SGML: A Meta-Language for Shape Grammar

by

Haldane Liew

M. Arch, Massachusetts Institute of Technology, 1996
B.A., University of California, Berkeley, 1992

Submitted to the Department of Architecture
in partial fulfillment of the requirements for the degree of
Doctor of Philosophy in Architecture: Design and Computation
at the Massachusetts Institute of Technology

September 2004

Signature of Author: ______________________________
Department of Architecture
July 5, 2004

Certified by: ______________________________
Takehiko Nagakura
Associate Professor of Design and Computation
Thesis Supervisor

Accepted by: ______________________________
Stanford Anderson
Professor of History and Architecture
Chairman, Committee for Graduate Students

Dissertation Committee

Takehiko Nagakura
Associate Professor of Design and Computation

George Stiny
Professor of Design and Computation

Patrick H. Winston
Professor of Computer Science

SGML: A Meta-Language for Shape Grammar

by

Haldane Liew

Submitted to the Department of Architecture on July 5, 2004 in
partial fulfillment of the requirements for the degree of
Doctor of Philosophy in Architecture: Design and Computation.

Abstract

A shape grammar develops a drawing through a series of transformations by repeatedly applying if-then rules. Although the rules can be designed, in principle, to construct any type of drawing, the drawings they construct may not necessarily develop in the manner intended by the designer of the grammar.

In this thesis, I introduce a shape grammar meta-language that adds power to grammars based on the shape grammar language. Using the shape grammar meta-language, the author of a grammar can: (1) explicitly determine the sequence in which a set of rules is applied; (2) restrict rule application through a filtering process; and (3) use context to guide the rule matching process, all of which provide a guided design experience for the user of the grammar.

Three example grammars demonstrate the effectiveness of the meta-language. The first example is the Bilateral Grid grammar which demonstrates how the meta-language facilitates the development of grammars that offer users multiple design choices. The second grammar is the Hexagon Path grammar which demonstrates how the meta-language is useful in contexts other than architectural design. The third and most ambitious example is the Durand grammar which embodies the floor plan design process described in *Précis of the Lectures of Architecture,* written by JNL Durand, an eighteenth century architectural educator. Durand's floor plan design process develops a plan through a series of transformations from grid to axis to parti to wall. The corresponding Durand grammar, which consists of 74 rules and 15 macros organized into eight stages, captures Durand's ideas and fills in gaps in Durand's description of his process.

A key contribution of this thesis is the seven descriptors that constitute the meta-language. The descriptors are used in grammar rules: (1) to organize a set of rules for the user to choose from; (2) to group together a series of rules; (3) to filter information in a drawing; (4) to constrain where a rule can apply; and (5) to control how a rule is applied. The end result is a language that allows the author to create grammars that guide users by carefully controlling the design process in the manner intended by the author.

Thesis Supervisor: Takehiko Nagakura
Title: Associate Professor of Design and Computation

Acknowledgements

I would like to thank my parents, Fah Seong and Polly, for all their love, encouragement, and support during my years at MIT.

I would like to thank my committee for their guidance and insightful conversations.

But most of all, I would like to thank my wife, Susan, for her patience and love. Without her, this book would not have been possible.

Table of Contents

1.0 Introduction

Shape grammar is a general computational language that manipulates shapes to generate designs. It has been used in the architectural field to develop designs such as Palladian villas (Stiny and Mitchell 1978), Prairie houses (Koning and Eizenburg 1981), and even Chinese government buildings based on the Yingzao Fashi book (Li 2001). But the use of shape grammars is not limited to the architectural field. It has also been used for designs such as window lattice designs in the Iceray grammar (Stiny 1977), Froebel block arrangements in the Kindergarten grammar (Stiny 1980), and coffee makers in the Coffee maker grammar (Agarwal and Cagan 1998). Each grammar generates a design by using a particular technique of the shape grammar language.

For the Iceray grammar, a sub-division technique is used to generate the design. Each polygon is divided to create more polygons which recursively gets divided. The Kindergarten grammar uses labeled points to generate distinctive arrangements of Froebel blocks. The labeled points use the symmetry of the blocks to control how the blocks can be assembled. The Yingzao Fashi grammar uses parallelism as its technique for generating a design. The parallel descriptions allow the design of the section and elevation to be in correspondence with the floor plan.

