Design with shape grammars and reinforcement learning

Manuela Ruiz-Montiel[a], Javier Boned[b], Juan Gavilanes[b], Eduardo Jiménez, Lawrence Mandow[a], José-Luis Pérez-de-la-Cruz[a]

[a]*Escuela Técnica Superior de Ingeniería Informática. Bulevar Louis Pasteur, Nº 35. Campus de Teatinos. Universidad de Málaga. 29071 Málaga, España*
[b]*Escuela Técnica Superior de Arquitectura. Campus de El Ejido. Universidad de Málaga. 29071, Málaga, España*

Abstract

Shape grammars are a powerful and appealing formalism for automatic shape generation in computer-based design systems. This paper presents a proposal complementing the generative power of shape grammars with reinforcement learning techniques. We use simple (*naive*) shape grammars capable of generating a large variety of different designs. In order to generate those designs that comply with given design requirements, the grammar is subject to a process of machine learning using reinforcement learning techniques. Based on this method, we have developed a system for architectural design, aimed at generating two-dimensional layout schemes of single-family housing units. Using relatively simple grammar rules, we learn to generate schemes that satisfy a set of requirements stated in a design guideline. Obtained results are presented and discussed.[1]

Keywords:
computational design, shape grammars, reinforcement learning, architecture

1. Introduction

The problem of computational design has been approached by Artificial Intelligence researchers since the 1960s [1, 2]. Design problem solving is

[1]This work is partially funded by: grant TIN2009-14179 (Spanish Government, Plan Nacional de I+D+i). Manuela Ruiz-Montiel is funded by the Spanish Ministry of Education through the National F.P.U. Program.

considered a particularly complex task, since these kind of problems are ill-structured [3], 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.

In this work we deal with the computational design problem of automatic, partially-directed generation of design alternatives according to certain criteria. The diverse variants of this problem are of interest in many fields like architecture, engineering or packaging, where geometric design plays a crucial role. Certainly, an assistant system that generates and proposes different design alternatives can be quite worthy for design stakeholders in the first stages of the design process.

More specifically, we propose a computational design methodology using *shape grammars* [4, 5, 6] as design synthesis mechanism. Shape grammars have been used in the literature for numerous design tasks related to areas like architecture or industrial design. In this context, shape grammars have been used to generate designs according to functional requirements in a number of ways. However, the development of adequate methodologies to create and control the application of shape grammars remains an open research topic.

The direct, traditional approach requires a human expert to design an *expert* shape grammar and thus hard-code the whole set of design requirements inside the rules [7, 8, 9, 10, 11, 12]. This practice is aimed at producing admissible solutions, but the involved grammars tend to be difficult to create, modify and maintain. Moreover, these grammars normally reduce the space of alternatives, since they have been designed with the foreknowledge of the nature of the results. Albeit this philosophy allows the quick synthesis of valid designs, expert shape grammars also facilitate obtaining designs that are prone to be very similar between them.

Alternatively, some researchers have proposed the use of more *naive* shape grammars. The blind execution of a generation system based on a naive grammar would lead to a great number of different designs, but usually the majority of them would be unfeasible. This problem has been generally addressed by means of optimization techniques that guide shape generation [13, 14, 15, 16, 17, 18, 19, 20, 21, 22]. Some of these methods yield a varied set of solutions, but we aim to go one step beyond and be able to generate every solution that meets a set of requirements.

In this paper we propose the combination of naive shape grammars with

reinforcement learning [23]. Instead of directly reaching solutions, the algorithm learns *how to* generate them; after a simple grammar is set up, our proposal incorporates a phase of *policy learning* in which a reinforcement learning algorithm is applied, yielding a policy that will guide the execution of the naive shape grammar towards feasible solutions. Design requirements are formulated as *rewards* in this phase, and the learning algorithm learns how to optimize the reward received during shape generation. This methodology allows the generation of a large variety of feasible designs using naive rules that are easy to understand and maintain.

We have applied the proposed methodology to a particular area of architectural design: the generation of single-family, basic house layout plans that comply with a certain guideline. This can be of great help in the first stages of architectural design, when the designer is at risk of undergoing the *blank page syndrome* and can benefit from the existence of many feasible and varied starting points that have been obtained effortlessly. These starting points are schematic, as they are not intended to be standard architectural plans, but layouts that could later work as inception for complete, standard projects carried out by an architect.

The article is structured as follows: in Section 2 some introductory material about shape grammars and reinforcement learning is presented in order to make the paper self-contained. Then, the structure of the system is described in Section 3, addressing generation and learning issues. Next, in Section 4, results of the system are presented and analysed in terms of time generation and feasibility. In Section 5 our system is discussed from several points of view. Finally, some conclusions are drawn and possible future lines of continuation and generalization of this work are pointed out.

2. Antecedents

2.1. Shape grammars

Shape grammars are a computational formalism for the generation of geometric shapes introduced by Stiny and Gips [4, 5, 6]. This section summarizes some basic definitions needed later in this paper. Informally, shape grammars are a set of rules that govern the composition of simple geometric and symbolic elements to generate complex shapes.

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 $\alpha \to \beta$, 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 $\alpha \to \beta$ 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 $\alpha \to \beta$ 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 1 displays a rule and one sample derivation, i.e., a possible sequence of shapes generated by successive applications of the rule.

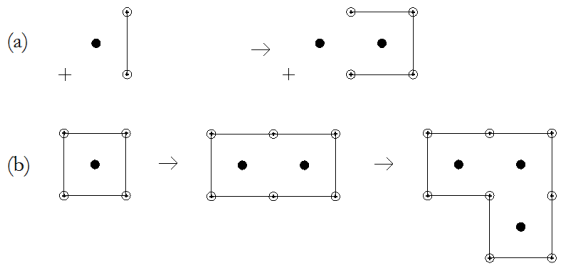


Figure 1: A rule (a) and one derivation starting from an initial squared shape (b)

Shape grammars have been used in the literature for many tasks related to architectural or industrial design. In architecture, the works of Stiny

and Gips [8] and Cagdas [9] 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 [24] 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 and Tang [25] to generate compact digital cameras, considering different combinations of form criteria; the one proposed by Agarwal, Cagan and Constantine [26] for the generation of coffee-makers, taking into account different cost-related aspects; and the one devised by MacCormack, Cagan and Vogel [12] 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 [27]. More recently, Shea et al. developed a framework for an autonomous design-to-fabrication system that automatically fabricates customized parts [28]; in this framework, the shape rules encode primitive movements of the machine tool.

The wide appeal of shape grammars comes on one hand from their versatility. It is well known that they are capable of producing any possible shape [29]. On the other hand, 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 [30]. As Chase pointed out, design systems based on grammars have a great potential to automate and explore many design alternatives without tedious work [31]. 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

	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 1: SWOT analysis of the use of shape grammars for design

means of a 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 1).

