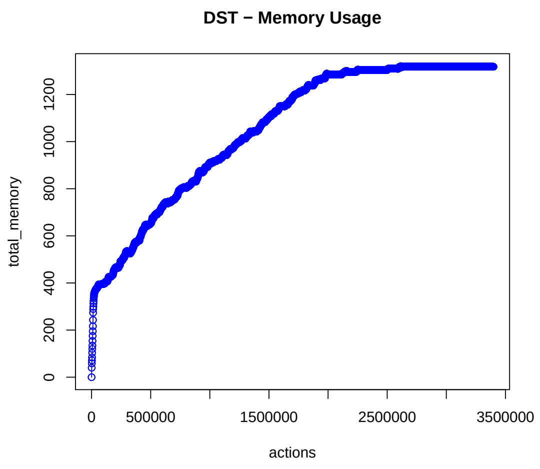
(a) DST Results



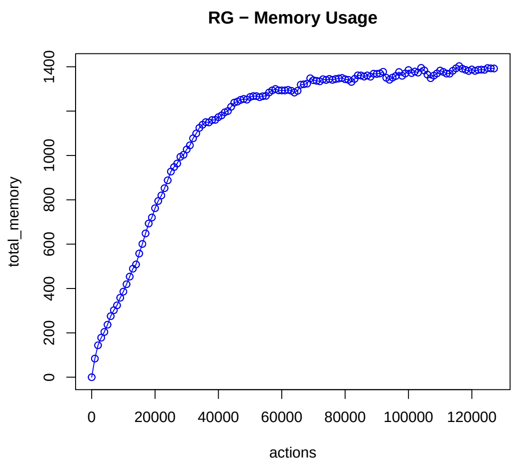
(b) RG Results

Figure 5.28: Train/test performance for DST (a) and RG (b)

5.3.1.6.3 Memory Requirements. We have also studied the memory requirements for each problem, in terms of the number of stored vectors inside the Q-sets and the V-sets. Results are depicted in Figure 5.29. As we can see, the number of stored vectors grows as the agent explores the environment, but it remains manageable and stable once the true front has been approximated.



(a) Memory requirements for DST



(b) Memory requirements for RG

Figure 5.29: Memory requirements (total number of stored vectors) of PQ-learning for the DST (a) and RG (b) problems

5.3.1.7 Comparison to Scalarized Q-learning

In this section we perform an empirical comparison of PQ-learning to the linear scalarization multi-policy approach, consisting in running Q-learning with a linear combination of the rewards and several sets of weights. We use a dichotomic scalarizing procedure analogous to the one described in Balachandran & Gero (1984) to solve a test set of biobjective problems.

Algorithm 1 (SCAL) displays an iterative description of the scalarizing procedure that finds the costs of all supported nondominated solutions to a bi-objective problem. The algorithm repeatedly optimizes a linear function $f(x, y) = w_1 \times x + w_2 \times y$ (that combines the values x and y of both scalar objectives for each candidate solution) with different values for w_1, w_2. It receives as input parameters the costs of the two extreme solutions p and q. The former is obtained solving the problem with $w_1 >> w_2$, and the latter with $w_1 << w_2$. If $p \neq q$, then the procedure is invoked to find all nondominated supported costs between p and q. The algorithm uses a stack S to keep track of all the optimizations that need to be carried out. Each one tries to find a new supported solution between two pairs of points a and b. In our case, the optimization would be carried out using the scalar Q-learning algorithm. If the new point found c is in the same line that joins a and b, then no new supported solution costs can be found between them. Otherwise, c is a new supported nondominated cost, and two new optimizations will need to be carried out, one between a and c, and another one between c and a. Therefore, these two pairs of points are stored in the stack. The procedure terminates when the stack becomes empty. For a biobjective problem with $n > 1$ supported solutions, this procedure guarantees that all of them are found in $2n - 1$ scalar applications of Q-learning.

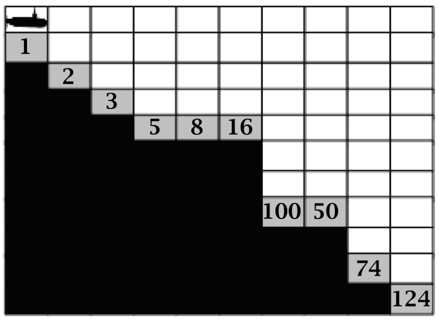
As in the previous section, performance has been measured according to the number of learning steps needed to reach the whole set of solutions inside $\mathbb{V}^*(s_0)$, where s_0 is the initial state of the environment. In the case of the scalarized procedure, this number is the sum of the steps in each individual application of scalar Q-learning.

Notice that the number of reachable solutions varies according to the method that is being used, as linear scalarization techniques can only find supported solutions.

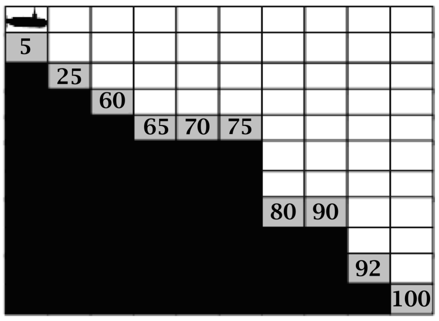
The original DST frontier only has two supported solutions out of the ten Pareto-optimal ones, and all treasures are reachable by Pareto-optimal policies. In order to perform a more complete comparison, we have compared PQ-learning and the scalarized version for three additional variants with different frontier configurations. In Figure 5.30 we illustrate the environments for each variant, and in Table 5.17 we gather the different features of their respective frontiers. In the following we explain in detail the considered variants.

```
**Input**: $p, q$ (2D points representing extreme solution costs), $\mathcal{P}$ (a problem)
**Output**: $R$ (list of supported solution costs)
**Local variables**: $a, b, c$ (2D points representing solution costs), $S$ (a stack)
$R \leftarrow$ list($p, q$);                              // A new list with $p$ and $q$
$S \leftarrow$ empty-stack;                                     // Create a new stack
$S \leftarrow$ push([$p, q$], $S$);          // Push a vector with $p$ and $q$ in the stack
while $S$ is not empty do
    // Pop the next pair of points from the stack:
    [$a, b$] $\leftarrow$ pop($S$);
    // Calculate the parameters of the new
    // linear objective function:
    $w_1 \leftarrow a_y - b_y$;
    $w_2 \leftarrow b_x - a_x$;
    // Solve $\mathcal{P}$ to find a new solution and get
    // its cost vector $c$:
    $c \leftarrow$ optimize($\mathcal{P}, w_1, w_2$)
    if ($w_1 \times a_x + w_2 \times a_y$) $\neq$ ($w_1 \times c_x + w_2 \times c_y$) then
        // $c$ is the cost of a new supported solution
        $S \leftarrow$ push([$a, c$], $S$);                  // Push new pair in the stack
        $S \leftarrow$ push([$c, b$], $S$);                  // Push new pair in the stack
        $R \leftarrow$ add($c, R$) ;                               // Add $c$ to the result
    end
end
return $R$ ;
```

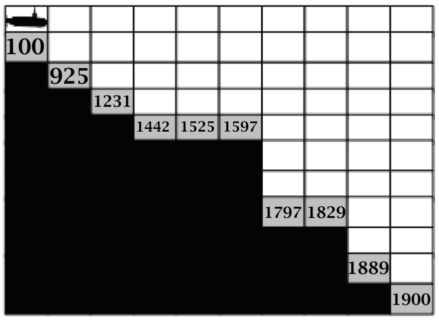
Algorithm 1: Algorithm to obtain all supported costs to a bi-objective optimization problem. Method ‘optimize($\mathcal{P}, w_1, w_2$)’ solves problem $\mathcal{P}$ with a linear objective function $f(x, y) = w_1 \times x + w_2 \times y$ that combines the values x and y of both scalar objectives for each candidate solution.



(a) DST-2 Environment



(b) DST-3 Environment



(c) DST-4 Environment

Figure 5.30: Deep Sea Treasure problem variants: environments for DST-2 (a), DST-3 (b) and DST-4 (c)

5.3.1.7.1 DST-2 In this second variant of DST we aim to test the algorithms behaviour when there are some non-optimal undersea treasures. Particularly, we have changed the value of the seventh treasure from 24 to 100, so treasures 50 and 74 are not Pareto-optimal any more.

5.3.1.7.2 DST-3 In the third variant of DST we want to test the algorithms behaviour when the number of unsupported solutions is closer to the number of Pareto-optimal solutions. Particularly, in this setting there are five supported solutions (as opposed to the two solutions in the original DST environment) out of ten Pareto-optimal solutions.

5.3.1.7.3 DST-4 In this fourth setting of DST we want to test the algorithms behaviour when the number of unsupported solutions equals the number of Pareto-optimal solutions, that is, when the frontier is fully convex. In this variant all ten Pareto-optimal solutions are supported.

DST setting	Supported solutions	Pareto solutions
Original	2	10
DST-2	3	8
DST-3	5	10
DST-4	10	10

Table 5.17: Frontier features for the considered variants of the Deep Sea Treasure environment

Problem	Avg. steps	Max steps
Original (SCAL)	1074645.3	1494433
Original (PQ)	1260451.1	3478320
DST-2 (SCAL)	14786851.0	23931695
DST-2 (PQ)	23905464.7	39318415
DST-3 (SCAL)	5862617.5	10698936
DST-3 (PQ)	4602536.4	6229090
DST-4 (SCAL)	15618235.0	19845973
DST-4 (PQ)	6294062.1	10193190

Table 5.18: Scalarized algorithm vs. PQ-learning: training steps until convergence of $V(0,0)$, over 100 agents

5.3.1.7.4 Results.

For each setting we have executed 100 agents of the scalarized algorithm and 100 agents of PQ-learning, with the same parameter setting that we used for solving the original DST problem with PQ in Section 5.3.1.6.

In Table 5.18 we gather the average and maximum number of training steps that each method needs to converge to the whole set of reachable solutions in $\mathbb{V}^*(s_0)$.

5.3.1.7.5 Discussion.

As we have already mentioned, it is well known that scalarizing methods based on linear functions can only obtain the subset of supported solutions of a MOMDP. On the contrary, PQ-learning can approximate the whole Pareto front.

It can be argued that in some cases it suffices to learn the set of supported solutions. However, in the light of the obtained results, even when only supported solutions are sought, in some cases PQ-learning can outperform the linear scalarizing method in terms of training steps, as we can see in Table 5.18. Particularly, it has been the case of the problems DST-3 and DST-4. It seems reasonable to speculate that when the ratio of supported solutions/Pareto solutions grows, PQ-learning tends to need less training steps than the linear scalarized method. In any case, if only supported solutions are required, these can be easily obtained from the set of all Pareto-optimal reward vectors X provided by PQ-learning. For example, algorithm 1 could be used. In this case the "optimize" function would only need to test the linear function over all points in X and return the optimal one.

5.3.2 Example: Compactness-Perimeter Problem

Once we have described PQ-learning as a general MORL technique, we can illustrate how can it be applied to a design problem. The application of this algorithm is more direct than in the previous single-policy methodology case, as we do not need to establish weights for defining the rewards. Regarding generalization techniques, the integration of function approximation with PQ-learning remains future work, so by the moment we just tackle tabular approaches. As a result, now we do not need to use features.

Let us return to our geometric example, in which we seek to generate rectilinear polygons with nine tiles according to two conflicting design requirements: obtain a high compactness factor and a high perimeter.