Shape grammar applies if-then rules composed of schemata to affect changes in a design. A grammar expresses the process of design through a series of schema rules. The rules are used to convey the design transformations used by the designer. A problem occurs when the designer can not express his design transformation in a rule explicitly and concisely with the shape grammar language. Although shape grammars can produce the resultant design, the manner in which that design is produced may not be the same. To resolve this problem, there are two choices. Either alter the designer’s process to fit within the descriptive means of shape grammars or define new descriptors to alter shape grammars to accommodate the designer’s process.

This thesis introduces seven new descriptors for the shape grammar language which provides an alternative method to write grammars for design. The new descriptors form a meta-language that empowers the author of a grammar to: (1) explicitly determine the sequence in which a set of rules is applied; (2) restrict rule application through a filtering process; and (3) use context to guide the rule matching process. The descriptors in the shape grammar meta-language (SGML) modify the conditions surrounding the process of applying a rule in shape grammars. These conditions are rule selection, drawing state, matching conditions, and application method.

1.1 Motivation

The new descriptors are based on the existing shape grammar language which is general enough to allow for different types of customizations. This is supported by Ulrich Flemming (1994) who writes,

> "My own work with shape grammars and related mechanisms has demonstrated to me that they offer, in principle, a particularly rich palette of customization possibilities."

SGML shows one such customization of the shape grammar language. The motivation for the work is based on my experiences with writing and using shape grammars for design. I developed the shape grammar meta-language because I wanted to be able to express a rule the way I see it as opposed to translating the rule into the technical mechanisms of the language. Although it is possible to describe my rules by using the descriptors in the language, the result is often less than desired because the rules can become overly complex and distract the user from the overall design. But more importantly, the rules do not emphasize what I want the user to see. The meta-language definitions allow me to describe, in my own methods, how I see the design transformations. Just like a Chinese artisan can image himself assembling an iceray in the same manner as the Iceray grammar, I want to be able to say that I can imagine myself applying my rules when designing.

The SGML also addresses the lack of a consistent method to communicate with the user what options are available when generating a design. Often times, I have had to search the entire grammar to find rules that can affect the appropriate changes. This can be a time consuming and frustrating endeavor. The users experience in using shape grammars can be greatly improved by providing a means to clearly show which rules can apply in a given situation. Some computer implementations of shape grammars have already begun to address this issue (Li 2002).

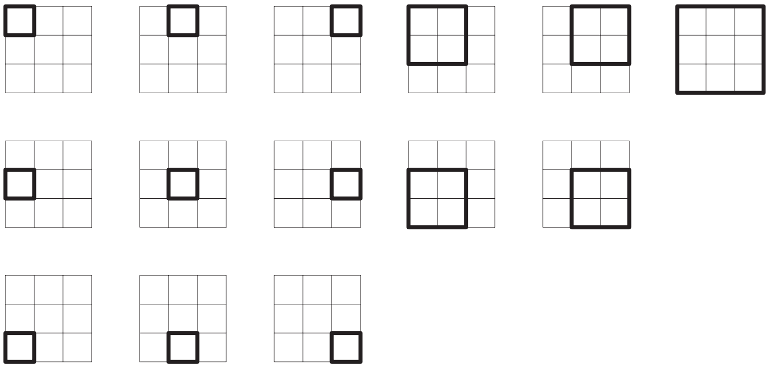
1.2 Problems using shape grammars for design

The goal of the descriptors is to make a more designerly grammar (Li 2001). Instead of making grammars that are mechanistic in nature, a designerly grammar acts as a guide to walk the user through the process used to generate the design. The grammar should be robust and have rules that encourage the user to explore the design space. Unfortunately not all grammars are designed in that manner. Some grammars permit the user to develop a dead-end state (Liew 2002). In this state, no subsequent rule application will bring the grammar to its conclusion. Designing a grammar that comes to an abrupt end inadvertently is not fruitful for exploring the design space.

Another problem is that many grammars treat all the rules the same. But not all rules have the same weight in a grammar. During the derivation of a grammar, some rules may affect the design greatly and some may not affect the design at all. These differences should be made obvious to the user. The rules made available to the user should always be salient design options.