This paper addresses the critical issue (in fact, one the *weaknesses* of shape grammars) of controlling the application of a set of shape grammar rules in a particular domain, so as to satisfy a set of design goals. This process is knowledge intensive, since it requires human expertise to define the appropriate set of requirements. The next sections briefly review previous research in the field. Then an innovative approach based on reinforcement learning is presented and evaluated.

2.2. Expert 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" [32, pag. 7].

The execution of such rules would thus easily 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. For example, the work of Duarte [24] considers a set of guidelines imposed by a housing program; and also many other works aim to design architectural or engineering objects in accordance with a particular style or brand identity [8, 9, 10, 11, 12]. Figure 2 displays a fragment of the Palladian grammar used by Stiny and Mitchell [8] to generate 2-dimensional plans of villas similar to those of the famous XVI century architect Andrea Palladio (see Figure 3).

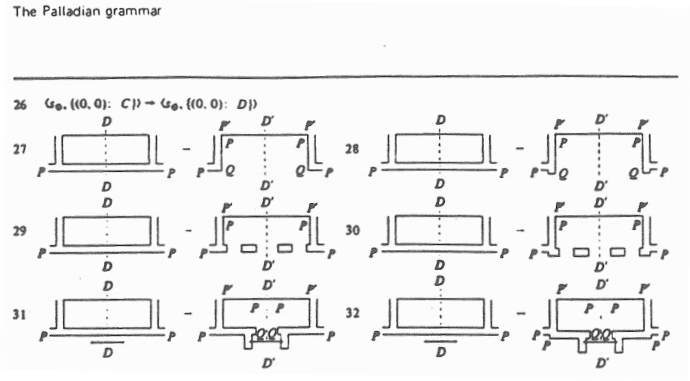


Figure 2: Seven rules of the Palladian grammar (reproduced from [8])

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

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).

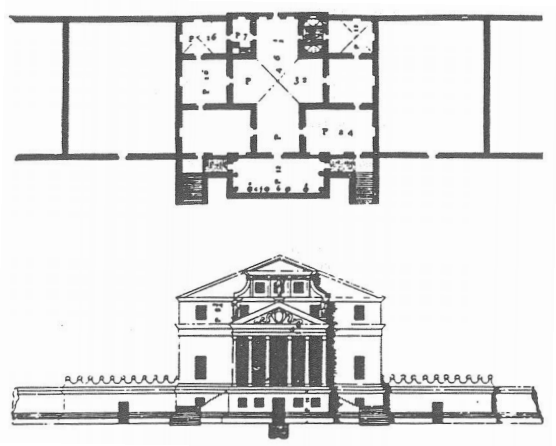


Figure 3: The Villa Malcontenta as drawn by Palladio (reproduced from [8])

A different approach is taken by Duarte in his discursive grammars [24]: 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. Another alternative is the one taken by Agarwal, Cagan and Constantine [26], in which the intermediate shapes produced by the rules are measured by means of cost expressions. Results are delivered to an expert designer who can use them to decide which rules to apply next.

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" [33]. Additionally, the use of the common mechanisms to introduce expert knowledge inside rules 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.

2.3. Naive shape grammars

A different approach calls for the combination of simple shape grammars with specific methods to guide the generation process. The grammars could lead to a great diversity in results, while the control methods would guarantee the fulfilment of design requirements. We can call these simpler shape grammars *naive*, because an arbitrary execution of their rules without any additional guiding mechanism would not guarantee feasible designs. 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.

Previous works on goal-oriented design generation with naive shape grammars have usually defined the task as an *optimization* problem, with a quantifiable objective function to maximize or minimize, and an optional set of design constraints. Two main optimization techniques have been used: *shape annealing* and *evolutionary algorithms*.

Simulated annealing [34] 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* [13, 15] applies simulated annealing to shape grammars. In this context, we seek a shape that minimizes the value of a given objective function, interpreted as the *energy* of the shape. 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 [15, 16] or dome design [14]. 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, Louis and Kundu [17] 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 and Kazakov [18] 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 [19] 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 and Tang [20]. It uses genetic programming in order to determine rule parameters for

the problem of designing the body of a camera. Chouchoulas [21] 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 [22] 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 [20, 19] offer just a single solution to a given problem (an optimal or near-optimal one), while others [17, 18, 21] provide few distinct, feasible solutions (less than 10), which are picked from the last generation.

2.4. Reinforcement Learning

In this work we aim to learn how to apply a set of given naive shape rules so as to produce feasible design alternatives. That is, we seek a function that determines a suitable *transformation* to be applied to a given shape. We will use this function at every step of the shape grammar derivation, in order to finally produce a feasible solution according to a set of design criteria. A different but related problem is that of grammatical inference [35]. However, the aim of this paper is to learn the order of rule application, and not the rules themselves.

Reinforcement learning [23] 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 (one of the most researched approaches in machine 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 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 [36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46], games [47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59], scheduling systems [60, 61, 62], computer vision [63, 64, 65, 66], dynamic channel allocation in cellular networks [67, 68, 69] or medicine [70, 71].

The solution to a reinforcement learning problem is a *policy*, which is basically a function that maps each possible *state* to an *action* to be taken when at that state. Independently of its immediate reward (which depends solely on the environment), each state has an associated *value*, 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.

There exist three elementary methods for solving a reinforcement learning problem [23]:

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.

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. TD(λ) techniques are a generalization of Monte Carlo methods and TD learning; 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. *Q-learning* [72] (see Figure 4) is a TD learning algorithm with minimal convergence requirements that has been intensively researched and successfully used in many works [36, 63, 65, 38, 73, 39, 69, 54, 62, 70, 71]. Here we use the generalized version of this method, that is, Q(λ).

In the following we describe this technique. For simplicity, we stick to the Q(0) case, that is, the pure TD approach. 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.

The framework of reinforcement learning adjusts well to our case: actions will be applications of shape grammar rules to the current shape. We will receive positive rewards depending on how well the resulting shape responds to certain design requirements. In our application of Q-learning to naive shape rules we have to address two practical issues.

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., ε-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 4: Q(0) algorithm (reproduced from [23])

1. The first one is related to how much we trust in our current policy during the learning stage. It is usual to follow an exploitation-exploration strategy. Most of the time, the algorithm exploits the current policy, 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 chosen with probability ϵ.
2. The second one deals with *generalization*. In many practical cases, the number of state-action pairs is so large that cannot be stored in a table. This is the case with shape grammars, since we have as many states as possible shapes than can be generated, and as many actions as transformations can be applied to match the left-hand side of the rule to the current shape. 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.
   In this work, we use a linear function of features, which worked well in our experiments over the housing unit domain. More precisely, we postulate a 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 learnt 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 in Q-learning but, at the same time, the use of features to represent states has certain advantages. Many different shapes 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 learnt even for states that were never visited during the learning stage. The features selected for our shape grammar domain will be described in Section 3.3.

The algorithm we have used for $Q(\lambda)$ (that is, the generalized version of Q-learning, so as to control the bootstrapping level) with linear approximation and binary features can be found in [23] (p. 213).