5.3.2.1 Definition of Rewards

The reward is in this case a vector of two values, namely the compactness factor and the perimeter:

$$r(s,a) = \begin{cases} (area(s')/perimeter(s')^2, perimeter) & \text{if } s' \text{ has nine tiles} \\ \vec{0} & \text{otherwise} \end{cases} \tag{5.15}$$

Where s' is the state that represents the shape generated by applying the action a over the state s. For the intermediate shapes with less than nine tiles, the reward is defined as the zero vector.

The Pareto frontier of possible optimal accumulated rewards for shapes with nine tiles is depicted in Figure 5.31. This frontier is non-convex, meaning that a linear scalarization approach would not be useful for discovering every policy yielding every Pareto optimal solution: only policies for the supported ones (in this case, the ones in the right and left extremes of the frontier) would be discovered.

As we can see in Figure 5.31, there are five non-dominated points, each one corresponding to a different optimal accumulated reward. Hence, the solution to the problem is a set of five policies, each one producing different kinds of shape, which are depicted in Figure 5.32.

We can see that, except for the accumulated reward that only maximizes the compactness (point $(1, 0.6)$), for which the unique associated shape is a square of 3×3

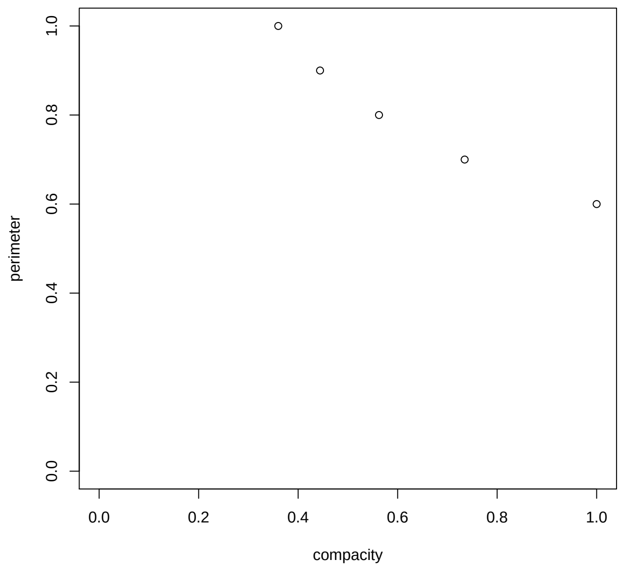


Figure 5.31: Pareto frontier of the compacity-perimeter problem

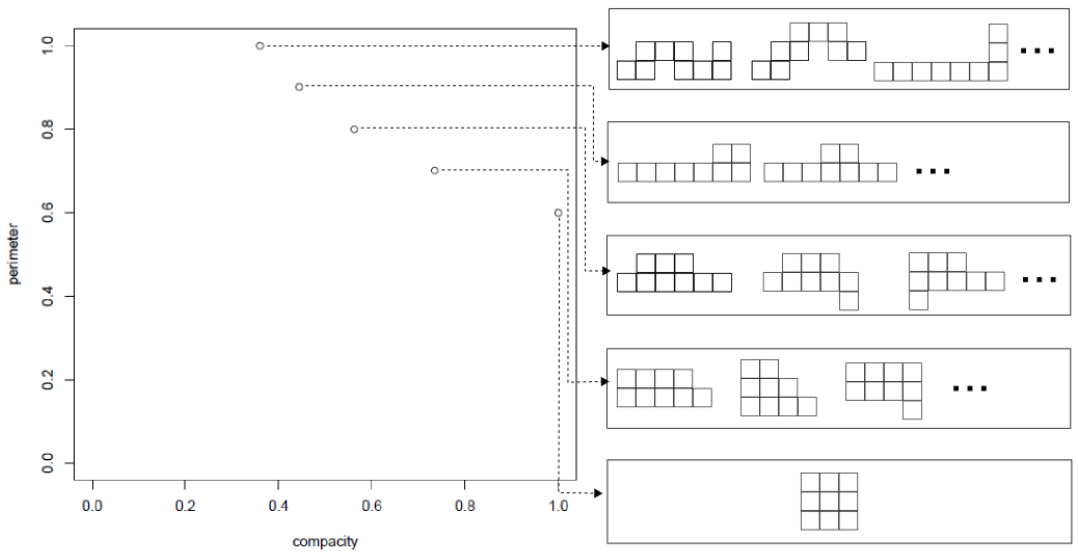


Figure 5.32: Kinds of shape solutions for the compactness-perimeter problem

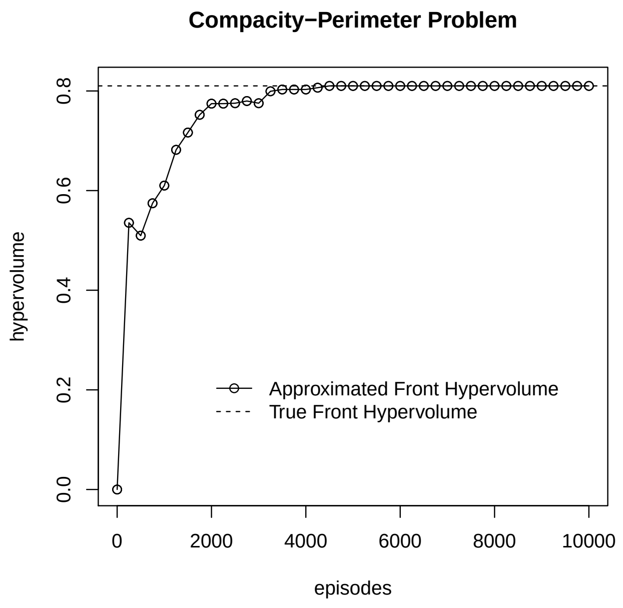


Figure 5.33: Train/test performance of PQ-learning applied to the compactness-perimeter problem

tiles, there is more than one possible shape associated to each long-term value.

5.3.2.2 Performance

The train/test performance of PQ-learning applied to our geometric problem (averaged over 10 runs) is depicted in Figure 5.33. The parameters has been set to the following values: $\gamma = 1, \alpha = 0.1, \epsilon = 0.5$. We can see that the hypervolume of the set of accumulated rewards yielded by the intermediate sets of learned policies, converges to the true hypervolume of the front, which is 0.8102. Both hypervolumes have been calculated considering $(0, 0)$ as the reference point.

Due to the nature of PQ-learning, a particular policy of a given run of the algorithm will yield the same shape every time it is executed. This is due to the index search that must be performed in order to follow a concrete policy, which was described in Section 5.3.1.4. So, if we want to obtain more than one of the shapes associated with a given accumulated reward, we must run PQ-learning several times.

One particular run of PQ-learning yielded an initial V-set (that is, the V-set of the initial state) with the following five values: $(1, 0.6), (0.5625, 0.8), (0.4444, 0.9), (0, 7347, 0.7)$ and $(0.3597, 0.9992)$, associated with five different policies. In order to apply one of these policies, we must choose one of the values and then follow the process described in Section 5.3.1.4, using the indexes in order to apply the proper actions that in the

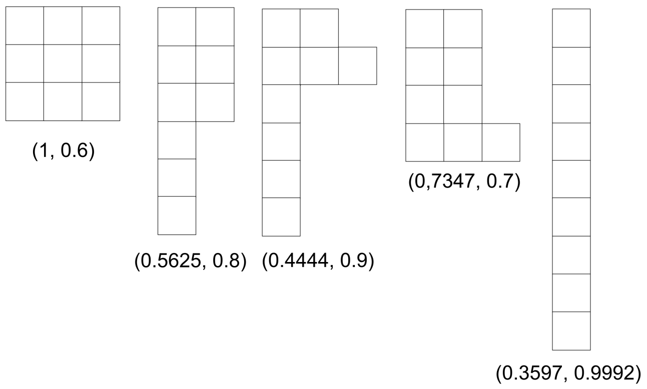


Figure 5.34: Shapes produced by the five policies learned in a single run of PQ-learning, applied to the compactness-perimeter problem

end will yield the desired long-term value. This process has been followed for the five learned policies, producing the five shapes depicted in Figure 5.34.

Chapter 6

Conclusions and Future Work

What is the future of design?
(no answer)

Question: Madame L'Amic. Answer: Charles Eames

Design problems usually involve the consideration of multiple criteria of different nature, involving technical, economical, social, and environmental needs, among others. Traditional CAD tools help designers in the representation, modification, analysis, documentation and evaluation of their designs. A more sophisticated role is that of computational design, which consists in the synthesis of new design solutions. This thesis has focused on this non trivial role, being its general goal the development of new computational design techniques able to deal with design criteria.

Some aspects of the design process can be understood as a search or exploration over a space of design alternatives. This perspective facilitates the exploitation of computational techniques in order to implement methods intended to synthesize solutions that fulfil the purpose of a given design problem.

In order to develop computational design methodologies, we need a generative system able to represent and generate a design space. In this thesis we have considered the framework of shape grammars due to its wide usage along the computational design literature, and also to its versatility. The traditional approach for using shape grammars as a computational design method consists in hard-coding the whole set design criteria inside the shape rules, giving room to expert shape grammars that are difficult to create, modify and maintain. They also promote predictable solutions, as the shape grammar designer crafts the rules with the foreknowledge of the requirements that the produced shapes need to meet. In this dissertation we have considered another alternative, which consists in reducing or minimizing the number of criteria hard-coded inside the rules. We deal thus with more naive shape grammars that cannot produce feasible solutions by themselves, and need for a control method that guides the derivation towards good designs. Particularly, we have relied on search processes and reinforcement learning methods in order to perform this control.

In the following section we detail the main conclusions of this dissertation and the possible lines of future work.

6.1 Conclusions

The main conclusions of this thesis can be summarized as follows:

1. **A classification scheme for possible approaches to computational design based on shape grammars has been proposed.** The different classes depend on the nature of the involved rules and the external control methods. Concretely, we have proposed a scheme in which two aspects are considered in order to classify shape grammar-based computational design techniques. The first aspect considers the amount of design criteria codified inside the shape rules, being totally expert shape grammars those in which the whole set of design requirements is codified in this manner. The less requirements are codified inside the rules, the more naive they can be considered. The second classification facet considers the complexity of the control method, ranging from non-existing control techniques to sophisticated ones like search methods or reinforcement learning algorithms. These two aspects give rise to the following four classes: (1) expert grammars without control, (2) expert grammars with sophisticated control, (3) naive grammars with sophisticated control and (4) naive grammars without control. The first class stands for the traditional shape grammars that have been used in many areas. The second and third classes encompass the main contributions of this thesis. In addition, the third class, standing for naive shape grammars with sophisticated control, also covers some recent computational design techniques, such as shape annealing or evolutionary algorithms applied to shape grammars. The fourth class has also been considered in the context of this dissertation, with the goal of studying how architecture students respond to computational design tools in which randomness is highly present.

2. **A computational design technique based on expert shape grammars and sophisticated control has been developed.** Concretely, this methodology is based on the idea of encoding some of the design requirements into rules, while explicitly using others for evaluating the shapes produced along the generative process and guiding it towards good designs. The involved shape grammars

are thus less expert than in the traditional approach of encoding every design requirement into the rules. In this setting we distinguish between design requirements that are easier to specify in a geometric manner and requirements that are amenable to be expressed as logic predicates. The former are encoded inside shape rules enhanced with layers, which can help in the organization and representation of the shapes, as well as in reducing the derivation time. The latter are programmed as design constraints and goals, which are used to guide a depth-first search algorithm.