The issues mentioned so far deal with the sequencing and availability of rules in a grammar. Another issue is concerned with defining the appropriate schema to find a match in the drawing. The key to applying a rule in shape grammars is the left-hand schema and its associated transformations and parameters. If the left-hand schema is a part of the drawing, then the rule applies. Otherwise the rule does not. It is therefore important that the left-hand schema is able to describe the necessary conditions for a match in the drawing.

In the shape grammar language, a schema is considered a match if the shapes are a part of or embedded in the drawing. This embedding relation gives shape grammars the powerful ability to find emergent shapes in a drawing. Figure 1.01 enumerates all 14 different possible squares that are embedded in a 3x3 grid. A schema that looks for squares in the shape grammar language can find all 14 possible squares.



1.01 The 14 different possible squares in the 3x3 grid.

The difficulty arises when the schema must match a specific square type. For instance, suppose we want a schema that matches only the four corner squares of the grid. Those four squares are a subset of the 14 possible squares in the 3x3 grid. The typical means of matching those four corner squares is by labeling them using labeled points.

An alternative strategy is to describe the condition that makes the corner square a corner square. What visual quality of the corner square distinguishes it from the other 14 squares? In this particular case, it is the fact that one corner of the square does not have any protruding shapes. A straightforward solution is to describe a schema that looks for a square and specify that one corner of the square must be devoid of any extraneous shapes. This is contextual information that distinguishes the corner square from the other squares. SGML provides a means to describe such a condition.

When looking for a schema in a drawing, typically the entire drawing is searched. But this may not always be the best scenario. For instance, the author of a grammar may want the user to concentrate on only a particular portion of the drawing. If that is the case, there should be some means to restrict where a rule can apply. Traditionally this has been done by using labeled points and lines to mark where the rule can apply.

With this approach, every rule that applies in a specific area must have the same set of labeled points or lines. Should that demarcation change during the development of the grammar, all the rules that apply in that area must also change to match the new labeled points or lines. This occurs because there is no separation between labeled shapes that are a part of the drawing and labeled shapes that dictate where a rule can apply. SGML resolves this issue by developing a separate mechanism that dictates the specific areas where a rule can apply independent of labeled shapes.

In addition to controlling the areas where a rule can apply, the author may also wish to restrict which shapes are available for matching. Typically all the elements of a drawing are used to search for a match. In order to match only certain types of labeled shapes, there needs to be a means to filter out information in a drawing. For instance, a particular stage of a grammar may affect only the walls in the design. In this case, the rules should look at only the wall labeled shapes and ignore the other labeled shapes in the drawing. SGML provides a descriptor to filter out labeled shapes in a drawing.

The three main category of problems addressed are: (1) controlling rule selection and sequencing in a grammar; (2) filtering out information in a drawing for rule application; (3) and specifying contextual requirements of a schema. The shape grammar meta-language addresses these issues by developing a set of descriptors to use in conjunction with the shape grammar language.

1.3 Shape grammar meta-language

The descriptors in SGML are based on the rule application process in shape grammars. This process is composed of four major phases: rule selection, drawing state, matching conditions, and application method. The matching condition phase can be expanded to include parameter requirements, transformation requirements, and contextual requirements. This expands the rule application process into six phases: rule selection, drawing state, parameter requirements, transformation requirements, contextual requirements, and application method.

The first step in applying a rule is to determine which rule to apply. The rule selection phase determines what rule to apply next and which rules are available for use at any point in the grammar. This phase has two descriptors: directive and rule-set. The directive controls what rule should be applied next based upon the success or failure of the given rule to apply in the drawing. The rule-set descriptor determines what set of rules are available to the user at any point during the derivation of the grammar.

