A Graphical Language for Generating Architectural Forms

Andrew Rau-Chaplin *Trevor J. Smedley*

Dalhousie University
Faculty of Computer Science
Halifax, Nova Scotia, Canada

Abstract

We describe a collaborative research effort between the faculties of Computer Science and Architecture, investigating a special-purpose visual language to be used to generate architectural forms. This language forms part of a larger system intended to automate the process of generating architecturally designed houses. The rules which can be specified using the language described here are used to prune a set of floor plans according to specifications provided by an architect. They are intended to replace rules written in Prolog, allowing the architect to directly specify and understand the rules.

1. Introduction

To converse with an Architect is to participate in a dialog in which a sketch book is filled with both representational images of structures and thought diagrams that express both symbolic and functional relationships. Software engineers also draw to communicate, but their diagrams are often more structured — seeking to express simpler relationships, but to have the expression more precise, complete and amenable to machine interpretation. This paper describes on-going work on a visual language intended to facilitate communication between Architects and Software Engineers involved in the LaHave House Project, as well as between the Software Engineers and their machines.

The *LaHave House Project* [8] is an ongoing research project of the Virtual House Lab of the Faculties of Architecture and Computer Science at the Technical University of Nova Scotia, Canada. The goal of the project is to explore the potential for an industrial design approach to architectural design in which Architects design families of similarly structured objects, rather than individual objects, thereby amortizing design costs. Currently in North America architects are involved in the design of only about 5% percent of the total new house market. Whereas custom architectural design will always have a premier role to play, we believe that an industrial design approach to architecture can bring much of the design quality and variety of custom design to the other 95% of the market, at an affordable price.

At the heart of the project is the use of a design space description formalism [1], based on shape grammars, [6, 9, 10], to specify and generate a design library (see Figure 1)

The role of the design engine in LaHave House Project is to generate a library of "base house designs" which are then used as a starting point for an end-user driven design development process. Given a shape grammar, this library is generated automatically, without human intervention. This differs from approaches such as co-operative CAD systems, such as that described in [3], which require human interaction throughout the design process.

One can think of this generated design library as the digital analog to the pattern books of the 19th century in that both contain house organizations, as much as they contain individual house designs. The generated design library contains houses that differ radically from each other in form, organization, size, amenity level, and "style", but despite this diversity share an underlying deep structure.

Each base house design or *house schema* consists of a complete description of the geometry of the house and the layout of each floor (see, for example, Figure 2). The design of individual spaces in the house, for example the layout of the kitchen or bathrooms, is not currently generated rather