3. **A software tool (ShaDe) for editing and executing layered shape grammars with constraints and goals has been developed.** ShaDe allows the creation of generic shape grammars that can be organized into layers. The typical CAD facilities related to layers, such as selective visualization and selection of shapes belonging to different layers, are available in ShaDe. It also provides the user with a set of predefined design constraints and goals, as well as means for customizing them and even creating new ones. The layered shape grammars can be executed inside ShaDe in order to produce design solutions complying with the specified constraints and goals.

4. **A computational design technique based on naive shape grammars and sophisticated control has been developed.** Concretely, this methodology is based on the idea of using the whole set of design requirements as the rewards of a reinforcement learning process, in order to learn a heuristic (or a set of heuristics) that determines how to apply the rules of the generative system. In the generation process, we just rely on the newly discovered information in order to apply the rules, so design requirements are used in an implicit manner. Two alternatives for learning the policies have been presented, depending on how we deal with the multi-objective nature of design problems. Namely, in the first alternative the rewards are scalarized according to a set of design preferences that are known a priori, so the algorithm yields a unique policy for each shape grammar. The second alternative does not scalarize the rewards, so the algorithm learns multiple policies that can be used to produce a set of Pareto optimal solutions.

5. **A new multi-objective reinforcement learning technique (PQ-learning) has been proposed.** In the context of the non-scalarizing methodology, we have developed a new reinforcement learning technique that directly extends the algorithm Q-learning in order to deal with vectorial rewards. This newly proposed method has been tested according to two benchmark problems.

6. **The developed methodologies have been put into practice in different scenarios related to the architectural domain**. Concretely, the technique based on layered shape grammars with constraints and goals has been used to synthesize two kinds of designs: (1) housing units configured according to the Hele Module proposed by the architect Rafael Leoz, and (2) virtual restaurants that can be used as scenarios for video games. The computational design technique based on naive shape grammars and reinforcement learning has been put into practice in two application cases related to the generation of housing unit schemes complying with habitability and energy-efficient requirements.

7. **Two empirical studies with architecture students have been performed.** Particularly, they were associated to the methodologies corresponding to naive shape grammars with and without control. The experiment corresponding to the methodology of naive shape grammars without control was carried out in order to study how architecture students react to a computational design tool where random commands are allowed. The main conclusion was that users are open to explore randomness, so computational design tools should provide a certain amount of that; however, these tools should also provide means for controlling and taming such randomness. The experiment associated to the methodology of naive grammars with control was performed in order to validate such technique when it is used to generate habitable housing units schemes, being the main conclusion that the tool can provide interesting starting points for architectural projects.

6.2 Future Work

In the following we detail those aspects of this dissertation that deserve further investigation:

- The extension of the proposed methodologies to more complex forms of shape grammars, such as three-dimensional or even parametric ones, in which shapes are defined not only by the disposition of geometric elements but also by variable parameters, such as the width and height of a rectangle.

- The application cases involving naive shape grammars and scalarized reinforcement learning relies on a division of the tackled design problem into several phases. This separation of the generation process in sequential phases manages to reduce the set of requirements that are taken into account at every step. However, it also introduces an important shortcoming that may have an impact when dealing with more complex design problems: the learning process deals only with the local requirements of each phase, and thus the policies cannot reflect global issues. For example, in the context of our housing design example, a globally optimal policy would give insight about how to place the kitchen modules regarding the future placing of non-specialized spaces. The shortcoming of not dividing the process into these phases is the arising necessity for considering the whole set of design criteria in a unique phase. A possible way of circumventing this issue is the integration with more powerful, non-linear generalization techniques such as neural networks.

- We have shown that the proposed methodology involving naive shape grammars and PQ-learning can be used to deal with geometric problems, but more research is needed in order to apply the methodology in real-world scenarios. We believe that this can be managed by means of the integration of PQ-learning with generalization techniques.

- Finally, the application of the proposed methodologies to other design domains is also an important line of research.

Appendix A

Resumen

A.1 Introducción

El diseño es una tarea compleja que implica considerar criterios de diversa índole para alcanzar un propósito. Involucra la integración de necesidades técnicas, económicas, sociales y medioambientales, entre otras.

Hoy en día, la importancia de los ordenadores en el proceso de diseño es incuestionable. Las herramientas CAD (computer-aided design, diseño asistido por ordenador) han expandido las capacidades humanas, proporcionando a los diseñadores mecanismos precisos para representar, describir, visualizar, organizar y simular proyectos de diseño.

En esta tesis partimos de la premisa de que los ordenadores también pueden participar en la *síntesis* de nuevas soluciones de diseño. Esta nueva área de investigación ha sido bautizada como *diseño computacional* (Cagan et al., 2005), y se ha visto acelerada gracias a diversos avances en la comprensión del diseño desde una perspectiva computacional. Desde cierto punto de vista, algunos aspectos del proceso de diseño pueden ser entendidos como una búsqueda o exploración en un espacio de alternativas de diseño (Simon, 1973). El hilo conductor de esta tesis consiste en la idea de emplear técnicas de inteligencia artificial para tratar este espacio de diseño de diversas maneras. Estas técnicas han de considerar requisitos de diseño de diversa naturaleza, relevantes para el propósito del problema considerado.

En este contexto, se necesitan mecanismos que puedan representar y generar de manera explícita el espacio de diseño, lo cual nos conduce al área de los sistemas generativos. Concretamente, en este trabajo nos centraremos en el formalismo de las gramáticas de formas (Stiny & Gips, 1972), ampliamente utilizado en el área del diseño computacional debido a su versatilidad.

Los criterios de diseño pueden ser directamente codificados en las reglas de los sistemas generativos, dando lugar a sistemas expertos que siempre sintetizan soluciones factibles. Sin embargo, estos sistemas son generalmente difíciles de crear, modificar y mantener. Otra alternativa es la externalización parcial o total de estos criterios, dando paso a sistemas más ingenuos en el sentido de que no pueden producir buenas soluciones por ellos mismos: necesitan un método de control que asegure la producción de diseños factibles. En esta tesis seguiremos este último enfoque, y utilizaremos distintas técnicas computacionales y de inteligencia artificial tales como algoritmos de búsqueda y métodos de aprendizaje por refuerzo (Sutton & Barto, 1998), siempre desde una perspectiva multi-criterio.

En esta tesis profundizaremos en dos alternativas para combinar el sistema generativo de las gramáticas de formas con un mecanismo externo de control, dependiendo de la manera de codificar y emplear los criterios de diseño. La primera alternativa consiste en reducir el número de criterios codificados dentro de las reglas, tratando parte de ellos de una manera explícita en la que son directamente usados para evaluar las formas generadas en cada paso del proceso generativo y guiar dicho proceso hacia soluciones factibles. La segunda alternativa va un paso más allá, e intenta utilizar el conjunto completo de criterios de diseño en un proceso de aprendizaje automático para obtener un heurístico. Tal heurístico determinará cómo aplicar las reglas de la gramática de formas. Así, en el proceso generativo únicamente se utiliza de manera explícita la nueva información aprendida, y los criterios de diseño se usan implícitamente.

A continuación describimos brevemente las principales contribuciones de esta tesis:

1. Un esquema de clasificación para posibles aproximaciones al diseño computacional basadas en gramáticas de formas. Las diferentes clases dependerán en la naturaleza de la reglas involucradas, la manera de formalizar los criterios de diseño y los métodos de control externo empleados.

2. El desarrollo de una metodología basada en la idea de codificar algunos criterios en las reglas de las gramáticas de formas, y utilizar el resto de manera explícita para evaluar las formas producidas a lo largo del proceso de generación y guiar dicho proceso hacia buenos diseños.

3. El desarrollo de una metodología basada en la idea de minimizar los criterios de diseño codificados en las reglas, y usarlos en un proceso de aprendizaje automático para aprender un heurístico que determine cómo aplicar la gramática de formas.

4. La aplicación de las metodologías desarrolladas en diferentes ámbitos relacionados con la arquitectura.

A.2 Resumen de los capítulos

A.2.1 Antecedentes

El Capítulo 2 proporciona una visión general de los distintos ámbitos de investigación de esta tesis. Comienza por ofrecer una definición de diseño, como un proceso caracterizado por un *propósito*. Desde cierto punto de vista, este proceso se configura como una búsqueda en un espacio de estados (Simon, 1973).

Los ordenadores han jugado desde hace años un papel incuestionable en los procesos llevados a cabo por los profesionales del diseño. Las herramientas CAD proporcionan asistencia en numerosas tareas relacionadas con la descripción, documentación y visualización de los proyectos de diseño. Adicionalmente, mediante técnicas de inteligencia artificial, los ordenadores pueden cumplir roles más activos, como la optimización y simulación de soluciones de diseño existentes e incluso la síntesis de nuevas soluciones. La presente tesis se centra en esta última tarea, habitualmente referida en la literatura como *diseño computacional* (Cagan et al., 2005).

La concepción del diseño como una búsqueda en un espacio de posibles alternativas convierte a ciertas técnicas de inteligencia artificial en herramientas adecuadas para desarrollar metodologías de diseño computacional. Para ello, en primer lugar, es necesario un sistema generativo capaz de representar y generar de manera explícita el espacio de diseño. Algunos ejemplos de este tipo de sistemas los constituyen los autómatas celulares (propuestos por von Neumann a finales de la década de los 50), los sistemas-L (Lindenmayer, 1968), las gramáticas de grafos (Gips & Stiny, 1980) y las gramáticas de formas (Stiny & Gips, 1972). En esta tesis nos centraremos en este último formalismo dada su versatilidad y potencia. Ciertamente, las gramáticas de formas han sido utilizadas en numerosos ámbitos de diseño, siendo algunos ejemplos recientes los trabajos de Garcia & Romão (2015) (para el diseño de sillas), Palacz et al. (2015) (para estructuras tipo rejilla o *grid*) y Kielarova et al. (2015) (para diseño de joyas).

En la literatura se han explorado tres posibilidades a la hora de utilizar los distintos criterios para explorar o generar el espacio de diseño. La primera de ellas consiste en codificar dichos criterios en las reglas del sistema generativo, dando lugar a sistemas expertos que siempre producen soluciones factibles pero que habitualmente son difíciles de crear, modificar y mantener. En el contexto de las gramáticas de formas, dichos sistemas se corresponden con las tradicionales gramáticas expertas (Stiny & Mitchell, 1978; Cagdas, 1996; Flemming, 1987; Pugliese & Cagan, 2002; McCormack et al., 2004). Las otras dos posibilidades se corresponden con gramáticas más ingenuas en las que una parte o la totalidad de los criterios se emplean de manera externa. Concretamente, la segunda alternativa consiste en utilizar de manera explícita los criterios de diseño para evaluar las distintas formas producidas a lo largo del proceso de generación y guiar dicho proceso hacia buenas soluciones (Grzesiak-Kopec & Ogorzalek, 2014b; Grzesiak-Kopeć & Ogorzałek, 2014; Grzesiak-Kopec & Ogorzalek, 2014a); y la tercera consiste en utilizar los criterios para aprender un heurístico que determine cómo aplicar las reglas de la gramática (Cagan & Mitchell, 1993; Shea & Cagan, 1997, 1999b,a; Gero et al., 1994; Jin & Li, 2007; Caldas, 2008; Wu et al., 2008).