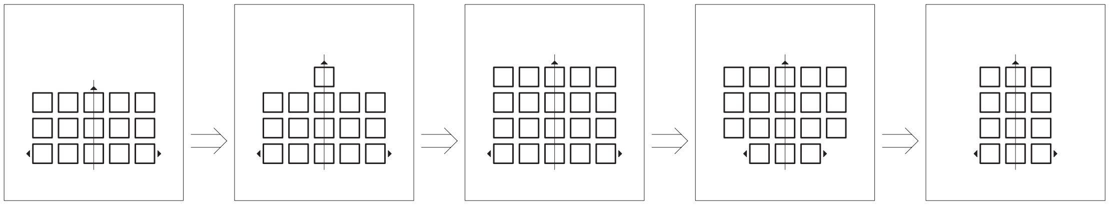
Once a rule is selected, the user has to determine where to apply the rule. The drawing state phase determines what portions of the drawing can be used for rule application. The two descriptors in this phase are label-filter and focus. Label-filter controls which labeled shapes are applicable based on the labeled shapes of the left-hand schema of a rule. The labeled shapes in the drawing that are not part of the left-hand schema are temporarily removed. The focus descriptor restricts the application of a rule to specific areas of the drawing demarcated by a special focus labeled polygon. A rule can only apply to the areas inside the polygons.

The next three phases, parameter requirements, transformation requirements, and contextual requirements, combine together to determine the matching conditions for finding a subshape in the drawing. In order for there to be a match, the requirements of all three phases must be fulfilled. The parameter requirements phase determines what values are assigned to the parameters of a schema in order to match a subshape in the drawing. The transformation requirements phase determines what combinations of transformations are necessary to match a subshape in the drawing. There are no meta-language descriptors in these two phases.

The contextual requirements phase adds an additional constraint that determines if the context of the subshape satisfies a predicate. This phase has two descriptors: maxline and zone. The maxline descriptor specifies that a line in the subshape has to be a maximal line in the drawing. The zone descriptor associates an area of the subshape with a predicate function which can be used to test portions of a drawing relative to the subshape. A commonly used predicate is the void function which tests if the area relative to the subshape is void of all shapes.

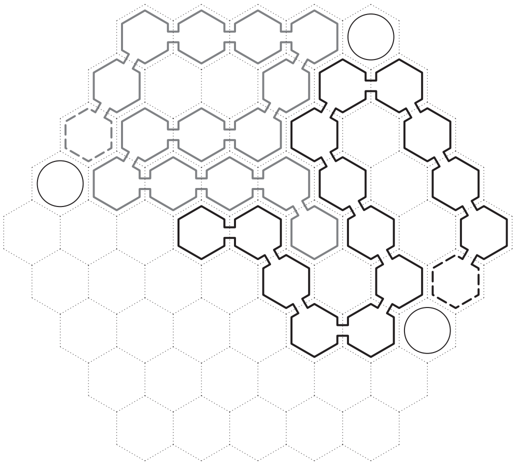
The application method phase is the last phase in the rule application process. This phase determines how a set of subshapes can be applied to the drawing. Typically, the user applies the rule to one selected subshape in the drawing. But there are other possibilities such as a parallel application where the rule applies to all the subshapes in a drawing. The only descriptor in the application method phase is apply-mode.

Three examples demonstrate how the seven descriptors in the shape grammar meta-language can be used in a variety of ways to generate designs with shape grammars. The first example is the Bilateral Grid grammar (figure 1.02) which produces a bilaterally symmetrical grid identical to the one generated from the first stage of the Palladian grammar. The Bilateral Grid grammar demonstrates how the meta-language descriptors can be used to redesign an existing grammar so that the user is presented with a salient set of design options and can no longer generates dead-end states.



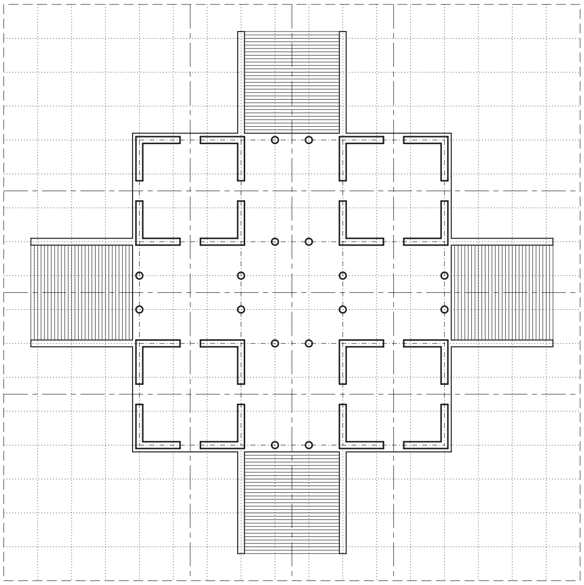
1.02 A sample derivation of the Bilateral Grid grammar.