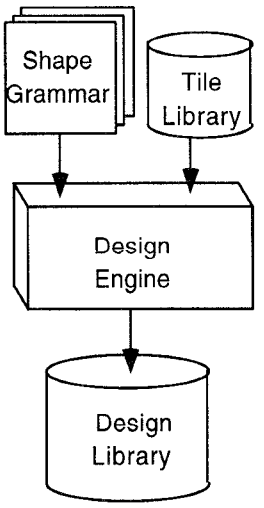


Figure 1 — The Design Engine

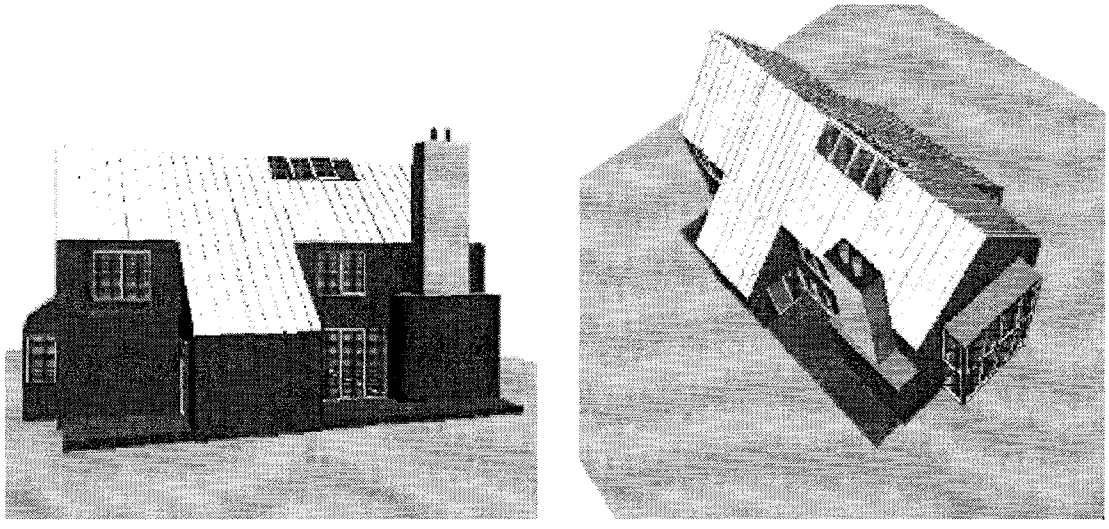


Figure 2 — One generated house from the design library, two views.

pre-designed room arrangements, called *tiles,* are allocated from a tile library. Our current tile library consists of over 800 room tiles and 450 wall tiles, while our generated design library contains approximately a 1,000,000 different house designs. Ascertaining the exact size of the design library is somewhat difficult as some of designs are near duplicates arrived at by different paths, but we believe that overall the library strikes a reasonable balance between variety and needless variation.

Currently, the "production version" of the Design Engine is written in Prolog and the shape grammars are specified as Prolog rules. In many ways Prolog has proved ideal for this application — its declarative nature and the ease with which generative processes can be expressed, given backtracking and unification mechanisms, recommend it highly. But over the last two years, as we have gained more experience in writing, maintaining, and extending shape grammars, we have become less satisfied with Prolog as a language for expressing generative rules, for the following reasons:

1. The project Architects find it nearly impossible to work directly with the shape grammars as expressed in Prolog. This leads to a very indirect iterative three stage development cycle in which
   a) Architects and Software Engineers meet, talk, and draw pictures, then
   b) the Software Engineers go away and try to encode the ideas captured in their notes and pictures into Prolog rules. Try to debug the rules (which maybe conceptually ill-formed) without the help of the Architect and generate typically several hundred pages of sample output.
   c) The Architects and Software Engineers meet again, look at the output and try to identify the ways in which the rules, as judged by the output, do and do not capture the ideas discussed in their last meeting. This leads them back to step a).

   Typically a single cycle takes at least a week to go through and typically has to be iterated between five and ten times to get a smallish grammar right. The Architects find it frustrating not being able to act directly on the rules. All parties often feel that they are not progressing, that they somehow lack "traction".
2. The same basic idea or constraints can be expressed in so many different ways in Prolog that it is often very difficult to tell if two rules are related to each other or are likely to interact. This greatly complicates the task of maintaining or extending grammars.
3. It is difficult to find good Software Engineers who are highly proficient in Prolog and have the necessary communication skills to do this kind of interdisciplinary work.

In finding a replacement for Prolog, we felt the need to design a special purpose language that, through its constructs, captured what we had learned about expressing generative systems in our context. We briefly considered using a graphical rewrite language like those found in the classical Shape Grammar literature, [6, 9, 10], but quickly realized that our design language was much richer than the type of 2D language of points, lines and labels that the phrase 'Shape Grammar' originally denoted.

Our overriding need for a shared language allowing the domain experts (Architects) and Software Engineers to directly apply their specialized domain expertise has lead us to design our own single purpose visual programming language. In the remainder of this paper we will first describe our generative system, and then describe how we

represent the rules graphically, giving examples. We conclude with a brief summary and pointers to future work.

2. Generating House Designs

Although we are interested in the general question of how arbitrary shape grammars might be visually expressed, we are at this stage using a large single shape grammar to drive our work. In this section we briefly describe the LaHave House Grammar and discuss some pragmatic issues relating to the balance between generation and pruning strategies.

2.1 The LaHave House Grammar

The LaHave House grammar is based on a systematic approach to house design developed by Brian MacKay-Lyons, an award-winning Canadian Architect, in over fifteen years of custom architectural design practice. Reviews and discussion of his design work can be found in [2, 5]. The forms generated by the grammar have been inspired by the vernacular architecture of the LaHave river valley of Nova Scotia, Canada. Using shape grammars to capture the design space of a living architect is in many ways different to much of the existing shape grammar work [4, 9]. The goal is not so much to create a grammar that generates an existing corpus of design, but rather to work with the Architect to extract from their existing design corpus a robust set of generation principles.