3. A system for the generation of housing unit designs

In this section we describe our approach combining shape grammars and reinforcement learning through a knowledge-intensive test case: the design of single-family housing units. In the conceptual stage 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.

3.1. 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 [74] 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 2 summarizes the imposed explicit constraints. The nature of the house contour is not constrained by our guidelines [74], 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.

In Figure 5 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 pos-

Global requirements
R1: Total area must be of at least 46 m^2
R2: The contour must be compact (not scattered)
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 2: Requirement set for a single-family basic house (adapted from [74])

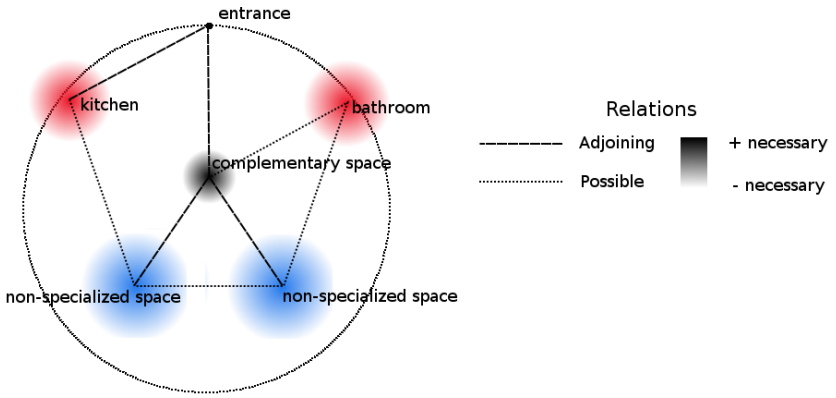


Figure 5: Proximity relationships in a single-family house (adapted from [74])

sible 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 2 and in Figure 5 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.

All the implementations along this work have been programmed using the Ruby language [75]. For visualization and geometry-related tasks we have used Trimble SketchUp [76], a 3D modelling tool that allows incorporation of Ruby scripts. The main advantages of this technological choice are that (a) SketchUp makes the edition and visualization tasks very easy, and (b) it additionally provides a complete set of geometry operations that are very useful when implementing ad-hoc shape grammars interpreters by means of Ruby. The main shortcoming of this technological choice is the increase in computing time that the use of an interpreted language may introduce.

3.2. Naive grammars and phases of generation

For the synthesis of housing units, we use naive shape grammars. Therefore, the creation and management of rules is to a great extent easy. However, we need additional mechanisms to control the execution of each rule, as explained in Section 2.3.

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 6 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 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 6, 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). 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 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 3.

This set of rules can be used without further guidance to generate schemes; when executing each rule, all possible transformations are computed and one of them is selected at random. Two examples are depicted in Figure 11 For the final presentation of the schemes, two additional operations are performed: (a) erasing of the labels and (b) addition of significant text in order to distinguish each space. As we will discuss in Section 4, the schemes are not actually admissible, highlighting the need of extra guiding mechanisms in order to comply with housing unit requirements.

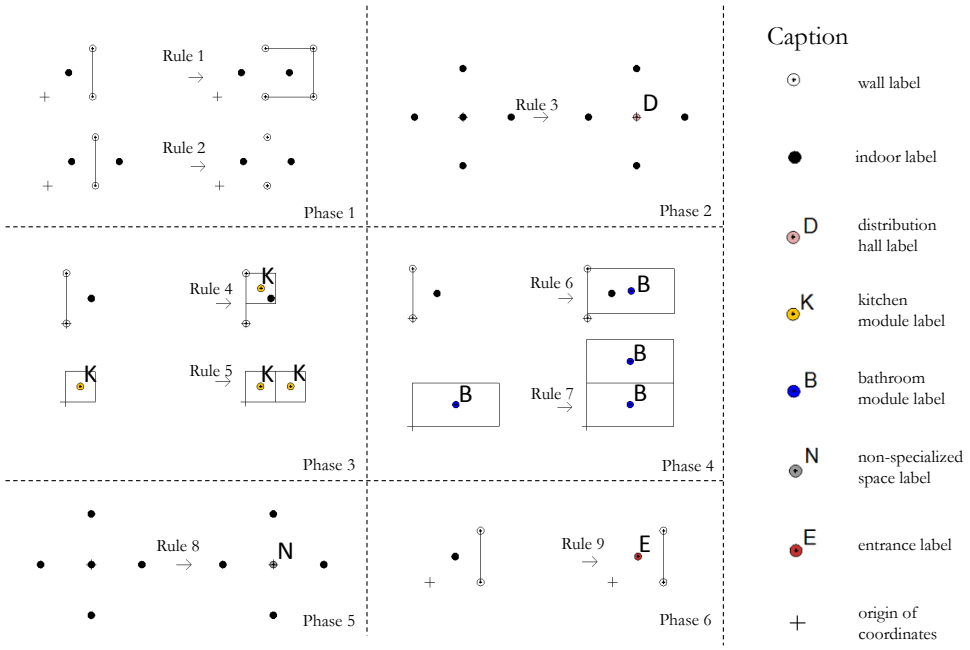


Figure 6: Naive grammars for phases 1-6

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 3: Final state tests for every rule

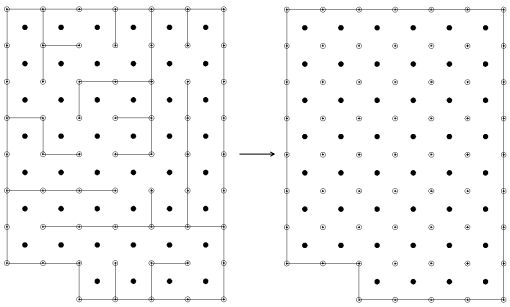


Figure 7: Effect of the application of rule 2

3.3. Learning processes for naive shape rules

In order to generate valid designs, we must incorporate a policy into each naive rule. 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 6 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 7).

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 are learnt through reinforcement learning. As explained in section 2.4, in a reinforcement learning process we interact with our environment by obtaining rewards; these will be determined from the conditions described in Section 3.1, concretely the ones in Table 2 and Figure 5. We also need to establish state 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.

As an example of how policy learning is accomplished, we will consider rule number 5 to explain in detail the features and rewards that have been used. Rule 5 is in charge of placing the kitchen modules. It is depicted in figure 6. As we can see, the rule simply states that an additional module can

be placed next to an existing kitchen module.

The reward for the learning process of this rule is defined as follows: the more requirements the 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. 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 [74]:

- $r_{5,1}(s) = 1$ 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.

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

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

Some extra expert knowledge can be used to define this reward. In the case of rule 5, this expertise is translated in giving 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:

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

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 4: Rewards for every learning process

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

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

The reward $r(s)$ refers to the short-term value of certain shape. 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 reward expressions used for each rule are gathered in Table 4. Rewards are assigned to states at every step, except in the first rule, where they are assigned only to final states. This is due to the fact that we only have valuable information of the compactness of a contour when all the modules have been placed.