En el contexto de esta tercera alternativa, en esta tesis proponemos el uso de técnicas de aprendizaje por refuerzo (Bertsekas & Tsitsiklis, 1996; Kaelbling et al., 1996; Sutton & Barto, 1998; Wiering & van Otterlo, 2012) para aprender el heurístico o política a seguir para aplicar las reglas de gramáticas de formas ingenuas. Este tipo de aprendizaje descubre qué acciones elegir en un entorno concreto, con el objetivo de maximizar la recompensa obtenida a largo plazo. El aprendizaje sucede mediante una interacción con el entorno, recibiendo recompensas positivas o negativas tras ciertas acciones. Las recompensas pueden ser escalares o vectoriales, siendo las primeras las que han sido tratadas tradicionalmente en el ámbito del aprendizaje por refuerzo. Las alternativas para manejar recompensas vectoriales han sido englobadas bajo la disciplina denominada *aprendizaje por refuerzo multi-objetivo*. Según Roijers et al. (2013), los enfoques multi-objetivo se dividen en: (1) aquellos que suponen la existencia de una función de escalarización que permite expresar valores vectoriales como una utilidad escalar, y (2) aquellos que no suponen dicha función y por tanto asumen que la solución al problema está configurada por una frontera de Pareto.

A.2.2 Control, síntesis y criterios

En el Capítulo 3 proponemos un esquema de clasificación para sistemas de diseño computacional basados en un sistema generativo particular. En nuestro caso, este sistema se corresponde con el formalismo de las gramáticas de formas.

En el capítulo anterior describimos distintas alternativas para combinar criterios de diseño con sistemas de diseño generativo. Los criterios pueden ser: (1) codificados en su totalidad dentro de las reglas del sistema generativo, (2) usados explícitamente para evaluar soluciones potenciales y efectuar una búsqueda sistemática en el espacio de diseño, o (3) usados para descubrir un heurístico que determine cómo aplicar las reglas del sistema generativo. En el contexto de las gramáticas de formas, cada posibilidad se asocia a un tipo particular de reglas: la codificación de los criterios en las reglas da lugar a las denominadas gramáticas expertas, mientras que el uso explícito o implícito de los criterios en procesos de control externos a las reglas da lugar a gramáticas ingenuas. Sin embargo, esta separación entre gramáticas expertas e ingenuas no es nítida, dado que podemos establecer diferentes grados de ingenuidad: cuantos más criterios de diseño sean considerados externamente a las reglas, menos factibles serán las formas generadas por la gramática, y por tanto más ingenuas podrán ser consideradas. Con respecto a los procesos de control susceptibles de ser combinados con sistemas generativos, el amplio espectro de técnicas descrito en el capítulo anterior (desde procesos directos de búsqueda hasta técnicas más complejas como el recocido simulado o los algoritmos evolutivos) también sugiere una clasificación basada en la complejidad de dichos procesos.

Así, proponemos el uso de las dos dimensiones definidas en el párrafo anterior para clasificar las distintas aproximaciones al diseño computacional basadas en gramáticas de formas, es decir: la ingenuidad de las gramáticas involucradas y la complejidad del proceso de control elegido. Podemos representar este esquema de clasificación gráficamente, con la ayuda de dos ejes perpendiculares que dan lugar a cuatro secciones, tal y como se muestra en la Figura A.1. En esta figura, el eje horizontal representa la ingenuidad de las gramáticas de formas involucradas en el sistema. El extremo izquierdo representa a las gramáticas completamente ingenuas, es decir, diseñadas sin tener en cuenta los criterios de diseño. Cuanto más expertas sean las gramáticas, más a la derecha estarán situadas, siendo el extremo derecho el representante de las gramáticas de formas puramente expertas. El eje vertical representa la complejidad del método de control usado en combinación con las gramáticas de formas, siendo el extremo inferior representativo de la no existencia de tal método, y el extremo superior de mecanismos de control complejos.

Así, este esquema de clasificación da lugar a cuatro clases: (1) gramáticas expertas sin control externo, (2) gramáticas expertas combinadas con un mecanismo de control complejo, (3) gramáticas ingenuas con un mecanismo de control complejo y (4) gramáticas ingenuas sin control externo. La primera clase se corresponde con las gramáticas de formas tradicionales. Las clases segunda y tercera comprenden las principales contribuciones de esta tesis. La cuarta clase representa otra técnica de diseño computacional desarrollada en el contexto de esta tesis, con el objetivo de estudiar cómo un grupo de estudiantes de arquitectura responde a herramientas en las que la aleatoriedad está presente.

A.2.3 Un Sistema de Diseño Computacional para Manejar Criterios en Problemas de Diseño

En este capítulo describimos un sistema de diseño computacional basado en la idea de codificar algunos de los criterios de diseño en las reglas, manejando otros de manera explícita en un proceso de búsqueda basado en predicados lógicos (Ruiz-Montiel et al., 2011b, 2014). Respecto al sistema de clasificación introducido en el capítulo anterior, este sistema quedaría representado por la clase (2), es decir, la correspon-

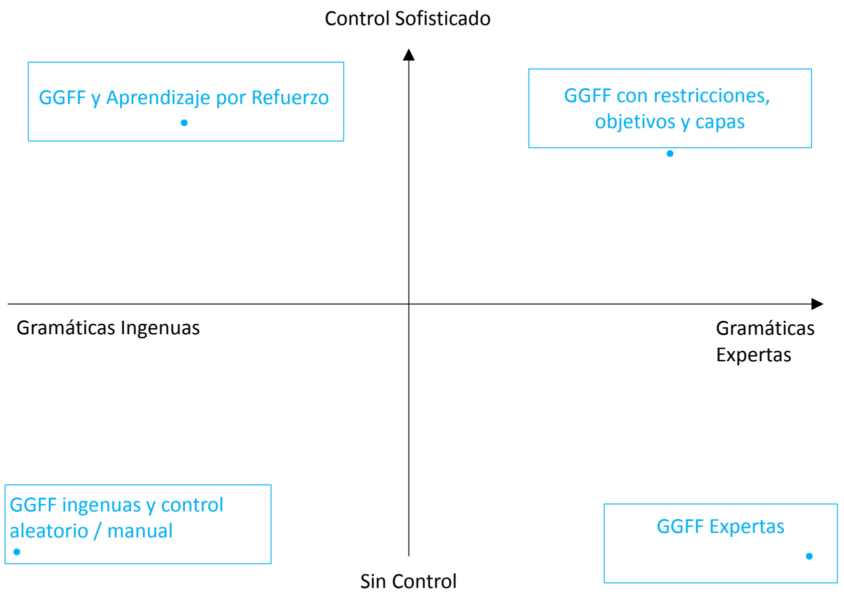

Control Sofisticado
GGFF y Aprendizaje por Refuerzo
GGFF con restricciones, objetivos y capas
Gramáticas Ingenuas
Gramáticas Expertas
GGFF ingenuas y control aleatorio / manual
GGFF Expertas
Sin Control

Figure A.1: Esquema de clasificación para sistemas de diseño computacional basados en gramáticas de formas

diente a gramáticas expertas (dado que se siguen codificando parte de los criterios de diseño en las reglas) con un mecanismo de control complejo. El proceso de búsqueda emplea predicados para evaluar las formas producidas en el proceso de generación. Un predicado puede ser una restricción o un objetivo. El proceso de búsqueda descartará las formas que violen alguna restricción, y devolverá una solución cuando encuentre una forma que satisfaga todos los objetivos de diseño.

Esta separación entre criterios definidos en las reglas y criterios utilizados como predicados obedece a dos razones. En primer lugar, en los problemas de diseño podemos encontrar requisitos de diversa naturaleza, quedando claro que algunos de estos requisitos son más fáciles de especificar por medio de su codificación en reglas visuales, mientras que otros se adaptan mejor a mecanismos simbólicos. El diseñador debería poder expresar los criterios de diseño tanto en formato visual, en forma de reglas de formas, como explícitamente, en forma de restricciones y objetivos de diseño. En segundo lugar, dado que la cantidad de criterios de diseño que se consideran en las reglas se reduce, las gramáticas de formas son más fáciles de crear y mantener, y pueden producir soluciones más diversas que una gramática de formas puramente experta.

Adicionalmente, el sistema permite el uso de capas para diseñar y visualizar las reglas. Este mecanismo ayuda a los diseñadores en el proceso de creación de reglas, dado que pueden beneficiarse de las ventajas relacionadas con las capas ofrecidas por las herramientas CAD. Una ventaja adicional es que las capas reducen el tiempo de ejecución del algoritmo de reconocimiento de subformas utilizado en las gramáticas de formas, dado que podemos tener varias formas simples separadas en capas distintas en lugar de una forma compleja dentro de una única capa.

Este sistema ha sido implementado en una herramienta software que permite a los usuarios la edición y ejecución de gramáticas de formas con restricciones, objetivos y capas. Además, ha sido puesto en práctica en dos ejemplos: (1) el diseño de viviendas configuradas según el módulo Hele propuesto por el arquitecto Rafael Leoz, y (2) restaurantes virtuales que pueden ser utilizados como escenarios para determinados videojuegos.

A.2.4 Técnicas Computacionales para Aprender y Detectar Criterios en Problemas de Diseño

En este capítulo describimos en detalle el sistema introducido en el capítulo A.2.2, basado en aprender un heurístico que determine cómo aplicar las reglas de un sistema ingenuo. En este enfoque, la cantidad de criterios de diseño codificados en las reglas es minimizada, de modo que la creación y modificación de tales reglas es fácil. Sin embargo, una ejecución a ciegas de estas reglas ingenuas conduce a soluciones no factibles. Aquí proponemos la combinación de gramáticas de formas ingenuas con técnicas de aprendizaje por refuerzo que codifiquen los criterios de diseño como recompensas que se obtienen durante el proceso de generación, y que descubran heurísticos que determinen cómo aplicar las reglas de modo que tales recompensas sean maximizadas y por tanto se produzcan soluciones factibles (Ruiz-Montiel et al., 2011a,b, 2013a, 2015). Respecto al sistema de clasificación introducido en el capítulo A.2.2, este sistema quedaría representado por la clase (3), es decir, la correspondiente a gramáticas ingenuas con un mecanismo de control complejo.

Dependiendo de la manera de tratar la naturaleza multi-criterio de los problemas de diseño, nuestra propuesta se divide en: (1) la aplicación de aprendizaje por refuerzo multi-objetivo escalarizado y (2) la aplicación de aprendizaje por refuerzo multi-objetivo basado en la premisa de que no existe una función de escalarización que permita expresar las preferencias, y por tanto el algoritmo ha de proporcionar un conjunto de políticas que se corresponda con un conjunto de soluciones Pareto-óptimas.

En la primera propuesta, utilizamos la técnica $Q(\lambda)$ (Sutton & Barto, 1998) junto con técnicas de generalización, empleadas para representar la información aprendida mediante una función en lugar de una tabla, lo cual permite reducir las necesidades de tiempo y memoria del sistema. Esta metodología es puesta en práctica en dos casos de aplicación relacionados con la arquitectura: el diseño de viviendas habitables (Ruiz-Montiel et al., 2011a, 2013a) y el diseño de viviendas energéticamente eficientes

(Ruiz-Montiel et al., 2015; Hidalgo et al., 2015). Adicionalmente, el primer caso de aplicación ha sido validado en un entorno educativo, en el cual un grupo de alumnos utiliza la metodología para obtener un conjunto de puntos de partida para sus proyectos de vivienda (Millán et al., 2014).