One feature of the grammar is its tendency to produce dense cores for services and sparse open spaces for living in. The grammar is constructed in terms of a set of five elementary components: Rooms, Tartans, Machines, Bays, and Totems. The rooms are the principle places for human action, the tartans provide space for circulation, the machines (bathrooms, kitchens, laundry, entry etc.) are the dense service spaces, the Bays provide outlook space and secondary living space, and the totems (hearths, staircases, cabinetry etc.) provide focus for the rooms.

The grammar makes extensive use of the idea of Served vs. Servant spaces advanced in [7]. This tends to be a three level hierarchy in which rooms are served by primary machines, which may in turn be served by secondary machines. For example, a dinning room may be served by a kitchen, which is in turn served by a pantry. Although we are interested in generating complete functional modern houses, (i.e. we care where the bathrooms and closets go), the grammar is driven purely by issues of form. The function plan is derived only when the form has been completely generated.

The grammar we are currently working with has a single longitudinal axis of growth, with a dense machine zone flanking a body zone, which is in turn surrounded by a bay zone (see Figure 3). The machine zone tends to be contiguous and completely filled, particularly in smaller houses, while the bay zone tends to be more sparsely populated.

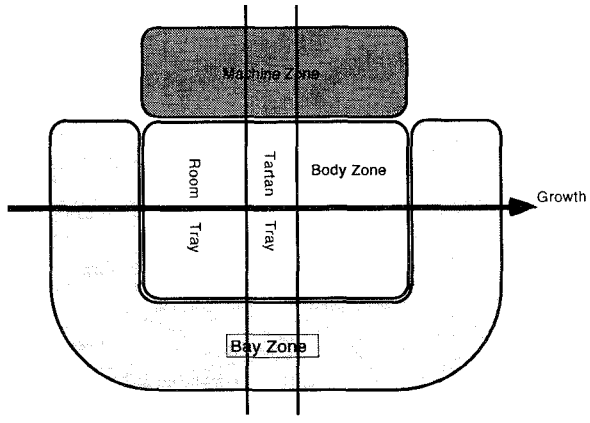


Figure 3 — Space Zoning

2.2 A Hierarchy of Shape Grammars

The LaHave House grammar is expressed as a hierarchical set of shape grammars (see Figure 4) in which each level of design schema (example instances are shown on the right) are generated by combining lower level design schema. At the lowest level there are atomic elements representing machines, rooms, bays and end-bumps. The machines, bays and end-bumps come in a variety of sizes, some of which contain totems. The rooms also come in a variety of sizes (tartans beginning classified as narrow rooms) and may be either "open" (i.e. primarily room) or "dense" (i.e. mostly occupied by machine). These atomic elements are combined together by the Tray Generation grammar to form a set of Tray Schema. A tray is a slice from a floor plan, as shown in 4(a). The Plan Generation grammar then combines trays to form valid Plan Schema, such as 4(b). Functional Zoning takes Plans and forms Zoned Plans in which the public end of the plan is identified, any double height space is labeled and the "great room" (i.e. large public open plan area) is identified 4(c). Section Schema are, like Plan Schema, built from atomic parts. They describe the primary section of a house (i.e. the number of floors, widths of each zone, and all of the roof shapes) 4(d). The Form Generation grammar takes compatible plan and section schema and combines them to create complete 3D forms 4(e). Associated with this step is a lot of procedural house-keeping in which floor plans are created from the plan schema, these plans are dimensioned, creating walls to define/enclose the interior spaces, roofs are added and generally all of the 3D information required to define a complete form is computed. Finally, form schema are transformed into completed house designs (house schema) 4(f) by a series of grammars that first assigns to each space unit within the form schema a function (i.e. kitchen, bedroom, entry, study etc.) and then assigns a room organizations from a library of room