The second example is the Hexagon Path grammar (figure 1.03) which demonstrates how a grammar can be used as a simple board game. The object of the game is to prevent your opponent from making a move by designing a path that surrounds your opponent. The first player unable to place a path within the playing field loses. The grammar shows how the directive and rule-set descriptors can control the sequencing of rules and how the zone descriptor can be used for collision detection.



1.03 A sample design using the Hexagon Path grammar.

The third example is the Durand grammar (figure 1.04) which demonstrates how the meta-language descriptors can be used to generate architectural floor plans. The grammar is based on *Précis of the Lectures on Architecture* written by the late 18th century architect, Jean Nicolas Louis Durand who taught architecture at the École Polytechnique. Unlike architectural educators of his time, who treated architecture as part of the arts, Durand treated architecture as part of science and describes a systematic method of generating architecture. The grammar is an interpretation of the design process presented in his book.



1.04 A sample design using the Durand grammar.

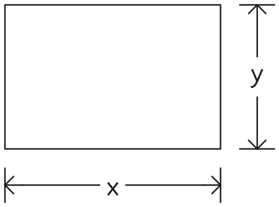
Durand’s process begins with the placement of axis lines on a grid. The next step is to develop a parti based on the axis lines. The parti also acts as the centerline for the construction of the walls. Once the walls are made, other architectural elements such as columns and stairs are placed in relationship to the walls and in alignment with the grid. Macros, which are sequence of rules linked together by the directive descriptor, are used extensively in the grammar to achieve many of the desired design transformations.

The next chapter gives some general background on the shape grammar language and illustrates how portions of the rule application process are derived from the equations governing the mechanics of applying a rule.

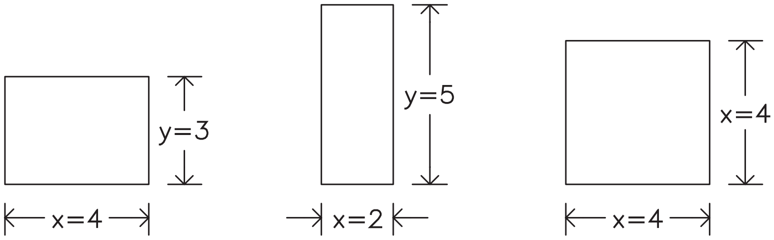
2.0 Shape grammars

Shape grammar is a general computational language that manipulates shapes. A shape can be a point, line, plane, or solid. Each of these elements is situated in one of the algebras U_{ij} V_{ij} or W_{ij} where $i \leq j$ (Stiny 1992, 2001). The U algebra represents generic shapes. The V algebra represents labeled shapes. And the W algebra represents weighted shapes. The subscript i represents the dimension of the element while the subscript j represents the dimension of the space. For instance, a point in a 2D plane is in the algebra U_{02}. A labeled line in a 2D plane is in the algebra V_{12}. A weighted plane in 3D space is in the algebra W_{23}. Most of the elements used in this work will be in the algebra V_{02} and V_{12}.

A shape can have dimensions which are fixed or parametric. A parametric shape is commonly called a schema. The function g() denotes the variables assigned to a parametric shape. For example, schema A defines all possible rectangles where x>0 and y>0 (figure 2.01) but g(A) defines a rectangle of a specific size as shown in figure 2.02.