En la segunda propuesta, proponemos una nueva técnica, denominada PQ-learning, que aprende todas las políticas que dan lugar a soluciones Pareto-óptimas (Ruiz-Montiel et al., 2013a; Ruiz-Montiel, 2013). Dicha técnica es una extensión directa del algoritmo Q-learning (Sutton & Barto, 1998). El nuevo algoritmo es aplicado a un problema geométrico con dos criterios de diseño.

A.3 Conclusiones y Trabajo Futuro

Los problemas de diseño suelen involucrar la consideración de criterios de diferente naturaleza, incluyendo necesidades técnicas, económicas, sociales y medioambientales, entre otras. Las herramientas CAD tradicionales ayudan a los diseñadores en la representación, modificación, análisis, documentación y evaluación de sus diseños. Sin embargo, los ordenadores pueden cumplir un papel más complejo: el del diseño computacional, consistente en la síntesis de nuevas soluciones de diseño. Esta tesis se ha enfocado en este rol no trivial, siendo su objetivo general el desarrollo de nuevas técnicas de diseño computacional capaces de considerar criterios de diseño.

Algunos aspectos del proceso de diseño pueden entenderse como una búsqueda o exploración en un espacio de alternativas de diseño. Esta perspectiva facilita la explotación de técnicas computacionales para implementar métodos que sinteticen soluciones de acuerdo al propósito de un problema de diseño dado.

Para el desarrollo de metodologías de diseño computacional, es necesario un sistema generativo capaz de representar y generar el espacio de diseño. En esta tesis hemos considerado el formalismo de las gramáticas de formas dado su uso intensivo en la literatura de diseño computacional y dada también su versatilidad. El enfoque tradicional a la hora de usar gramáticas de formas consiste en codificar el conjunto completo de criterios de diseño en las mismas reglas, dando lugar a gramáticas expertas que son difíciles de crear, modificar y mantener. Este tipo de gramáticas también promueve soluciones previsibles, dado que las reglas han sido creadas con el conocimiento previo de los requisitos que las formas han de cumplir. En esta tesis hemos considerado otra alternativa, que consiste en reducir o minimizar el número de criterios codificados en las reglas. Así, tratamos con gramáticas más ingenuas que no pueden producir soluciones factibles por ellas mismas, luego necesitan de un mecanismo de control que guíe la derivación hacia buenos diseños. Concretamente, hemos utilizado algoritmos de búsqueda y métodos de aprendizaje por refuerzo para llevar a cabo dicho control.

Las principales conclusiones de esta tesis pueden ser resumidas como sigue:

1. **Se ha propuesto un esquema de clasificación para posibles enfoques al diseño computacional basados en gramáticas de formas.** Las distintas clases dependen de la naturaleza de las reglas involucradas y de los métodos de control utilizados. Concretamente, hemos propuesto un esquema en que se consideran dos aspectos para clasificar técnicas de diseño computacional basadas en gramáticas de formas. El primer aspecto considera la cantidad de criterios de diseño codificados en las reglas, siendo las gramáticas puramente expertas aquellas en las que la totalidad de los criterios han sido codificados de esta manera. Cuantos menos criterios sean codificados en las reglas, más ingenua puede ser considerada la gramática. El segundo aspecto considera la complejidad del método de control empleado, desde sistemas que carecen de dicho sistema de control hasta sistemas que emplean mecanismos complejos como métodos de búsqueda o algoritmos de aprendizaje por refuerzo. Estos dos aspectos dan lugar a las siguientes cuatro clases: (1) gramáticas expertas sin un mecanismo de control, (2) gramáticas expertas combinadas con un mecanismo complejo de control, (3) gramáticas ingenuas combinadas con un mecanismo complejo de control, y (4) gramáticas ingenuas sin un mecanismo de control. La primera clase representa a las gramáticas de formas tradicionales, utilizadas en numerosas áreas. Las clases segunda y tercera comprenden las principales contribuciones de esta tesis. La cuarta clase también ha sido considerada en el contexto de esta tesis, con el objetivo de estudiar cómo reaccionan los estudiantes de arquitectura ante herramientas de diseño computacional en las que la aleatoriedad está presente.

2. **Se ha desarrollado una metodología de diseño computacional basada en gramáticas de formas expertas y un mecanismo de control complejo.** Concretamente, esta metodología está basada en la idea de codificar algunos requisitos de diseño en las reglas y utilizar el resto de manera explícita para evaluar las formas producidas a lo largo del proceso de generación, guiando dicho proceso hacia buenos diseños. Las gramáticas de formas involucradas son por tanto menos expertas que en el enfoque tradicional en el cual se codifica la totalidad de los criterios de diseño en las reglas. En esta configuración distinguimos entre criterios que son más fáciles de especificar geométricamente y criterios que son más fáciles de expresar como predicados lógicos. Los primeros son codificados dentro de reglas con capas, las cuales pueden ayudar en la organización y la representación de las formas, así como reducir el tiempo de derivación de las mismas. Los segundos son programados como restricciones y objetivos de diseño, los

cuales son usados para guiar un algoritmo de búsqueda primero-en-profundidad.

3. **Se ha desarrollado una herramienta software (ShaDe) para editar y ejecutar gramáticas de formas con capas, restricciones y objetivos.** ShaDe permite la creación de gramáticas de formas genéricas que pueden ser organizadas en capas. La herramienta también ofrece un conjunto predefinido de restricciones y objetivos de diseño, asi como medios para personalizarlos e incluso crear nuevos predicados. Las gramáticas de formas pueden ser ejecutadas en ShaDe para producir soluciones que cumplan los criterios y objetivos de diseño especificados.

4. **Se ha desarrollado una metodología de diseño computacional basada en gramáticas de formas ingenuas y un mecanismo de control complejo.** Concretamente, esta metodología está basada en la idea de usar el conjunto completo de criterios de diseño como recompensas en un proceso de aprendizaje por refuerzo, con el objetivo de aprender un heurístico (o conjunto de heurísticos) que determine cómo aplicar las reglas del sistema generativo. En el proceso de generación, únicamente tenemos en cuenta la información aprendida para aplicar las reglas, de modo que los criterios de diseño se utilizan de manera implícita. Se han presentado dos alternativas para aprender las políticas de aplicación de reglas, dependiendo de la manera de tratar la naturaleza multi-objetivo de los problemas de diseño. En la primera, las recompensas son escalarizada de acuerdo a un conjunto de preferencias de diseño que se conocen a priori, de modo que el algoritmo averigua una única política para una gramática de formas dada. La segunda alternativa no escalariza las recompensas, luego el algoritmo aprende múltiples políticas que pueden ser utilizadas para producir un conjunto de soluciones Pareto-óptimas.

5. **Se ha propuesto un nuevo algoritmo de aprendizaje por refuerzo multi-objetivo (PQ-learning).** En el contexto de la metodología en la que no se escalarizan las recompensas, hemos propuesto una nueva técnica de aprendizaje por refuerzo, basada en una extensión directa del algoritmo Q-learning, que trabaja con recompensas vectoriales. Este nuevo método ha sido probado en dos problemas pertenecientes a un *benchmark* de aprendizaje por refuerzo multi-objetivo.

6. **Las metodologías desarrolladas han sido puestas en práctica en diferentes escenarios relacionados con la arquitectura.** Concretamente, la técnica basada en gramáticas de formas con capas, restricciones y objetivos ha sido utilizada para sintetizar dos tipos de diseños: (1) viviendas configuradas de acuerdo al módulo Hele propuesto por el arquitecto Rafael Leoz, y (2) restauran-

tes virtuales que pueden ser usados como escenarios en ciertos videojuegos. La técnica basada en gramáticas de formas ingenuas y aprendizaje por refuerzo ha sido puesta en práctica en dos casos de aplicación relacionados con la generación de viviendas habitables y energéticamente eficientes.

7. **Se han llevado a cabo dos estudios empíricos con estudiantes de arquitectura.** Particularmente, fueron asociados con las metodologías correspondientes a gramáticas de formas ingenuas con y sin control. El experimento relativo a la metodología de las gramáticas ingenuas sin control fue llevado a cabo para estudiar cómo los estudiantes de arquitectura reaccionan ante una herramienta de diseño computacional donde se permiten comandos aleatorios. La principal conclusión fue que los estudiantes están dispuestos a explorar la aleatoriedad, de modo que las herramientas de diseño computacional deberían tenerla en cuenta; sin embargo, también deberían proporcionar medios para controlar dicha aleatoriedad. El experimento asociado a la metodología de gramáticas de formas ingenuas con control fue llevado a cabo para validar dicha metodología al ser utilizada para generar esquemas de viviendas habitables, siendo la principal conclusión que la herramienta puede proporcionar puntos de partida interesantes para los proyectos arquitectónicos.

A continuación detallamos los aspectos de esta tesis que merecen una investigación más profunda:

- La extensión de las metodologías propuestas a tipos más complejos de gramáticas de formas, tales como gramáticas tridimensionales o incluso paramétricas, en las cuales las formas se definen no sólo a partir de la disposición de distintos elementos geométricos sino también a partir de parámetros variables, tales como la anchura y altura de un rectángulo.

- Los casos de aplicación que involucran gramáticas ingenuas y aprendizaje por refuerzo escalarizado se basan en una división en fases del problema de diseño considerado. Esta separación del proceso de diseño en fases secuenciales sirve para reducir el conjunto de criterios que han de tenerse en cuenta en cada paso. Sin embargo, también introduce una limitación importante que puede afectar al proceso de generación en el caso de problemas de diseño más complejos: el proceso de aprendizaje sólo trata con los criterios locales de cada fase, y por tanto las políticas no pueden reflejar aspectos globales. Por ejemplo, en el contexto de nuestro ejemplo de diseño de viviendas, una política globalmente óptima daría pistas acerca de cómo ubicar la cocina teniendo en cuenta la futura ubicación de los ámbitos no especializados. El problema de no dividir el proceso en fases es la necesidad de considerar el conjunto completo de criterios de diseño en una única fase. Una posibilidad para afrontar dicho problema es la integración con técnicas más potentes como la generalización no lineal ofrecida por las redes neuronales.

- Hemos mostrado cómo la metodología basada en gramáticas ingenuas y PQ-learning puede ser utilizada para abordar problemas geométricos, pero es necesaria más investigación para aplicar dicha metodología en escenarios reales. Creemos que esto puede conseguirse por medio de la integración de PQ-learning con técnicas de generalización.

- Finalmente, la aplicación de las metodologías propuestas a otros ámbitos de diseño es también una línea importante de investigación.

Bibliography

Ábalos, I. (2008). *La belleza termodinámica*. Circo.

Ábalos, I. (2012). *Thermodynamics applied to highrise and mixed use prototypes: a studio research report of the Harvard Graduate School of Design*. Harvard Graduate School of Design.

Abbeel, P., Coates, A., Quigley, M., & Ng, A. Y. (2007). An application of reinforcement learning to aerobatic helicopter flight. In *In Advances in Neural Information Processing Systems 19* (pp. 2007).: MIT Press.

Abounadi, J., Bertsekas, D. P., & Borkar, V. (2000). *Stochastic Approximation for Non-Expansive Maps: Application to Q-Learning Algorithms*. Technical report, SIAM Journal on Control and Optimization.