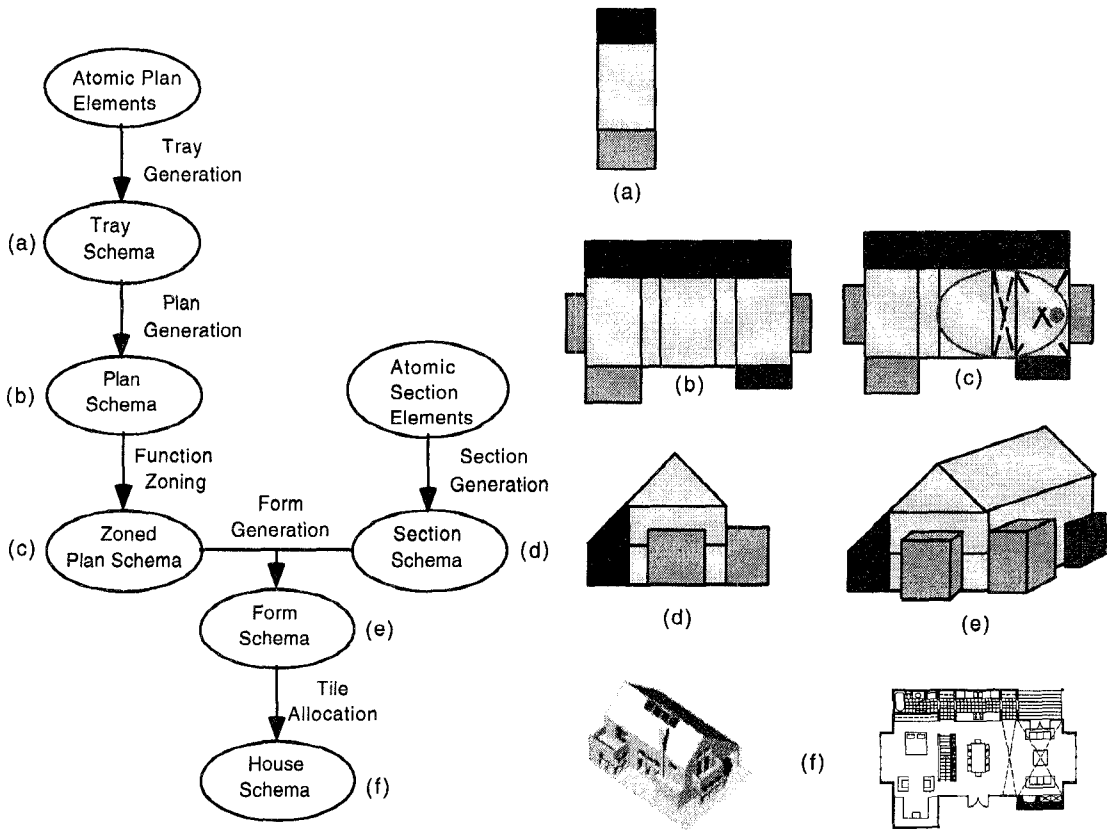


Figure 4 — Phases of Generation

level designs. We call these pre-designed room organization tiles and the problem of assigning tiles to space units, in a formally pleasing and functional arrangement, the tile assignment problem. Note that by the time one reaches the house schema stage one has a rich representation of a house, complete with plan, section and 3D form information.

2.3 Generation vs. Pruning Rules

There are two obvious approaches to generating valid schema. The first is to generate all possible combinations of all input elements, and then prune away all those combinations which prove invalid using a system of pruning rules. The second is to generate only valid schema. The basic advantage of this the first approach is simplicity. The generation rules are relatively simple and regular and each completed production can be validated by a equally simple and regular set of rules. But there are two primary disadvantage to this approach. Firstly, it is unlikely that one has enough time or space to generate all possible configurations before testing for validity. Secondly, if a generated schema is invalid for more than one reason, (i.e. it has features A, B, and C which are each invalid by themselves) then it is possible (and as it turns out quite likely) that the combination of invalid features are jointly consistent enough to slip past the validation rules.

The second approach, that of generating only valid schema, relies on one being able to write the generation rules so craftily that invalid schema are never generated. Unfortunately, this approach is impossible to realize for anything but very trivial classes of productions. There are just too many design issues to be considered at every choice point to make this approach workable.