The complete set of requirements that define rewards for rules 4-9 is gathered in Table 5. All these requirements are binary, i.e one or zero when the associated predicate is satisfied or not respectively.

Table 5: Requirements for rules 4-9

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
Continued on Next Page...

Table 5 – Continued

$r_{5,4}(s) = 1$ 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

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. Other approaches for features definition are possible (apart from directly translating design requirements). For example, in the case of rule 1, the design requirement is a high compactness factor, and we have not directly defined the features from this requirement, as in rules 4-9. Using any approach, 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 the case of rule 1, 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:

- $f_{1,1}(s,a)$ = number of modules of 1 m^2 with 1 neighbour
- $f_{1,2}(s,a), f_{1,3}(s,a), f_{1,4}(s,a)$ analogously to $f_{1,1}(s,a)$ with 2, 3 and 4 neighbours respectively

As these features are continuous, they have been normalized between 0 and 1, by dividing by the total number of modules of 1 m^2 that are going to exist at the end of one learning episode; in this case this number is 46.

In the end, the learning algorithm will learn how much these features relate to each other. If the relation were not linear, then the learning algorithm would fail. However, our results suggest linear relations capture this relation well, since scattered contours are not generated.

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 6.

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 6: Policies learnt for every rule

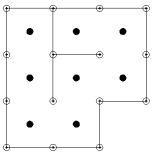


Figure 8: Shape produced from one derivation of rule 1

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.

In order to clarify how the learnt policies guide rule derivation, we will go through one step of the guided application of rule 1 (the process is the same for every rule). Let us suppose that, at a certain moment, the derivation of rule 1 has yielded the shape in Figure 8.

There are many possible transformation alternatives for the next application of rule 1. 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

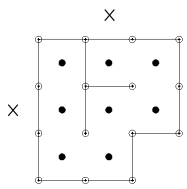


Figure 9: Pattern that represents two shapes obtained from the shape of Figure 8 sharing the same features

rule 1 could add the extra black label of its right part (see Figure 6). For example, in Figure 9 we can see the pattern for two shapes that share the following features:

- $f_{1,1}(s, A) = 1/46$
- $f_{1,2}(s, A) = 5/46$
- $f_{1,3}(s, A) = 1/46$
- $f_{1,4}(s, A) = 2/46$

Where s is the shape depicted in Figure 8 and A is the set of transformations of rule 1 that yield some of the shapes included in the pattern of Figure 9 (in this case $|A| = 2$). We can now compute the value for these features by means of the coefficients for rule 1 (see Table 6):

$$Q_1(s, A) = \theta_1 f_{1,1}(s, A) + \cdots + \theta_4 f_{1,4}(s, A) = 0,1843/46$$

The application of rule 1 to the shape in Figure 8 yields four different combinations of features. In Table 7 we gather the four corresponding patterns along with their features and the computed value.

In light of the computed values gathered in Table 7, the chosen transformation is one of those that yields the pattern in the first row of Table 7 (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 10.

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 4 we show some obtained results.

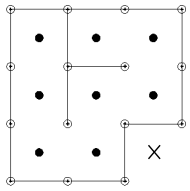
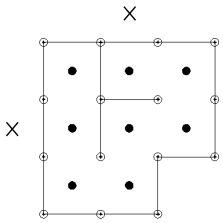
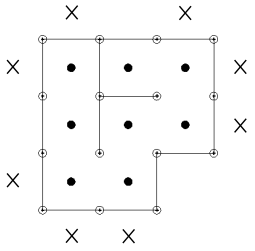
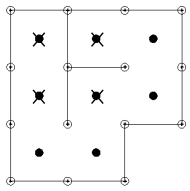
Pattern	Features	Value
	$f_{1,1}(s, A_1) = 0$ $f_{1,2}(s, A_1) = 4/46$ $f_{1,3}(s, A_1) = 4/46$ $f_{1,4}(s, A_1) = 1/46$	$0, 2152/46$
	$f_{1,1}(s, A_2) = 1/46$ $f_{1,2}(s, A_2) = 5/46$ $f_{1,3}(s, A_2) = 1/46$ $f_{1,4}(s, A_2) = 2/46$	$0, 1843/46$
	$f_{1,1}(s, A_3) = 1/46$ $f_{1,2}(s, A_3) = 4/46$ $f_{1,3}(s, A_3) = 3/46$ $f_{1,4}(s, A_3) = 1/46$	$0, 1851/46$
	$f_{1,1}(s, A_4) = 0/46$ $f_{1,2}(s, A_4) = 5/46$ $f_{1,3}(s, A_4) = 2/46$ $f_{1,4}(s, A_4) = 1/46$	$0, 152/46$

Table 7: Patterns yielded by one application of rule 1 to the shape in Figure 8

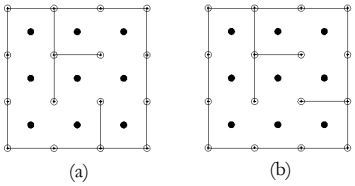


Figure 10: Shapes represented by the best pattern obtained from the shape of Figure 8

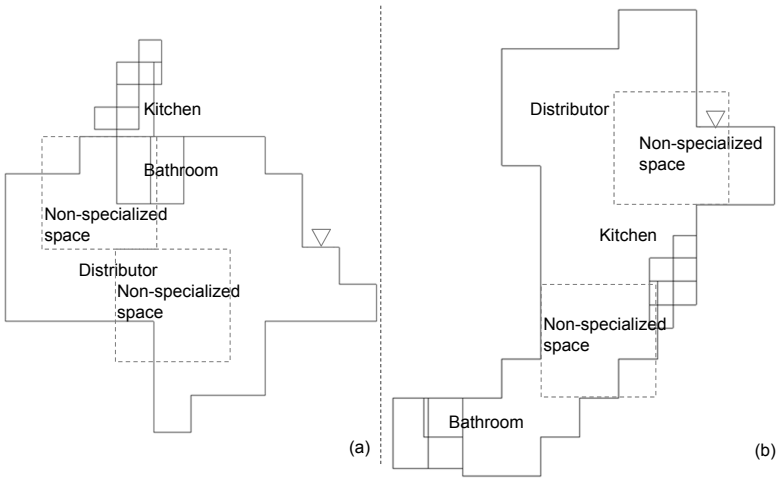


Figure 11: Two schemes obtained with naive grammars

4. Results

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

4.1. Naive grammars without policies

A number of schemes were generated with the naive set of rules depicted in Figure 6, without benefiting from any process of learning. Namely, the system was used to generate 100 schemes (two of them are shown in Figure 11). 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 3.1. In particular, those in Figure 11 violate the following requirements of Table 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, 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.

4.2. Learning processes

In our system we have used the algorithm $Q(\lambda)$ outlined in section 2.4 with the following parameter choice: learning rate $\alpha = 0.1$; exploration rate $\epsilon = 0.3$; discount rate $\gamma = 0.8$; bootstrapping level $\lambda = 0.5$. In Section 5 we describe a parametric study that has been carried out in order to determine how different settings of parameters affect the learning processes.