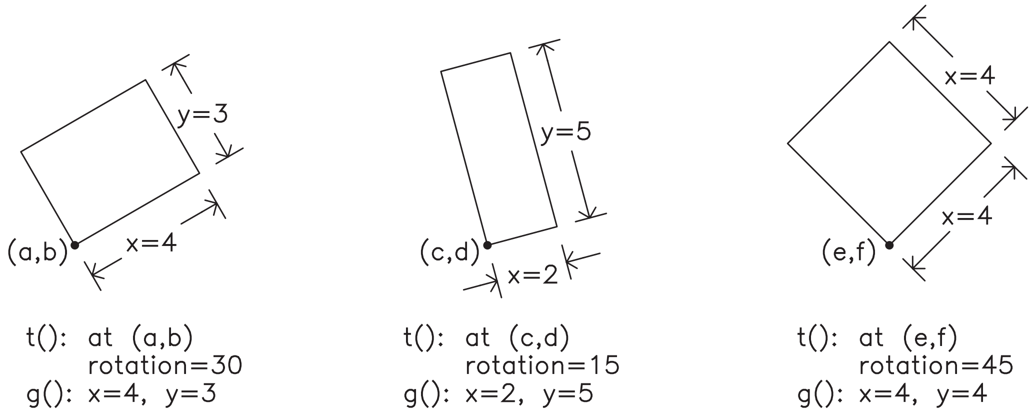


2.01 Schema A which defines all possible rectangles where x>0 and y>0.



2.02 Three specific sizes of schema A (figure 2.01) which are instances of g(A).

In order to locate a specific sized schema in a drawing, each schema has a transformation function associated with it. The transformations include translation, rotation, reflection, and scaling. The function t() denotes a specific transformation for a given shape. A specific instance of the shape can be specified by combining both the parameters, g() and the transformations, t() to produce t(g(A)). Figure 2.03 shows three specific instances of schema A. The first instance is a rectangle where x=4 and y=3 rotated 30 degrees clockwise about the point (a,b).



2.03 Three specific transformations and parameters of schema A (figure 2.01) which are instances of t(g(A)).

A shape grammar generates a design through the application of schema rules. A rule is in the form A → B where A and B are schemata. The application process in the shape grammar language is composed of three phases: parameter requirements, transformation requirements, and application method. This is evident by the two formulas for rule application (Stiny 1980a, 1990, 1991). For a given schema rule A → B and a drawing C, first determine if schema A, is a subshape of drawing C:

$$t(g(A)) \leq C \qquad (1)$$

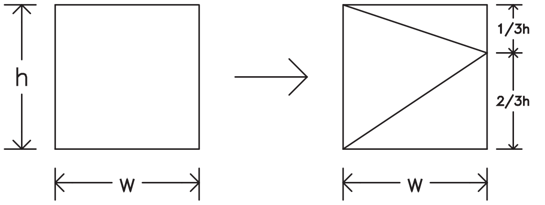
Second, the new drawing C' is the result of subtracting schema A from shape C and adding schema B:

$$C' = C - t(g(A)) + t(g(B)) \qquad (2)$$

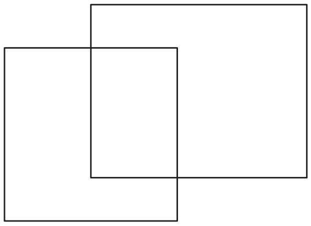
This two step process describes the mechanics of applying a rule in the shape grammar language. Associated with each schema is a parameter function g() and a transformation function t(). These functions determine what values are assigned to the parameters of a schema and what transformations are necessary to match a subshape in drawing C. These two functions make up the first two phases: parameter requirements and transformation requirements. Figures 2.04-2.07 provide an example to illustrate the two formulas.

Figure 2.04 shows a schema rule that inserts two diagonal lines into a rectangle at a specific ratio to the height of the rectangle. Figure 2.05 shows the drawing C which is a shape composed of two interlocking rectangles. To apply schema rule 2.04 to the drawing, we must first determine what values of g() and t() are possible for a match. Figure 2.06 shows one possible match where the center rectangle is selected. In this case, the parameter function g() assigns 6 to the variable w and 9 to the variable h and the transformation function t() states that the rectangle is rotated 180 degrees at point (x,y).

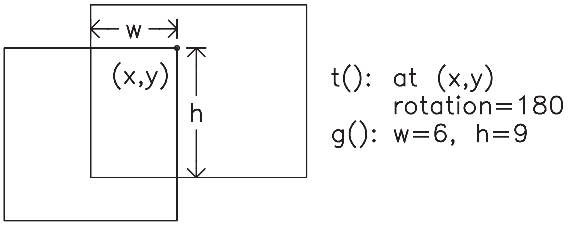
A derivation is show in figure 2.07. In total, there are 24 possible squares to choose from. There are 3 different rectangles and each rectangle has 8 different possible pairs of parameters and transformations (4 rotation transformations and 4 rotation transformations with reflection). All the rectangles are enumerated in figure 2.08 using schema rule 2.04.