We have found that only by combining these two approaches can one derive correct, efficient and robust grammars. Each of the grammars is therefore expressed in two parts: A set of generation rules that aim to be simple, but try to exclude combinations that can be easily recognized as being invalid and a set of pruning rules that capture more subtle invalidation criteria.

3. A New Visual Language

In approaching the problem of designing visual methods for expressing the rules involved in the LaHave House Project shape grammars, we decided to initially focus on the bottom of the hierarchy — the shape grammars used for Plan Schema and Zoned Plan Schema. In the current system Plan Schema and Zoned Plan Schema are generated and then pruned using rules written in Prolog. Figure

```
%prune any plan without a fat totem, and too few machine zones
prune_plan(Trays)  :-
      not(member(T,Trays),fetch(T,width,main),fetch(T,body_zone,dense)),
      flag(num_main_trays,NMT,NMT),
      flag(num_12_mz,NMZ,NMX),
      NMZ < NMT - 1.
```

Figure 5 — Prolog Rule (Same as Figure 8c)

5 shows and example of such a rule. We describe here a language which can be used to express rules for matching plans. Clearly these matching rules can be used in the definition of both generation rules and pruning rules. In the remainder we focus on pruning

We first give definitions of plans and their parts, and then define the rules that can be described using the language and how they can be 'matched' with plans. We then give examples of the graphical representations of the rules.

3.1 Definitions

Plan. A Plan is a sequence of trays.

Tray. A Tray has a width, height (single or double), booleans for *public end* and *great hall,* and a Machine Zone, Room Zone, and Bay Zone.

Machine Zone, Room Zone, and Bay Zone. Each of these has a depth and a fill specifier (OPEN, DENSE or EMPTY).

Rule. A Rule can be an AND Rule, OR Rule, NOT Rule, Symmetric OR Rule, Neighbour Rule, Quantity Rule, or Distance Rule. It contains a list of rule parts. A Plan is matched by a Rule if any contiguous subsequence of trays matches the Rule.

Rule Part. A Rule, a Tray Specifier or an End Specifier.

AND Rule. A list of Rule Parts, all of which must be matched in order for the rule to be matched.

OR Rule. A list of Rule Parts, any one of which must be matched in order for the rule to be matched.

NOT Rule. A Rule Part, which must not be matched in order for the rule to be matched.

Symmetric OR Rule. A Neighbour Rule which behaves as a two-element OR Rule. The first element is the Neighbour Rule specified, and the second element is the Neighbour Rule with its list of rule parts reversed. Note that this rule is not required (it can be expressed using an OR Rule), but it is a common enough occurrence that it is worth integrating into the language.

Neighbour Rule. A list of Rule Parts, $(p_1, p_2, \ldots, p_n)$, which is matched iff there is a contiguous sequence of trays, $(t_1, t_2, \ldots, t_m)$, and a monotonically increasing sequence of integers, i_j; $j = 1 \ldots l$, such that $(t_1, \ldots, t_{i_1})$ matches p_1; $(t_{i_k+1}, \ldots, t_{i_{k+1}})$ matches p_i for $j = 1 \ldots l$, and $(t_{i_l}, \ldots, t_m)$ matches p_n.

Quantity Rule. A single Rule Part, and a boolean expression in one variable, $E(x)$. Matched iff there are n trays which satisfy the Rule Part, and $E(n)$ is TRUE.

Distance Rule. Three Rule Parts: Source; Separator; and Destination. The Source and Destination can be any Rule Part, but the Separator must be a Quantity Rule. This rule is matched iff, for every tray sequence satisfying the Separator Rule Part, and every tray sequence satisfying the Destination Rule part, the sequence of trays which separate them must satisfy the Separator Rule Part. Although it is possible to build such a rule up from AND, OR, NOT and Quantity Rules, it is exceedingly complicated, and rules of this form are common enough to warrant special inclusion.

Tray Specifier. A tray width specifier (a valid tray width or the value ANY), a height specifier (SINGLE, DOUBLE or ANY), a public end specifier (PUBLIC, NOT PUBLIC or ANY), a great room specifier (GREAT ROOM, NOT GREAT ROOM, or ANY), a Machine Specifier, a Room Specifier, and a Bay Specifier. A Tray Specifier is matched by any tray which matches the width, public end, and great hall of the specifier (ANY matches with any value), and whose Machine, Room and Bay Zones satisfy the Machine Specifier, Room Specifier, and Bay Specifier respectively.