Agarwal, M., Cagan, J., & Constantine, K. G. (1999). Influencing generative design through continuous evaluation: Associating costs with the coffeemaker shape grammar. *Artificial Intelligence in Engineering Design, Analysis and Manufacturing*, 13, 253–275.

Ang, M. C., Chau, H. H., Mckay, A., & de Pennington, A. (2006). Combining evolutionary algorithms and shape grammars to generate branded product design. In J. S. Gero (Ed.), *Design Computing and Cognition '06* chapter 27, (pp. 521–539). Dordrecht: Springer Netherlands.

Asada, M., Noda, S., Tawaratsumida, S., & Hosoda, K. (1996). Purposive behavior acquisition for a real robot by vision-based reinforcement learning. *Machine Learning*, 23, 279–303. 10.1023/A:1018237008823.

Aydin, M. & Oztemel, E. (2000). Dynamic job-shop scheduling using reinforcement learning agents. *Robotics and Autonomous Systems*, 33(2-3), 169 – 178.

Balachandran, M. & Gero, J. S. (1984). A comparison of three methods for generating the Pareto optimal set. *Engineering Optimization*, 7(4), 319–336.

Barrett, L. & Narayanan, S. (2008). Learning all optimal policies with multiple criteria. ICML '08 (pp. 41–47).: ACM.

Baupanel (2015). `http://www.baupanel.com/en`.

Baxter, J., Tridgell, A., & Weaver, L. (2000). Learning to play chess using temporal differences. *Machine Learning*, 40, 243–263. 10.1023/A:1007634325138.

Behling, S., B. T. (2012). *Sol Power. La evolución de la arquitectura sostenible*. Gustavo Gili.

Beirão, J. N., Duarte, J. P., & Stouffs, R. (2011). Creating specific grammars with generic grammars: towards flexible urban design. *Nexus Network Journal*, 13(1), 73–111.

Belmonte, M.-V., Millán, E., Ruiz-Montiel, M., Badillo, R., Boned, J., Mandow, L., & de-la Cruz, J.-L. P. (2014). Randomness and control in design processes: An empirical study with architecture students. *Design Studies*, 35(4), 392 – 411.

Bertsekas, D. P. & Tsitsiklis, J. N. (1996). *Neuro-Dynamic Programming*. Athena Scientific.

Bolognini, F., Seshia, A. A., & Shea, A. K. (2007). A computational design synthesis method for mems using comsol. In *COMSOL Users Conference*.

Boutilier, C., Dean, T., & Hanks, S. (1999). Decision-theoretic planning: Structural assumptions and computational leverage. *Journal of Artificial Intelligence Research*, 11(1), 94.

Brown, K. N., McMahon, C. A., & Williams, J. H. S. (1996). Describing process plans as the formal semantics of a language of shape. *Advanced Engineering Informatics*, 10(2), 153–169.

Buchli, J., Stulp, F., Theodorou, E., & Schaal, S. (2011). Learning variable impedance control: A reinforcement learning approach. *Int. J. Rob. Res.*, 30(7), 820–833.

Cagan, J., Campbell, M. I., Finger, S., & Tomiyama, T. (2005). A framework for computational design synthesis: model and applications. *Journal of Computing and Information Science in Engineering*, 5(3), 171–181.

Cagan, J. & Mitchell, W. J. (1993). Optimally directed shape generation by shape annealing. *Environment and Planning B: Planning and Design*, 20(1), 5–12.

Cagdas, G. (1996). A shape grammar model for designing row-houses. *Design Studies*, 17, 35–51.

Caldas, L. (2008). Generation of energy-efficient architecture solutions applying gene_arch: An evolution-based generative design system. *Advanced Engineering Informatics*, 22(1), 59–70.

Campbell, M. I. & Shea, K. (2014). Computational design synthesis.

Castelletti, A., Corani, G., Rizzolli, A., Soncini-Sessa, R., & Weber, E. (2002). Reinforcement learning in the operational management of a water system. In *IFAC Workshop on Modeling and Control in Environmental Issues* (pp. 325–330).

Chase, S. C. (2002). A model for user interaction in grammar-based design systems. *Automation in Construction*, 11, 161–172.

Chau, H. H., Chen, X., McKay, A., & de Pennington, A. (2004). Evaluation of a 3D shape grammar implementation. In *Design Computing and Cognition.* (2004) 357–376.

Chouchoulas, O. (2003). *Shape Evolution. An Algorithmic Method for Conceptual Architectural Design Combining Shape Grammars and Genetic Algorithms.* PhD thesis, University of Bath.

Conway, J. (1970). The game of life. *Scientific American*, 223(4), 4.

Corbin, J. & Strauss, A. (2008). *Basics of qualitative research : Grounded Theory procedures and techniques.* Thousand Oaks, CA: SAGE Publications, 3rd. ed. edition.

Correia, R., Duarte, J., & Leitao, A. (2012). Gramatica: A general 3d shape grammar interpreter targeting the mass customization of housing. In *Proceedings of the 30th eCAADe Conference - Volume 1* (pp. 489–496).

Droste, S. & Furnkranz, J. (2008). Learning the piece values for three chess variants. *International Computer Games Association Journal*, (pp. 209–233).

Duarte, J. P. (2005). A discursive grammar for customizing mass housing: the case of Siza's houses at Malagueira. *Automation in Construction*, 14, 265–275.

Duarte, J. P., Beirão, J. N., Montenegro, N., & Gil, J. (2012). City induction: A model for formulating, generating, and evaluating urban designs. In S. M. Arisona, G. Aschwanden, J. Halatsch, & P. Wonka (Eds.), *Digital Urban Modeling and Simulation*, volume 242 of *Communications in Computer and Information Science* (pp. 73–98). Springer Berlin Heidelberg.

Eastman, C. M. (1969). Cognitive processes and ill-defined problems: a case study from design. In *International Joint Conferences on Artificial Intelligence '69* (pp. 669–690).

Eckert, C., Stacey, M., & Clarkson, P. (2010). Algorithms and inspirations: Creative reuse of design experience. In *Proceedings of the Greenwich 2000 Symposium: Digital Creativity* (pp. 1–10).

Eggert, R. (2004). *Engineering design*. Pearson/Prentice Hall.

Endo, G., Morimoto, J., Matsubara, T., Nakanishi, J., & Cheng, G. (2008). Learning cpg-based biped locomotion with a policy gradient method: Application to a humanoid robot. *Int. J. Rob. Res.*, 27(2), 213–228.

Enerbuilding (2008). Enerbuilding: Eficiencia energética en viviendas.

Erden, M. S. & Leblebicioglu, K. (2008). Free gait generation with reinforcement learning for a six-legged robot. *Robotics and Autonomous Systems*, 56(3), 199 – 212.

Flemming, U. (1987). More than the sum of parts: the grammar of queen anne houses. *Environment and planning B*, 14, 323–350.

Garcia, S. & Romão, L. (2015). A design tool for generic multipurpose chair design. In G. Celani, D. M. Sperling, & J. M. S. Franco (Eds.), *Computer-Aided Architectural Design Futures. The Next City - New Technologies and the Future of the Built Environment*, volume 527 of *Communications in Computer and Information Science* (pp. 600–619). Springer Berlin Heidelberg.

Gaskett, C., Fletcher, L., & Zelinsky, A. (2000). Reinforcement learning for a vision based mobile robot. In *Intelligent Robots and Systems, 2000. (IROS 2000). Proceedings. 2000 IEEE/RSJ International Conference on*, volume 1 (pp. 403 –409 vol.1).

Gelly, S. & Silver, D. (2008). Achieving master level play in 9x9 computer go. In *Proceedings of the 23rd national conference on Artificial intelligence - Volume 3*, AAAI'08 (pp. 1537–1540).: AAAI Press.

Gero, J. S. & Kazakov, V. A. (1996). Evolving building blocks for design using genetic engineering: A formal approach. In *Advances in formal design mehotds for CAD* (pp. 31–50).: Hall.

Gero, J. S., Louis, S. J., & Kundu, S. (1994). Evolutionary learning of novel grammars for design improvement. *Artificial Intelligence for Engineering Design, Analysis and Manufacturing*, 8(02), 83–94.

Gips, J. (1999). Computer implementation of shape grammars. In *Proceedings of the Workshop of Shape Computation*: MIT.

Gips, J. & Stiny, G. (1980). Production systems and grammars: a uniform characterization. *Environment and Planning B*, 7(4), 399–408.

Glaser, B. G. & Strauss, A. L. (1967). *The discovery of Grounded Theory*. Chicago, IL: Aldine.

Goel, V. (1994). A comparison of design and nondesign problem spaces. *Artificial Intelligence in Engineering*, 9(1), 53–72.

Goel, V. & Pirolli, P. (1992). The structure of design problem spaces. *Cognitive science*, 16(3), 395–429.

Goldschmidt, G. (1994). On visual design thinking: the vis kids of architecture. *Design Studies*, 15(2), 158–174.

Grabska, E., Lachwa, A., & Slusarczyk, G. (2012). New visual languages supporting design of multi-storey buildings. Advanced Engineering Informatics, On press.

Granadeiro, V., Duarte, J. P., ao R. Correia, J., & Leal, V. M. (2013). Building envelope shape design in early stages of the design process: Integrating architectural design systems and energy simulation. *Automation in Construction*, 32, 196 – 209.

Grasl, T. & Economou, A. (2013). From topologies to shapes: parametric shape grammars implemented by graphs. *Environment and Planning B: Planning and Design*, 40(5), 905–922.

Grzesiak-Kopeć, K. & Ogorzałek, M. (2014). Emergent phenomena in constrained 3d layout design. In *Artificial Intelligence and Soft Computing* (pp. 479–489).: Springer.

Grzesiak-Kopec, K. & Ogorzalek, M. (2014a). New approach to block-level 3d ic layout design. In *Circuits and Systems (LASCAS), 2014 IEEE 5th Latin American Symposium on* (pp. 1–4).: IEEE.

Grzesiak-Kopec, K. & Ogorzalek, M. J. (2014b). Computer-aided 3d ics layout design. *Computer-Aided Design and Applications*, 11(3), 318–325.

Handa, H. (2009). Solving multi-objective reinforcement learning problems by eda-rl - acquisition of various strategies. In *Intelligent Systems Design and Applications, 2009. ISDA '09. Ninth International Conference on* (pp. 426–431).

Hanna, R. & Barber, T. (2001). An inquiry into computers in design: attitudes before - attitudes after. *Design Studies*, 22(3), 255 – 281.

Helms, B. & Shea, K. (2012). Computational synthesis of product architectures based on object-oriented graph grammars. *Journal of Mechanical Design*, 134, 021008.

Herr, C. M. & Kvan, T. (2007). Adapting cellular automata to support the architectural design process. *Automation in Construction*, 16(1), 61–69.

Hidalgo, P., Ruiz-Montiel, M., & Belmonte, D. (2015). *Generador de esquemas de viviendas unifamiliares energéticamente eficientes, https://http://www.lcc.uma.es/~perez/ntidapa/TR_EE.pdf*. Technical report, University of M'alaga.

Hiraoka, K., Yoshida, M., & Mishima, T. (2008). Parallel reinforcement learning for weighted multi-criteria model with adaptive margin. In M. Ishikawa, K. Doya, H. Miyamoto, & T. Yamakawa (Eds.), *Neural Information Processing*, volume 4984 of *Lecture Notes in Computer Science* (pp. 487–496).