In table 8 some information about the learning processes for the different rules is shown. For each rule amenable to be learnt (namely, rules 1, 4, 5, 6, 7, 8 and 9) the number of learning episodes (in Section 5 we describe how these numbers have been established) and different initial states are shown. Every learning episode ends when a final state is reached (see Table 3). The last column shows time in seconds taken by the learning process.

4.3. 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 6). Only 11 out of these 100 schemes violated some of the constraints in Table 2. Therefore, compared to the ones generated by the naive grammars alone, these are closer to fulfil the housing program described in Section 3.1.

We arbitrarily chose 12 schemes out of the valid ones. The generated schemes are depicted in Figures 12 and 13. Computation time was fast; the minimum generation time for a scheme was 33,94 s and the maximum 44,35

Rule	No. of Episodes	No. of initial states	Learning time (in seconds)
1	5000	1	4728,42
4	100	10	76,57
5	100	10	270,758
6	100	10	126,75
7	100	10	481,215
8	100	10	130,198
9	100	10	53,611

Table 8: Information of learning processes

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 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.

5. Discussion

The results shown in section 4 can be discussed from several points of view. We will focus on four issues:

1. Admissibility of generated schemes
2. Architectural evaluation of the generated schemes
3. Variability of generated schemes and generation times
4. Knowledge engineering effort

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 2 and the relationships depicted in Figure 5). 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 4.1.

However, when policies are learnt the rules generate better designs, as the rewards considering Montaner criteria are maximized. Regarding conditions of Table 2 and Figure 5, we found that the requirements were violated only 22

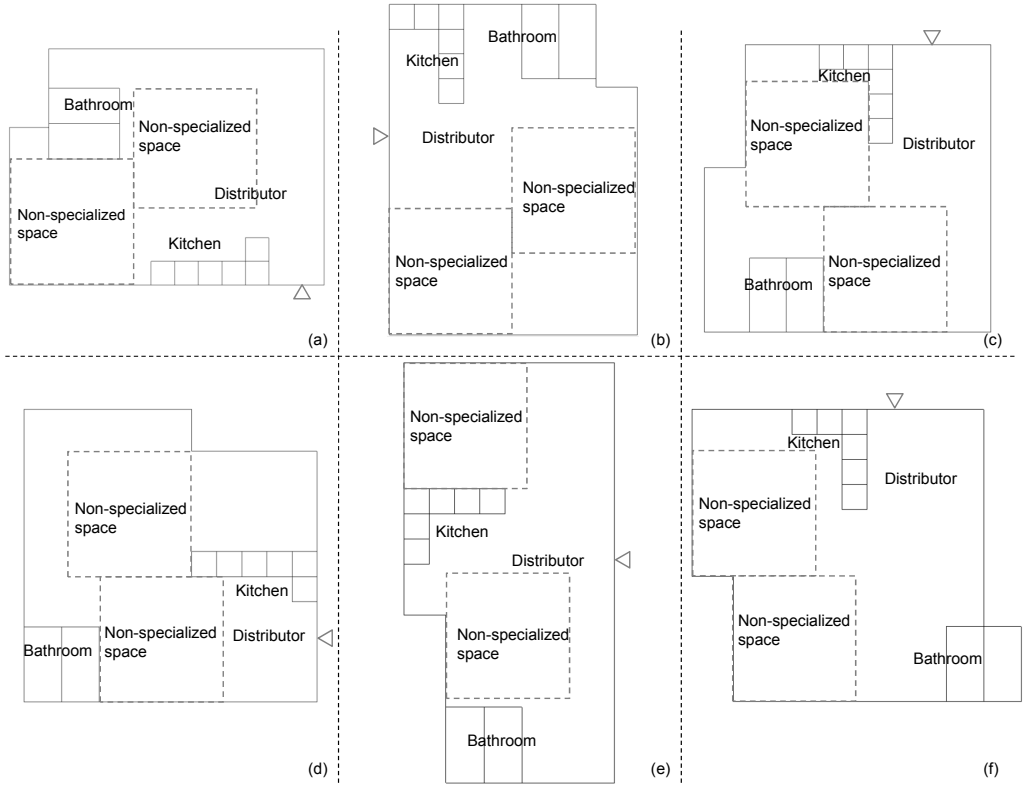


Figure 12: Some generated designs (results a-f)

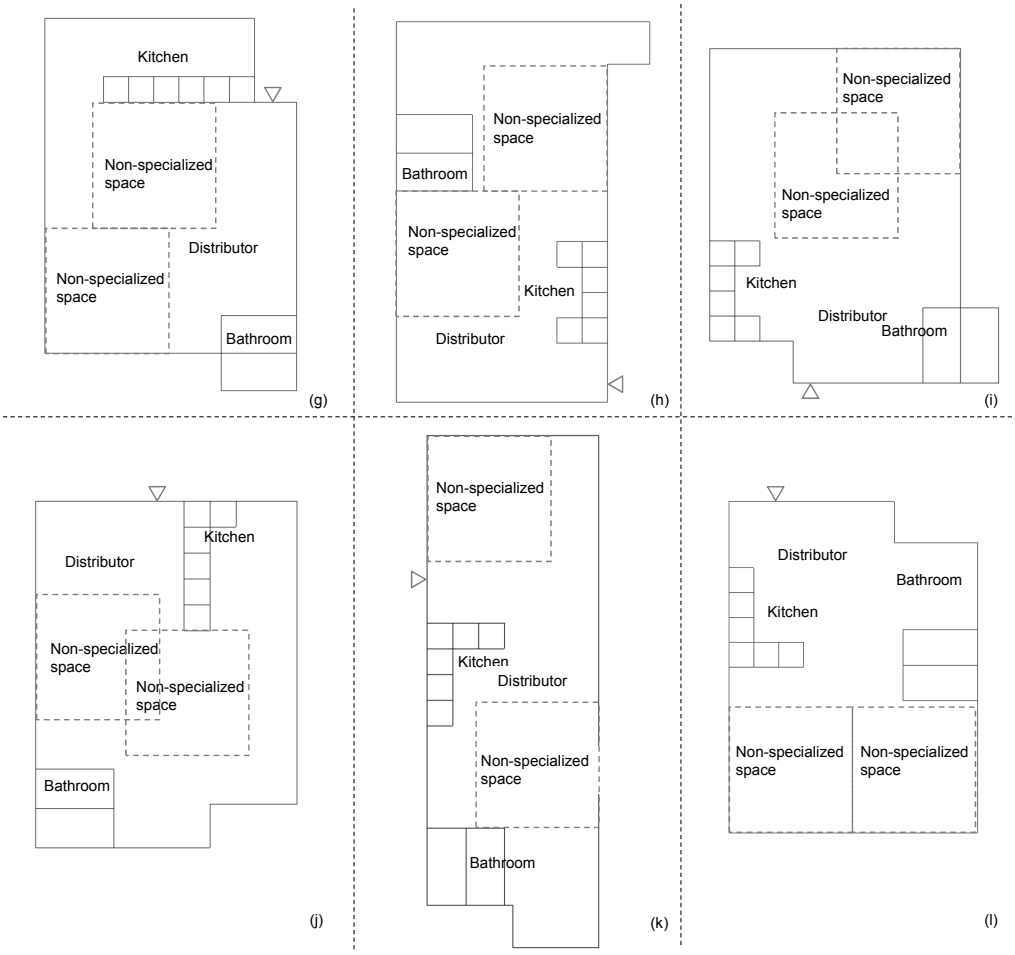


Figure 13: Some generated designs (results g-l)