Machine Specifier, Room Specifier, and Bay Specifier. A depth specifier (a valid part depth or the value ANY), and a fill specifier (OPEN, DENSE, EMPTY or ANY). A Machine, Room or Bay Zone matches a Machine Specifier, Room Specifier, or Bay Specifier (respectively) if their depths and fills match (again, ANY matches any value).

End Specifier. The value BEGINNING or END. These can

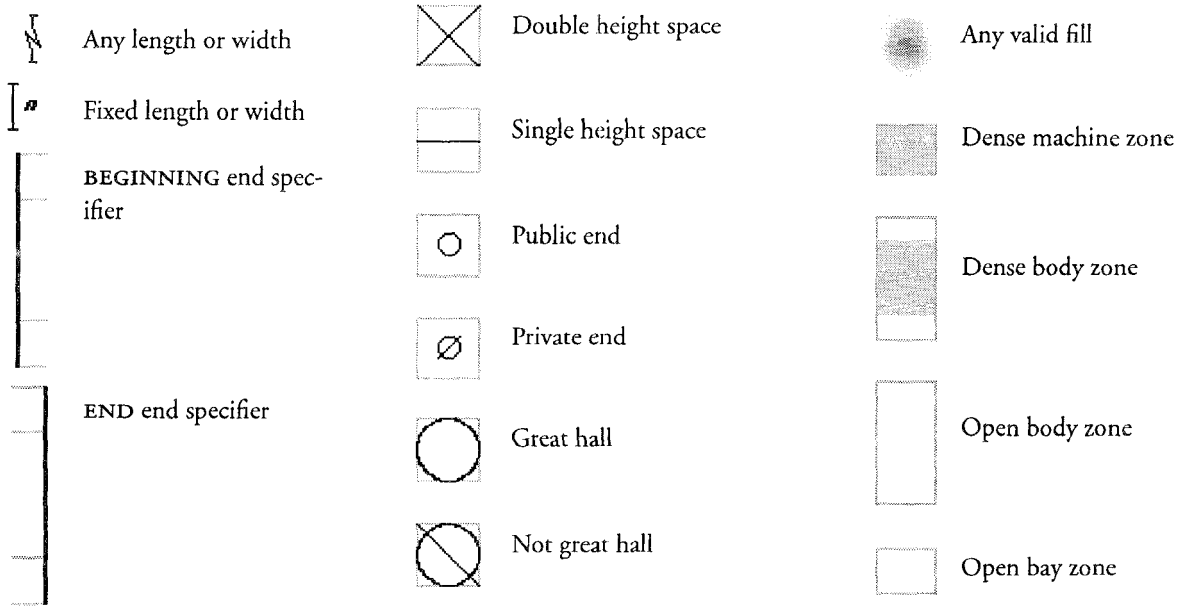
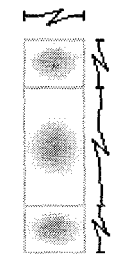
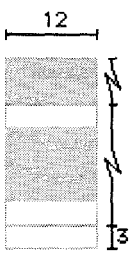


Figure 6 — Graphical Vocabulary



Matches any tray at all. None of the widths are specified, all three zones are designated 'any valid fill', and there are no specifiers for height, public end, or great hall.

Matches trays of any width with empty machine and bay zones, and an open, public end, double height, great hall, body zone of any depth.



Matches any 12' wide tray, with a dense machine zone of any depth, a dense body zone of any depth, and a 3' deep open bay zone.

Matches 4' wide trays, with dense machine zones and any body or bay zones.

Figure 7 — Example tray specifiers

only be used as part of a Neighbour Rule: BEGINNING can only be used as Rule Part p_1 ; END can only be used as Rule Part p_n as defined above. If p_1 is BEGINNING, then p_2 must match a sequence of trays which includes the first tray in the plan. If p_n is END, then p_{n-1} must match a sequence of trays which includes the last tray in the plan.

3.2 Graphical Depictions

We present here the graphical representations of the rules defined in the previous section. We start off with the basic graphical vocabulary, and then present examples of the rules.