Hoisl, F. & Shea, K. (2011). An interactive, visual approach to developing and applying parametric three-dimensional spatial grammars. *Artificial Intelligence for Engineering Design, Analysis and Manufacturing*, 25(Special Issue 04), 333–356.

Howard, R. A. (1960). *Dynamic programming and Markov processes*. Cambridge: Technology Press of Massachusetts Institute of Technology.

Hu, J. & Wellman, M. P. (2003). Nash q-learning for general-sum stochastic games. *J. Mach. Learn. Res.*, 4, 1039–1069.

Istepanian, R. S. H., Philip, N. Y., & Martini, M. G. (2009). Medical qos provision based on reinforcement learning in ultrasound streaming over 3.5g wireless systems. *IEEE J.Sel. A. Commun.*, 27(4), 566–574.

Jamieson, S. (2004). Likert scales: how to (ab)use them. *Medical Education*, 28(12), 1212–1218.

Jin, Y. & Li, W. (2007). Design concept generation: a hierarchical coevolutionary approach. *Journal of Mechanical Design*, 129(10), 1012–1022.

Kaelbling, L. P., Littman, M. L., & Moore, A. W. (1996). Reinforcement learning: A survey. *Journal of Artificial Intelligence Research*, 4, 237–285.

kang Li, A. I., Chau, H. H., Chen, L., & Wang, Y. (2009). A system for developing two- and three-dimensional shape grammars. In *Proceedings of the 14th Conference of Computer-Aided Architectural Design Research in Asia (CAADRIA) (2009)* (pp. 717–726).

Kielarova, S., Pradujphongphet, P., & Bohez, E. (2015). New interactive-generative design system: Hybrid of shape grammar and evolutionary design - an application of jewelry design. In Y. Tan, Y. Shi, F. Buarque, A. Gelbukh, S. Das, & A. Engelbrecht (Eds.), *Advances in Swarm and Computational Intelligence*, volume 9140 of *Lecture Notes in Computer Science* (pp. 302–313). Springer International Publishing.

Kirkpatrick, S., Gelatt, C. D., & Vecchi, M. P. (1983). Optimization by simulated annealing. *Science, Number 4598, 13 May 1983*, 220, 4598, 671–680.

Knight, T. (2015). Shapes and other things. *Nexus Network Journal*, (pp. 1–18).

Knight, T. W. (1999). *Applications in architectural design, and education and practice.* Technical report, NSF/MIT Workshop on Shape Computation.

Kotsopoulos, S. D. (2005). *Constructing Design Concepts A Computational Approach to the Synthesis of Architectural Form.* PhD thesis, Department of Architecture and Urban Planning, Massachusetts Institute of Technology.

Kraft, B. & Nagl, M. (2007). Visual knowledge specification for conceptual design: Definition and tool support. *Advanced Engineering Informatics*, 21(1), 67 – 83.

Krishnamurti, R. (1980). The arithmetic of shapes. *Environment and Planning B*, 7(4), 463–484.

Krishnamurti, R. (1981). The construction of shapes. *Environment and Planning B*, 8(1), 5–40.

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.

Lee, H. C. & Tang, M. X. (2009a). Evolving product form design using parametric shape grammars integrated with genetic programming. *Artificial Intelligence in Engineering Design, Analysis and Manufacturing*, 23, 131–158.

Lee, H. C. & Tang, M. X. (2009b). Evolving product form designs using parametric shape grammars integrated with genetic programming. *Artificial Intelligence for Engineering Design, Analysis and Manufacturing*, 23(02), 131–158.

Leoz, R. (1969). *Redes y Ritmos Espaciales.* Blume.

Leoz, R. (1978). *Rafael Leoz: exposición homenaje: abril, mayo, junio 1978.* Ministerio de Cultura (España).

Levinson, R. & Weber, R. (2001). Chess neighborhoods, function combination, and reinforcement learning. In T. Marsland & I. Frank (Eds.), *Computers and Games*, volume 2063 of *Lecture Notes in Computer Science* (pp. 133–150). Springer Berlin / Heidelberg.

Lilith, N. & Dogancay, K. (2004). Dynamic channel allocation for mobile cellular traffic using reduced-state reinforcement learning. In *Wireless Communications and Networking Conference, 2004. WCNC. 2004 IEEE*, volume 4 (pp. 2195 – 2200 Vol.4).

Lin, L., Wu, T., Porway, J., & Xu, Z. (2009). A stochastic graph grammar for compositional object representation and recognition. *Pattern Recognition*, 42(7), 1297–1307.

Lindenmayer, A. (1968). Mathematical models for cellular interactions in development i. filaments with one-sided inputs. *Journal of theoretical biology*, 18(3), 280–299.

Mandow, L. & De La Cruz, J. L. P. (2000). The role of multicriteria problem solving in design. In *Artificial Intelligence in Design Ó0* (pp. 23–41). Springer.

Mataric, M. J. (1997). Reinforcement learning in the multi-robot domain. *Autonomous Robots*, 4, 73–83. 10.1023/A:1008819414322.

McCormack, J. P., Cagan, J., & Vogel, C. M. (2004). Speaking the Buick language: capturing, understanding and exploring brand identity with shape grammars. *Design Studies*, 25, 1–29.

McKay, A., Chase, S., Shea, K., & Chau, H. H. (2012). Spatial grammar implementation: From theory to useable software. *Artificial Intelligence for Engineering Design, Analysis and Manufacturing*, 26(Special Issue 02), 143–159.

Michels, J., Saxena, A., & Ng, A. Y. (2005). High speed obstacle avoidance using monocular vision and reinforcement learning. In *Proceedings of the 22nd international conference on Machine learning*, ICML '05 (pp. 593–600). New York, NY, USA: ACM.

Millán, E., Belmonte, M.-V., Ruiz-Montiel, M., Gavilanes, J., & de-la Cruz, J.-L. P. (2014). An intelligent tool to assist architecture students in the early stages of design. In *22nd International Conference on Computers in Education* (pp. 22–31).

Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., Graves, A., Riedmiller, M., Fidjeland, A. K., Ostrovski, G., Petersen, S., Beattie, C., Sadik, A., Antonoglou, I., King, H., Kumaran, D., Wierstra, D., Legg, S., & Hassabis, D. (2015). Human-level control through deep reinforcement learning. *Nature*, 518(7540), 529–533.

Montaner Muxí arquitectes (2008). *Propuesta de nueva normativa de viviendas.* Technical report, Dirección general de ordenación del territorio, Junta de Andalucía.

Mukai, Y., Kuroe, Y., & Iima, H. (2012). Multi-objective reinforcement learning method for acquiring all pareto optimal policies simultaneously. In *Systems, Man, and Cybernetics (SMC), 2012 IEEE International Conference on* (pp. 1917–1923).

Müller, P., Wonka, P., Haegler, S., Ulmer, A., & Van Gool, L. (2006). Procedural modeling of buildings. *ACM Trans. Graph.*, 25(3), 614–623.

Natarajan, S. & Tadepalli, P. (2005). Dynamic preferences in multi-criteria reinforcement learning. ICML ’05 (pp. 601–608).: ACM.

Nemec, B., Zorko, M., & Zlajpah, L. (2010). Learning of a ball-in-a-cup playing robot. In *Robotics in Alpe-Adria-Danube Region (RAAD), 2010 IEEE 19th International Workshop on* (pp. 297 –301).

O’Neill, M., McDermott, J., Swafford, J. M., Byrne, J., Hemberg, E., Brabazon, A., Shotton, E., McNally, C., & Hemberg, M. (2010). Evolutionary design using grammatical evolution and shape grammars: designing a shelter. *International Journal of Design Engineering*, 3(1).

Orsborn, S., Cagan, J., & Boatwright, P. (2008). A methodology for creating a statistically derived shape grammar composed of non-obvious shape chunks. *Research in Engineering Design*, 18(4), 181–196.

Pacheco, R., nez, J. O., & Martínez, G. (2012). Energy efficient design of building: A review. *Renewable and Sustainable Energy Reviews*, 16(6), 3559 – 3573.

Palacz, W., Ryszka, I., & Grabska, E. (2015). A graph grammar tool for generating computational grid structures. In L. Rutkowski, M. Korytkowski, R. Scherer, R. Tadeusiewicz, L. A. Zadeh, & J. M. Zurada (Eds.), *Artificial Intelligence and Soft Computing*, volume 9120 of *Lecture Notes in Computer Science* (pp. 436–447). Springer International Publishing.

Paletta, L. & Pinz, A. (2000). Active object recognition by view integration and reinforcement learning. *Robotics and Autonomous Systems*, 31(1-2), 71 – 86.

Palubicki, W., Horel, K., Longay, S., Runions, A., Lane, B., Měch, R., & Prusinkiewicz, P. (2009). Self-organizing tree models for image synthesis. In *ACM Transactions on Graphics (TOG)*, volume 28 (pp.$\tilde{5}$8).: ACM.

Papalambros, P. Y. & Wilde, D. J. (2000). *Principles of optimal design: modeling and computation.* Cambridge university press.

Parish, Y. I. & Müller, P. (2001). Procedural modeling of cities. In *Proceedings of the 28th annual conference on Computer graphics and interactive techniques* (pp. 301–308).: ACM.

Passivhaus (2011). Guía del estándar passivhaus - edificios de consumo energético casi nulo.

Pauwels, P., Van Deursen, D., Verstraeten, R., De Roo, J., De Meyer, R., Van De Walle, R., & Van Campenhout, J. (2011). A semantic rule checking environment for building performance checking. *Automation in Construction*, 20(5), 506–518.

Peng, J. & Bhanu, B. (1998). Closed-loop object recognition using reinforcement learning. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 20(2), 139 –154.

Peters, J., Mulling, K., Kober, J., Nguyen-Tuong, D., & Kromer, O. (2011). Towards motor skill learning for robotics. In C. Pradalier, R. Siegwart, & G. Hirzinger (Eds.), *Robotics Research*, volume 70 of *Springer Tracts in Advanced Robotics* (pp. 469–482). Springer Berlin - Heidelberg.

Petre, M., Sharp, H., & Johnson, J. (2006). Complexity through combination: an account of knitwear design. *Design Studies*, 27(2), 183–222.

Piggott, P. & Sattar, A. (1994). Reinforcement learning of iterative behaviour with multiple sensors. *Applied Intelligence*, 4, 351–365. 10.1007/BF00872474.

Pugliese, M. & Cagan, J. (2002). Capturing a rebel: modeling the Harley-Davidson brand through a motorcycle shape grammar. *Research in Engineering Design*, 13(3), 139–156.

Roijers, D. M., Vamplew, P., Whiteson, S., & Dazeley, R. (2013). A survey of multi-objective sequential decision-making. *Journal of Artificial Intelligence Research (JAIR)*, 48, 67–113.

Roof Maker (2015). `http://www.valiarchitects.com/sketchup_scripts/instant-roof`.

Rowe, P. G. (1987). *Design Thinking.* Cambridge, Mass, MIT Press.

Ruby Language (2015). `http://www.ruby-lang.org/`.

Ruiz-Montiel, M. (2012a). Shade for sketchup technical reference, `http://www.lcc.uma.es/~perez/ntidapa/aug-v2-4.pdf`.