Requirement reference	Number of schemes that violate the requirement
R3	6
R4	2
R8	5
Adjoining relation between entrance and distribution hall	3
Adjoining relation between non-specialized space and distribution hall	6
rest of requirements	0

Table 9: Requirements violated by the generated schemes

times. This fact is detailed in Table 9, where we list the number of schemes that violate each criterion.

Architectural evaluation of the generated schemes. Our system, applied to the generation of housing units according to a guideline, aims to help architects in overcoming the *blank page syndrome*, providing them with many distinct and feasible solutions. Once these solutions have been obtained, design practitioners can evaluate them in order to discover promising inceptions for their projects. This evaluation is done from a more general and subjective architectural perspective. In this section we describe this process of evaluation.

The results in Figures 12 and 13 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 archi-

tectural 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 12 and 13 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.

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 alternatives for the first stages of the design process. 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 4 and shown in Figures 12 and 13, 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 2.4, see Table 7). 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 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.

Knowledge engineering effort. The use of conventional expert shape grammars requires an important knowledge engineering effort. In the case considered in Section 3.1, a set of rules that generate all valid designs of housing units according to the program under consideration would have to be defined. The main difficulties to overcome would be brittleness, maintenance, precedence, and unanticipated interactions of the expert shape rules.

In contrast, the learning system proposed here makes use of naive shape grammars and an explicit rule ordering dictated by elementary considerations. For example, rules 4 and 5 in Figure 6 add standard kitchen modules of 60×60 cm., but do not prescribe how these modules are to be exactly placed in the final design. These sets of naive rules are robust and easy to create, understand, modify and maintain.

Policies on rule application are the necessary complement to this simple approach. These guarantee that rules interact in desirable ways according to the housing design program. Policies are learned automatically from rewards and features as explained in Section 3.3. Therefore, the knowledge engineer-

ing effort concentrates mainly on defining these rewards. In the presented case, these also follow easily from the design program.

However, stochastic learning algorithms, like the one used in this study, incorporate a number of parameters that need to be set before learning is actually carried out. Analysing the effect of all parameters simultaneously in algorithm performance is a combinatorially difficult task and is never carried out in practice. Instead, each parameter is usually analysed in isolation, using standard values for the others. Our case involves only four parameters, which can be summarized in order of importance as follows,

- Bootstrapping level ($\lambda \in [0, 1]$). If $\lambda = 0$, then the algorithm uses pure temporal differences. If $\lambda = 1$, then the algorithm becomes a Monte-Carlo method.

- Discount rate ($\gamma \in [0, 1]$) . If $\gamma = 0$, then the algorithm favours actions with immediate rewards, while with $\gamma = 1$ long-term rewards are sought. Although our approach seeks long-term rewards, we somewhat reduce this value according to the recommendations of Thrun and Schwartz [77].

- Exploration rate ($\epsilon \in [0, 1]$). This is the probability that the algorithm will ignore the current policy in favour or random exploration during the learning process.

- Learning rate ($\alpha \in [0, 1]$). This is a standard parameter in many learning algorithms that controls how fast learning takes place. However, large values can lead to convergence problems.

In the experiments described along the paper, these parameters were set to the following "standard", conservative values: $\lambda = 0.5, \gamma = 0.8, \epsilon = 0.3, \gamma = 0.1$. In order to study the influence of the parameters on the results obtained by the Q-learning algorithm, additional experiments were carried out. For the compact contour generation problem, parameters were sequentially analysed and set according to the initial values and phases described in Table 10. The first parameter analysed was the bootstrapping level (λ). A wide range of five different values was analysed (0, 0.25, 0.5, 0.75, 1) while the rest of parameters were set to the values we chose ($\gamma = 0.8, \epsilon = 0.3, \alpha = 0.1$). The best value for λ was obtained assessing the quality of the generated designs according

Phase	Parameter	Plausible values	Fixed values
1	λ	$(0, 0.25, 0.5, 0.75, 1)$	$\gamma = 0.8, \epsilon = 0.3, \alpha = 0.1$
2	γ	$(0.7, 0.8, 0.9, 1)$	$\lambda = \lambda_{phase_1}, \epsilon = 0.3, \alpha = 0.1$
3	ϵ	$(0.1, 0.2, 0.3, 0.4)$	$\lambda = \lambda_{phase_1}, \gamma = \gamma_{phase_2}, \alpha = 0.1$
4	α	$(0.1, 0.15, 0.2)$	$\lambda = \lambda_{phase_1}, \gamma = \gamma_{phase_2}, \epsilon = \epsilon_{phase_3}$

Table 10: Sensitivity analysis for the involved parameters in the algorithm Q(λ)

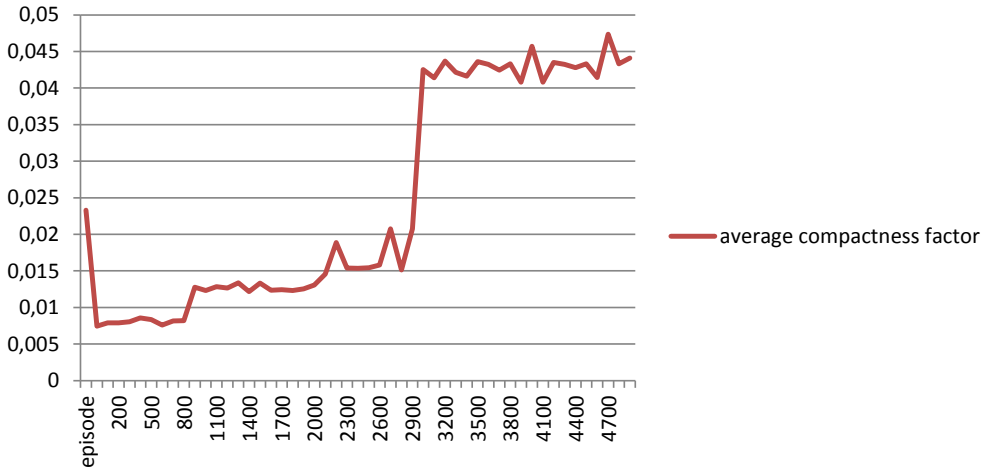


Figure 14: Average compactness factor during the learning process

to their compactness factors. Then, the analysis of the other parameters was carried out in a similar fashion.