Figure 6 shows the symbols used to give tray specifications. There is no symbol for open machine zone, empty room zone, or dense bay zone, as these are illegal. Also, the ANY specifications for height, public end and great hall are given by the absence of a symbol. Using this basic vocabulary, we can build up tray specifiers such as those shown in Figure 7.

Rules are drawn inside boxes, with annotations indicating what type of rule it is. For AND rules, the rule parts are drawn horizontally, separated by a dashed line, as shown in Figure 8c). OR Rules are done similarly, but vertically instead of horizontally, as shown in Figure 8d). The vertical organisation is meant to visually suggest a choice, and a horizontal row to indicate that all the parts are required. A NOT Rule is shown in Figure 8c).

A Symmetric OR Rule, Figure 8b), looks like an OR Rule except it uses the symbol OR, and when creating or editing such a rule, only the top half can be modified, and the lower half is generated automatically. A Neighbour Rule is shown in Figure 8a), with the symbol . When a neighbour rule is used inside another rule, the box and symbol are omitted to simplify the diagrams.

A Quantity Rule is shown on the right in Figure 8c). For these, the expression to be satisfied is indicated along the top of the box. Finally, a Distance Rule is show in Figure 8d). The three parts are in separate sections, with the symbol above the Source part, and the expression for the Distance Rule above the Separator part. The arrows on the box sides are intended to indicate the Source/Destination relationship.

Rule Explanations

Figure 8a). Neighbour Rule, which matches any plan which has two consecutive trays of width 4.

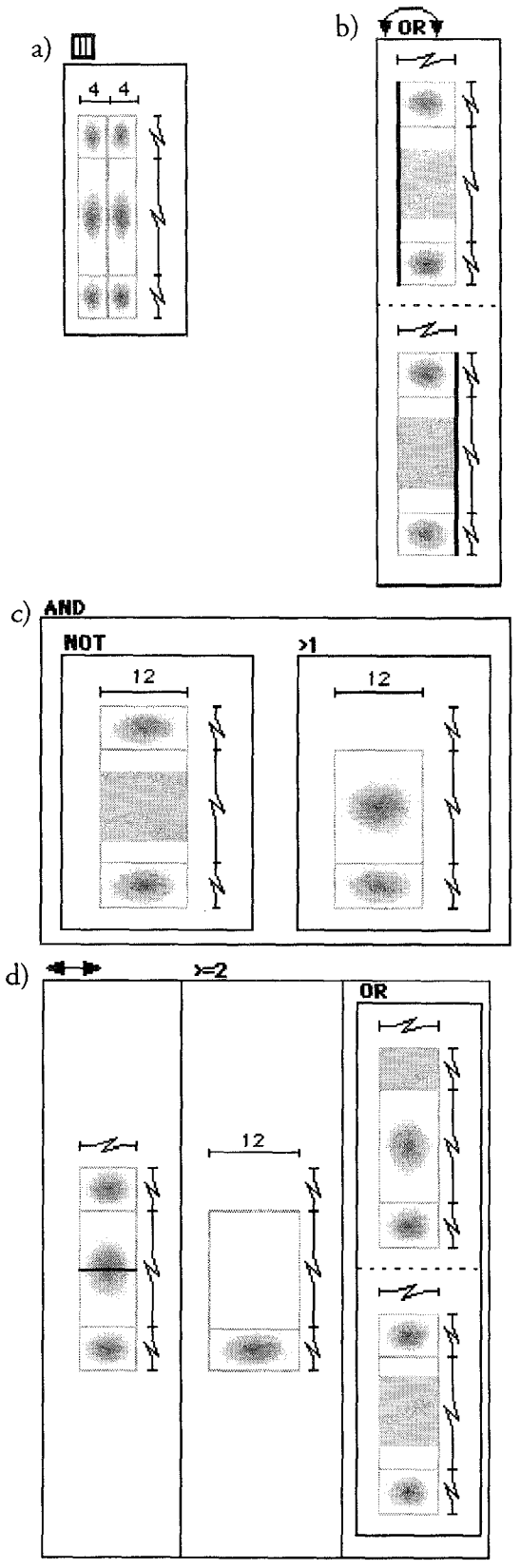


Figure 8 — Sample Rules

Figure 8b). This is a Symmetric OR Rule, which matches any tray with a dense body at either the beginning or end of the plan.