Ruiz-Montiel, M. (2012b). Shade for sketchup user's guide, `http://www.lcc.uma.es/~perez/ntidapa/ug-v2-4.pdf`.

Ruiz-Montiel, M. (2013). Multi-objective reinforcement learning. In *European Workshop on Reinforcement Learning (EWRL 11)* (pp.Ĩ8).

Ruiz-Montiel, M., Belmonte, M.-V., Boned, J., Mandow, L., Millán, E., Badillo, A. R., & de-la Cruz, J.-L. P. (2014). Layered shape grammars. *Computer-Aided Design*, 56, 104 – 119.

Ruiz-Montiel, M., Boned, J., Gavilanes, J., Hidalgo, P., Belmonte, D., Mandow, L., & Pérez-de-la Cruz, J.-L. (2015). Proyecto arquitectónico energéticamente eficiente mediante gramáticas de formas y aprendizaje por refuerzo. In *Conferencia de la Asociación Española para la Inteligencia Artificial.* (2015).

Ruiz-Montiel, M., Boned, J., Gavilanes, J., Jiménez, E., Mandow, L., & de-la Cruz, J.-L. P. (2013a). Design with shape grammars and reinforcement learning. *Advanced Engineering Informatics*, 27(2), 230 – 245.

Ruiz-Montiel, M., Boned, J., Gavilanes, J., Jiménez, E., Mandow, L., & Pérez-de-la Cruz, J.-L. (2011a). Proyecto arquitectónico mediante gramáticas de formas sencillas y aprendizaje. In *Avances en Inteligencia Artificial: Actas de CAEPIA 11, 1.*

Ruiz-Montiel, M., Mandow, L., & Pérez-de-la Cruz, J. L. (2013b). Pq-learning: Aprendizaje por refuerzo multiobjetivo. In *Conferencia de la Asociación Española para la Inteligencia Artificial.*

Ruiz-Montiel, M., Mandow, L., Pérez-de-la Cruz, J.-L., & Gavilanes, J. (2011b). Shapes, grammars, constraints and policies. In *CEUR: SHAPES 1.0, 812.*

Schaeffer, J., Hlynka, M., & Jussila, V. (2001). Temporal difference learning applied to a high-performance game-playing program. In *Proceedings of the 17th international joint conference on Artificial intelligence - Volume 1*, IJCAI'01 (pp. 529–534). San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.

Schmidt, L., Shetty, H., & Chase, S. (2000). A graph grammar approach to mechanism synthesis. *ASME J. Mech. Des*, 122(3), 371–376.

Schmidt, L. C. & Cagan, J. (1997). Ggreada: a graph grammar-based machine design algorithm. *Research in Engineering Design*, 9(4), 195–213.

Schraudolph, N., Dayan, P., & Sejnowski, T. (2001). Learning to evaluate go positions via temporal difference methods. In N. Baba & L. Jain (Eds.), *Computational*

Intelligence in Games, volume 62 of *Studies in Fuzziness and Soft Computing* (pp. 77–98). Springer Berlin / Heidelberg.

Senouci, S.-M. & Pujoile, G. (2003). Dynamic channel assignment in cellular networks: a reinforcement learning solution. In *Telecommunications, 2003. ICT 2003. 10th International Conference on*, volume 1 (pp. 302 – 309 vol.1).

Shea, K. & Cagan, J. (1997). Innovative dome design: Applying geodesic patterns with shape annealing. *Artificial intelligence for engineering design, analysis and manufacturing*, 11(5), 379–394.

Shea, K. & Cagan, J. (1999a). The design of novel roof trusses with shape annealing: assessing the ability of a computational method in aiding structural designers with varying design intent. *Design Studies*, 20(1), 3–23.

Shea, K. & Cagan, J. (1999b). Languages and semantics of grammatical discrete structures. *Artificial Intelligence for Engineering Design, Analysis and Manufacturing*, 13(04), 241–251.

Shea, K., Ertelt, C., Gmeiner, T., & Ameri, F. (2010). Design-to-fabrication automation for the cognitive machine shop. *Advanced Engineering Informatics*, 24(3), 251–268.

Shelton, C. (2001). *Importance Sampling for Reinforcement Learning with Multiple Objectives*. AI Memo 2001-003, MIT AI Lab.

Siddique, Z. & Rosen, D. W. (1999). Product platform design: a graph grammar approach. In *Proceedings of the ASME International Design Engineering Technical Conferences and Computers and Information in Engineering Conference, Las Vegas, NV*.

Simon, H. A. (1968). *The sciences of the artificial*. Cambridge, MA.: MIT Press.

Simon, H. A. (1973). The structure of ill structured problems. *Artificial Intelligence*, 4, 181–201.

Singh, S. & Bertsekas, D. (1997). Reinforcement learning for dynamic channel allocation in cellular telephone systems. In *In Advances in Neural Information Processing Systems: Proceedings of the 1996 Conference* (pp. 974–980).: MIT Press.

Smart, W. & Pack Kaelbling, L. (2002). Effective reinforcement learning for mobile robots. In *Robotics and Automation, 2002. Proceedings. ICRA '02. IEEE International Conference on*, volume 4 (pp. 3404 – 3410 vol.4).

Stanković, T., Štorga, M., Shea, K., & Marjanović, D. (2013). Formal modelling of technical processes and technical process synthesis. *Journal of engineering design*, 24(3), 211–238.

Starling, A. C. & Shea, K. (2005). A parallel grammar for simulation-driven mechanical design synthesis. In *ASME 2005 International Design Engineering Technical Conferences and Computers and Information in Engineering Conference* (pp. 427–436).: American Society of Mechanical Engineers.

Stiny, G. (1975). *Pictorial and formal aspects of shape and shape grammars and aesthetic system*. Basel and Stuttgart: Birkhauser.

Stiny, G. (1980). Introduction to shape and shape grammars. *Environment and Planning B*, 7, 343–351.

Stiny, G. (1992). Weights. *Environment and Planning B*, 19(4), 413–430.

Stiny, G. (2006). *Shape. Talking about seeing and doing*. Cambridge, Ma.: MIT Press.

Stiny, G. & Gips, J. (1972). Shape grammars and the generative specification of painting and sculpture. In *Information Processing 71* (pp. 1460–1465). North-Holland.

Stiny, G. & Mitchell, W. J. (1978). The palladian grammar. *Environment and planning B*, 5, 5–18.

Stouffs, R. (2006). Design spaces: The explicit representation of spaces of alternatives. *AIE EDAM: Artificial Intelligence for Engineering Design, Analysis, and Manufacturing*, 20(02), 61–62.

Studer, R., Benjamins, V. R., & Fensel, D. (1998). Knowledge engineering: Principles and methods. *Data and Knowledge Engineering*, 25, 161–197.

Sutton, R. S. & Barto, A. G. (1998). *Reinforcement learning: an introduction*. Cambridge, Ma.: MIT Press.

Szita, I., Chaslot, G., & Spronck, P. (2010). Monte-carlo tree search in settlers of catan. In H. van den Herik & P. Spronck (Eds.), *Advances in Computer Games*, volume 6048 of *Lecture Notes in Computer Science* (pp. 21–32). Springer Berlin / Heidelberg.

Tedrake, R., Zhang, T., & Seung, H. (2004). Stochastic policy gradient reinforcement learning on a simple 3d biped. In *Intelligent Robots and Systems, 2004. (IROS 2004). Proceedings. 2004 IEEE/RSJ International Conference on*, volume 3 (pp. 2849 – 2854 vol.3).

Tesauro, G. (1995). Temporal difference learning and td-gammon. *Commun. ACM*, 38(3), 58–68.

Thiery, C. & Scherrer, B. (2009). Building controllers for tetris.

Thrun, S. (1995). Learning to play the game of chess. In *Advances in Neural Information Processing Systems 7* (pp. 1069–1076).: The MIT Press.

Thrun, S. & Schwartz, A. (1993). Issues in using function approximation for reinforcement learning. In *In Proceedings of the Fourth Connectionist Models Summer School*: Erlbaum.

Trescak, T., Esteva, M., & Rodriguez, I. (2012). A shape grammar interpreter for rectilinear forms. *Computer-Aided Design*, 44(7), 657–670.

Vamplew, P., Dazeley, R., Berry, A., Issabekov, R., & Dekker, E. (2011). Empirical evaluation methods for multiobjective reinforcement learning algorithms. *Mach. Learn.*, 84(1-2), 51–80.

Vamplew, P., Yearwood, J., Dazeley, R., & Berry, A. (2008). On the limitations of scalarisation for multi-objective reinforcement learning of pareto fronts. In *AI 2008: Advances in Artificial Intelligence* chapter 37, (pp. 372–378). Springer.

Van Moffaert, K., Drugan, M., & Nowe, A. (2013b). Scalarized multi-objective reinforcement learning: Novel design techniques. In *Adaptive Dynamic Programming And Reinforcement Learning (ADPRL), 2013 IEEE Symposium on* (pp. 191–199).

Van Moffaert, K. & Nowé, A. (2014). Multi-objective reinforcement learning using sets of pareto dominating policies. *The Journal of Machine Learning Research*, 15(1), 3483–3512.

Veness, J., Silver, D., Uther, W. T. B., & Blair, A. (2009). Bootstrapping from game tree search. In Y. Bengio, D. Schuurmans, J. D. Lafferty, C. K. I. Williams, & A. Culotta (Eds.), *NIPS* (pp. 1937–1945).: Curran Associates, Inc.

Verbeeck, K. (2006). Randomness as a generative principle in art and architecture. Master Thesis, School of Architecture, MIT.

Wang, Y.-C. & Usher, J. M. (2005). Application of reinforcement learning for agent-based production scheduling. *Engineering Applications of Artificial Intelligence*, 18(1), 73 – 82.

Watkins, C. J. (1989). *Learning from delayed rewards.* PhD thesis, University of Cambridge.

White, D. J. (1982). Multi-objective infinite-horizon discounted markov decision processes. *Journal of Mathematical Analysis and Applications*, 89.

Wiering, M. & van Otterlo, M. (2012). *Reinforcement Learning: State-of-the-Art*. Springer.

Wiering, M. A. & de Jong, E. D. (2007). Computing optimal stationary policies for Multi-Objective markov decision processes.

Wolfram, S. (2002). *A new kind of science*, volume 5. Wolfram media Champaign.

Woodbury, R. F. & Burrow, A. L. (2006). Whither design space? *AIE EDAM: Artificial Intelligence for Engineering Design, Analysis, and Manufacturing*, 20(02), 63–82.

Wu, Z., Campbell, M. I., & Fernández, B. R. (2008). Bond graph based automated modeling for computer-aided design of dynamic systems. *Journal of Mechanical Design*, 130(4), 041102.

Wyatt, D. F., Wynn, D. C., Jarrett, J. P., & Clarkson, P. J. (2012). Supporting product architecture design using computational design synthesis with network structure constraints. *Research in Engineering Design*, 23(1), 17–52.

Yue, K. & Krishnamurti, R. (2013). Tractable shape grammars. *Environment and Planning B: Planning and Design*, 40(4), 576–594.

Zhang, W. & Dietterich, T. G. (1995). A reinforcement learning approach to job-shop scheduling. In *In Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence* (pp. 1114–1120).: Morgan Kaufmann.

Zhao, Y., Kosorok, M. R., & Zeng, D. (2009). Reinforcement learning design for cancer clinical trials. *Stat Med*.