Most of the experimentation was devoted to parameter λ. Up to 3000 learning episodes were needed to produce good designs. Figure 14 shows the average compactness factor of produced designs with intermediate learned policies during the process for $\lambda = 0.5$. However, algorithm $Q(\lambda)$ 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 finally chosen to take advantage of the speed provided by bootstrapping methods and the safety of Monte-Carlo techniques (see [23], Section 8.6).

Regarding the discount rate, values of $\gamma = 0.9$ and $\gamma = 1$ provided poor results, as already noticed by Thrun and Schwartz [77]. Therefore, the initial value of $\gamma = 0.8$ was preserved. Finally, the algorithm proved to be robust regarding variations of ϵ and α, so the initial values were also preserved.

In summary, standard parameter settings proved to perform well and quite robustly in this domain.

6. Conclusions and future work

This work deals with the computational design problem of automatic, partially-directed generation of design alternatives according to certain criteria. A novel proposal is presented to complement the generative power of shape grammars with reinforcement learning techniques. The approach makes use of simple (naive) shape grammars that are easy to elaborate and understand. Unlike previous approaches based on naive grammars, the generation process is controlled by policies learnt with reinforcement learning techniques. The power of shape grammars allows to generate a large variety of designs, while reinforcement learning helps to obtain policies that guide the generation process towards those designs that satisfy given design requirements.

This methodology has been applied to generate two-person, basic housing 2D scheme units. Working with floor plans simplifies geometry issues, and is enough to allow the establishment of relationships between spaces. A simple set of rules has been presented for the different phases of housing unit design. Adequate policies for design generation were learnt according to an adapted set of the constraints and conditions proposed by a design guide elaborated for the regional government of Andalusia, Spain. Learning is based on rewards and features, taken directly from the conditions established in the design guide. Learning processes are easy to define and run in reasonable time.

The system generates schemes which are clearly better than the ones generated by the same grammars without learning. Schemes also present a great variability and are reasonable solutions to the housing unit design problem analysed. Time generation is also reasonable, allowing architects to consider simultaneously a huge number of automatically generated starting points (possibly unexpected) that will generally meet the considered set of design criteria.

The natural continuation of this work would be to apply the same methodology (simple shape grammars and reinforcement learning) to more complex problems of architectural and engineering design. Our 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 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. A more ambitious continuation would be to relax rule ordering, allowing more flexibility in the processes of generation and learning. The application of multi-objective reinforcement learning techniques to naive design grammars is also an interesting line of future research.

References

[1] H. A. Simon, The sciences of the artificial, MIT Press, Cambridge, MA., 1968.

[2] C. M. Eastman, Cognitive processes and ill-defined problems: a case study from design, in: International Joint Conferences on Artificial Intelligence '69, pp. 669–690.

[3] H. A. Simon, The structure of ill structured problems, Artificial Intelligence 4 (1973) 181–201.

[4] G. Stiny, J. Gips, Shape grammars and the generative specification of painting and sculpture, in: Information Processing 71, North-Holland, 1972, pp. 1460–1465.

[5] G. Stiny, Introduction to shape and shape grammars, Environment and Planning B 7 (1980) 343–351.

[6] G. Stiny, Shape. Talking about seeing and doing, MIT Press, Cambridge, Ma., 2006.

[7] T. W. Knight, Shape grammars: six types, Environment and Planning B 26 (1999) 15–31.

[8] G. Stiny, W. J. Mitchell, The Palladian grammar, Environment and Planning B 5 (1978) 5–18.

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

[10] U. Flemming, More than the sum of parts: the grammar of queen anne houses, Environment and planning B 14 (1987) 323–350.

[11] M. Pugliese, J. Cagan, Capturing a rebel: modeling the Harley-Davidson brand through a motorcycle shape grammar, Research in Engineering Design 13 (2002) 139–156.

[12] J. P. McCormack, J. Cagan, C. M. Vogel, Speaking the Buick language: capturing, understanding and exploring brand identity with shape grammars, Design Studies 25 (2004) 1–29.

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

[14] K. Shea, J. Cagan, Innovative dome design: Applying geodesic patterns with shape annealing, Artificial intelligence for engineering design, analysis and manufacturing 11 (1997) 379–394.

[15] K. Shea, J. Cagan, Languages and semantics of grammatical discrete structures, Artificial Intelligence for Engineering Design, Analysis and Manufacturing 13 (1999) 241–251.

[16] K. Shea, J. Cagan, 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 (1999) 3–23.

[17] J. S. Gero, S. J. Louis, S. Kundu, Evolutionary learning of novel grammars for design improvement, Artificial Intelligence for Engineering Design, Analysis and Manufacturing 8 (1994) 83–94.

[18] J. S. Gero, V. A. Kazakov, Evolving building blocks for design using genetic engineering: A formal approach, in: Advances in formal design mehotds for CAD, Hall, 1996, pp. 31–50.

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

[20] H. C. Lee, M. X. Tang, Evolving product form designs using parametric shape grammars integrated with genetic programming, Artificial Intelligence for Engineering Design, Analysis and Manufacturing 23 (2009) 131–158.

[21] O. Chouchoulas, Shape Evolution. An Algorithmic Method for Conceptual Architectural Design Combining Shape Grammars and Genetic Algorithms, Ph.D. thesis, University of Bath, 2003.

[22] M. O'Neill, J. McDermott, J. M. Swafford, J. Byrne, E. Hemberg, A. Brabazon, E. Shotton, C. McNally, M. Hemberg, Evolutionary design using grammatical evolution and shape grammars: designing a shelter, International Journal of Design Engineering 3 (2010).

[23] R. S. Sutton, A. G. Barto, Reinforcement learning: an introduction, MIT Press, Cambridge, Ma., 1998.

[24] J. P. Duarte, A discursive grammar for customizing mass housing: the case of Siza's houses at Malagueira, Automation in Construction 14 (2005) 265–275.

[25] H. C. Lee, M. X. Tang, Evolving product form design using parametric shape grammars integrated with genetic programming, Artificial Intelligence in Engineering Design, Analysis and Manufacturing 23 (2009) 131–158.

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

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

[28] K. Shea, C. Ertelt, T. Gmeiner, F. Ameri, Design-to-fabrication automation for the cognitive machine shop, Advanced Engineering Informatics 24 (2010) 251–268.

[29] G. Stiny, Pictorial and formal aspects of shape and shape grammars and aesthetic system, Birkhèauser (Basel and Stuttgart), 1975.

[30] P. G. Rowe, Design Thinking, Cambridge, Mass, MIT Press, 1987.

[31] S. C. Chase, A model for user interaction in grammar-based design systems, Automation in Construction 11 (2002) 161–172.

[32] T. W. Knight, Applications in architectural design, and education and practice, Technical Report, NSF/MIT Workshop on Shape Computation, 1999.