Figure 8c). An AND Rule with two parts, the first a NOT Rule, and the second a Quantity Rule. This matches any plan which has no 12' wide trays with a dens body, and more than one 12' wide tray with an empty machine zone.

Figure 8d). A Distance Rule, where the Source matches any tray with a single height body, the Separator matches any 12' tray with empty machine and an open body, and the Destination is an OR rule which matches any tray with either a dense machine zone, or a dense body zone. Thus, this rule matches any tray with single height space that is two or more 12' trays away from the nearest dense space.

3.3 Applying Rules

As mentioned earlier, the implementation for this project has been carried out primarily using Prolog, and rules such as the ones given in Figure 8 have been implemented for pruning zoned plan schema. In order to integrate this work with the current implementation, Prolog rules are generated from the visual descriptions explained here. The translation to Prolog is straightforward, but tedious, and we omit its description.

4. Concluding Remarks

We have described a visual language for expressing matching rules that can be used to both generate and prune plan schema. We are currently prototyping an environment which would allow Architects, either alone or in conjunction with Software Engineers, to enter matching rules. This environment will provide powerful rule editing features, and allow for a much shorter and more direct development cycle for our shape grammars. It will export Prolog rules to be used in conjunction with the existing code. In the future we expect to extend this tool to allows to directly specify both pruning and generation rules, perhaps extending the language to be used for other grammars in the hierarchy. Having Architects use this system directly will provide us with valuable feedback about our visual representations, and allow us to improve them as required.

The concepts embodied in our visual representations are those which Architects wish to deal with directly. They capture abstract notions such as those of functional zoning, scale, sparse vs. dense forms, etc. in way that simple graphical rewrite rules are unable to. We expect this language, and our continuing work in this area, to have a significant impact on the LaHave House Project as a whole.

5. Acknowledgments

We would like to thank the TUNS Faculty of Architecture and School of Computer Science for their support and acknowledge the key role P. Spierenburg has played in the creating of the Prolog design engine. The work of the following other individuals is also acknowledged:

Architects: B. MacKay-Lyons, J. Smirnis, D. Wigle, E. Jannasech, N. Savagen, P. McClelland

Computer Scientists: H. Ning, T. Doucette, X. Hu, G. Li, D. Gemmell, A. Gajewski, D. Curry, D. Peters, G. Burrell, H. Lui, S. Gauvin, W. Wu

6. References

[1] C. Carson. "Shape Grammars in Design: Discussion. Design Space Description Formalisms", Formal Methods for CAD, 1994, Editors J.S. Gero and E. Tyugu., pp. 121-130.

[2] T. Fisher, "Folk Tech", Progressive Architecture, August 1995, pp. 63-72.

[3] M. Friedell and S. Kochhar, "Design and Modeling with Schema Grammars", Journal of Visual Languages and Computing, 1991, Vol. 2, pp. 247-273.

[4] H Koning and J Eizenberg, "The Language of the Prairie: Frank Lloyd Wright's Prairie Houses", Environment and Planning B: Planning and Design 8, 1981, pp. 295-323.

[5] B. MacKay-Lyons, "The Village Architect". Design Quarterly 165, MIT Press, Editor R. Jensen, Summer 1996.

[6] W. Mitchell, "The Logic of Architecture". MIT Press, 1990.

[7] C. Moore, G. Allen and D. Lyndon, "The Place of Houses", Holt, Reinhart and Winston publishers, 1974.

[8] A. Rau-Chaplin, B. MacKay-Lyons, P. Spierenburg, "The LaHave House Project: Towards an Automated Architectural Design Service", in *Proc. of the International Conference on Computer-Aided Design (CADEX'96)*, IEEE Computer Society Press, Sept. 1996, pp. 25-31.

[9] G. Stiny and W. Mitchell, "The Palladian Grammar", Environment and Planning B: Planning and Design 5, 1978, pp. 5-18.

[10] G. Stiny, "Introduction to Shape Grammars", Environment and Planning B: Planning and Design 7, 1980, pp. 343-351.