[33] R. Studer, V. R. Benjamins, D. Fensel, Knowledge engineering: Principles and methods, Data and Knowledge Engineering 25 (1998) 161–197.

[34] S. Kirkpatrick, C. D. Gelatt, M. P. Vecchi, Optimization by simulated annealing, Science, Number 4598, 13 May 1983 220, 4598 (1983) 671–680.

[35] C. de la Higuera, Grammatical Inference: Learning Automata and Grammars, Cambridge University Press, New York, NY, USA, 2010.

[36] P. Piggott, A. Sattar, Reinforcement learning of iterative behaviour with multiple sensors, Applied Intelligence 4 (1994) 351–365. 10.1007/BF00872474.

[37] M. J. Mataric, Reinforcement learning in the multi-robot domain, Autonomous Robots 4 (1997) 73–83. 10.1023/A:1008819414322.

[38] C. Gaskett, L. Fletcher, A. Zelinsky, 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.

[39] W. Smart, L. Pack Kaelbling, 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.

[40] R. Tedrake, T. Zhang, H. Seung, 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.

[41] P. Abbeel, A. Coates, M. Quigley, A. Y. Ng, An application of reinforcement learning to aerobatic helicopter flight, in: In Advances in Neural Information Processing Systems 19, MIT Press, 2007, p. 2007.

[42] G. Endo, J. Morimoto, T. Matsubara, J. Nakanishi, G. Cheng, Learning cpg-based biped locomotion with a policy gradient method: Application to a humanoid robot, Int. J. Rob. Res. 27 (2008) 213–228.

[43] M. S. Erden, K. Leblebicioglu, Free gait generation with reinforcement learning for a six-legged robot, Robotics and Autonomous Systems 56 (2008) 199 – 212.

[44] B. Nemec, M. Zorko, L. Zlajpah, 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.

[45] J. Peters, K. Mulling, J. Kober, D. Nguyen-Tuong, O. Kromer, Towards motor skill learning for robotics, in: C. Pradalier, R. Siegwart, G. Hirzinger (Eds.), Robotics Research, volume 70 of *Springer Tracts in Advanced Robotics*, Springer Berlin - Heidelberg, 2011, pp. 469–482.

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

[47] G. Tesauro, Temporal difference learning and td-gammon, Commun. ACM 38 (1995) 58–68.

[48] S. Thrun, Learning to play the game of chess, in: Advances in Neural Information Processing Systems 7, The MIT Press, 1995, pp. 1069–1076.

[49] D. P. Bertsekas, J. N. Tsitsiklis, Neuro-Dynamic Programming, Athena Scientific, 1996.

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

[51] N. Schraudolph, P. Dayan, T. Sejnowski, 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*, Springer Berlin / Heidelberg, 2001, pp. 77–98.

[52] R. Levinson, R. Weber, Chess neighborhoods, function combination, and reinforcement learning, in: T. Marsland, I. Frank (Eds.), Computers and Games, volume 2063 of *Lecture Notes in Computer Science*, Springer Berlin / Heidelberg, 2001, pp. 133–150.

[53] J. Schaeffer, M. Hlynka, V. Jussila, 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, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2001, pp. 529–534.

[54] J. Hu, M. P. Wellman, Nash q-learning for general-sum stochastic games, J. Mach. Learn. Res. 4 (2003) 1039–1069.

[55] S. Droste, J. Fürnkranz, Learning the piece values for three chess variants, International Computer Games Association Journal (2008) 209–233.

[56] S. Gelly, D. Silver, Achieving master level play in 9x9 computer go, in: Proceedings of the 23rd national conference on Artificial intelligence - Volume 3, AAAI'08, AAAI Press, 2008, pp. 1537–1540.

[57] C. Thiery, B. Scherrer, Building controllers for tetris (2009).

[58] J. Veness, D. Silver, W. T. B. Uther, A. Blair, Bootstrapping from game tree search, in: Y. Bengio, D. Schuurmans, J. D. Lafferty, C. K. I. Williams, A. Culotta (Eds.), NIPS, Curran Associates, Inc., 2009, pp. 1937–1945.

[59] I. Szita, G. Chaslot, P. Spronck, 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*, Springer Berlin / Heidelberg, 2010, pp. 21–32.

[60] W. Zhang, T. G. Dietterich, A reinforcement learning approach to job-shop scheduling, in: In Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence, Morgan Kaufmann, 1995, pp. 1114–1120.

[61] M. Aydin, E. Oztemel, Dynamic job-shop scheduling using reinforcement learning agents, Robotics and Autonomous Systems 33 (2000) 169 – 178.

[62] Y.-C. Wang, J. M. Usher, Application of reinforcement learning for agent-based production scheduling, Engineering Applications of Artificial Intelligence 18 (2005) 73 – 82.

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

[64] J. Peng, B. Bhanu, Closed-loop object recognition using reinforcement learning, Pattern Analysis and Machine Intelligence, IEEE Transactions on 20 (1998) 139 –154.

[65] L. Paletta, A. Pinz, Active object recognition by view integration and reinforcement learning, Robotics and Autonomous Systems 31 (2000) 71 – 86.

[66] J. Michels, A. Saxena, A. Y. Ng, High speed obstacle avoidance using monocular vision and reinforcement learning, in: Proceedings of the 22nd international conference on Machine learning, ICML '05, ACM, New York, NY, USA, 2005, pp. 593–600.

[67] S. Singh, D. Bertsekas, Reinforcement learning for dynamic channel allocation in cellular telephone systems, in: In Advances in Neural Information Processing Systems: Proceedings of the 1996 Conference, MIT Press, 1997, pp. 974–980.

[68] N. Lilith, K. Dogancay, 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.

[69] S.-M. Senouci, G. Pujoile, 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.

[70] Y. Zhao, M. R. Kosorok, D. Zeng, Reinforcement learning design for cancer clinical trials, Stat Med (2009).

[71] R. S. H. Istepanian, N. Y. Philip, M. G. Martini, Medical qos provision based on reinforcement learning in ultrasound streaming over 3.5g wireless systems, IEEE J.Sel. A. Commun. 27 (2009) 566–574.

[72] C. J. Watkins, Learning from delayed rewards, Ph.D. thesis, University of Cambridge, 1989.

[73] J. Abounadi, D. P. Bertsekas, V. Borkar, Stochastic Approximation for Non-Expansive Maps: Application to Q-Learning Algorithms, Technical Report, SIAM Journal on Control and Optimization, 2000.

[74] Montaner Muxí arquitectes, Propuesta de nueva normativa de viviendas, Technical Report, Dirección general de ordenación del territorio, Junta de Andalucía, 2008.

[75] Ruby Language, http://www.ruby-lang.org/, 2011.

[76] Trimble SketchUp, http://sketchup.google.com/intl/es/, 2011.

[77] S. Thrun, A. Schwartz, Issues in using function approximation for reinforcement learning, in: In Proceedings of the Fourth Connectionist Models Summer School, Erlbaum, 1993.

List of Figures