2.04 A schema rule that inserts two diagonal lines into a rectangle at a specific ratio to the height of the rectangle.

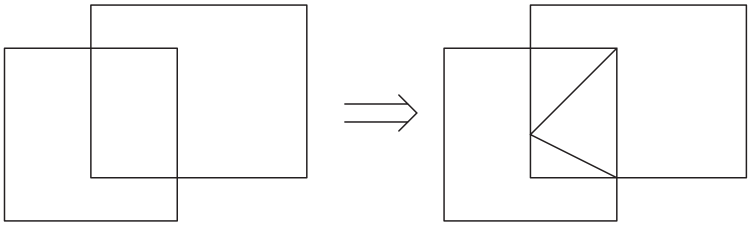


2.05 A sample drawing C.

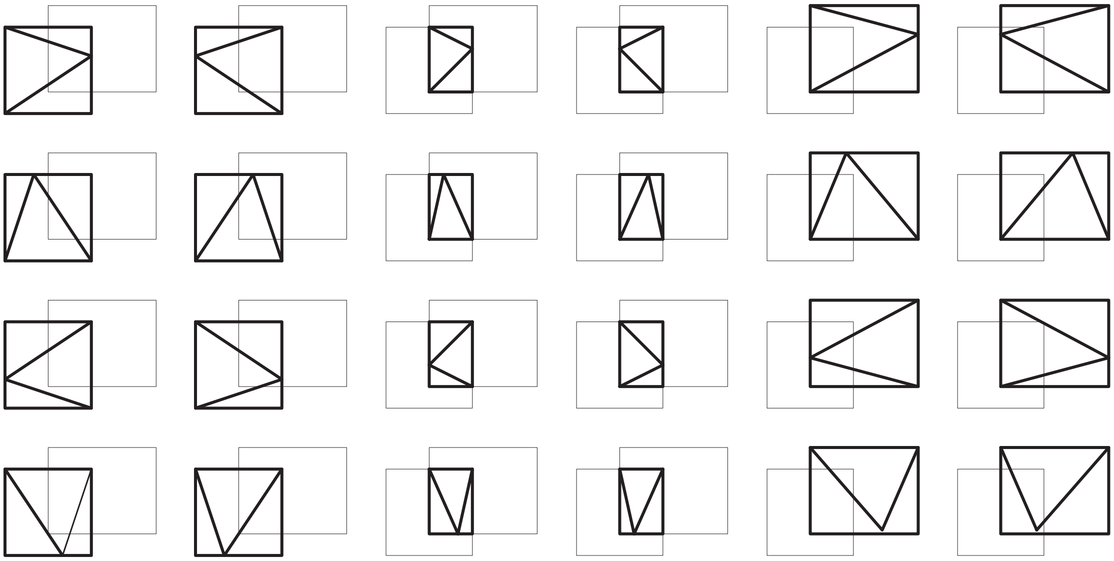


t(): at (x,y)
     rotation=180
g(): w=6, h=9

2.06 One possible selection of a subshape in drawing C using the schema rule in 2.04. The value for the parameter g() is w=6, h=9 and the value for the transformation t() is a rotation of 180 degrees at point (x,y).



2.07 Derivation of schema rule 2.04 when applied to the center rectangle of drawing C (figure 2.05) using the transformation and parameter settings in figure 2.06.



2.08 Enumeration of all the possible squares in drawing C (figure 2.05) using rule 2.04.

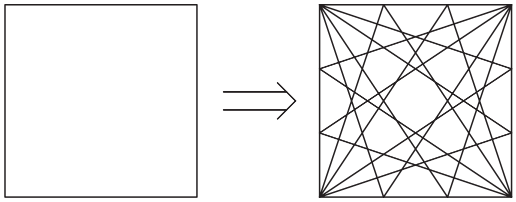
The third and last phase of the rule application process is the application method phase. The first formula (1) will only find one pair of values for t() and g(). Typically, there are multiple subshapes in a drawing which would require a set of values for t() and g() as illustrated in figure 2.08. To produce this set of values, a slight modification is made to the formulas:

$$\text{For all t and g such that } t(g(A)) \leq C \qquad (3)$$

To apply the entire set of subshapes to drawing C:

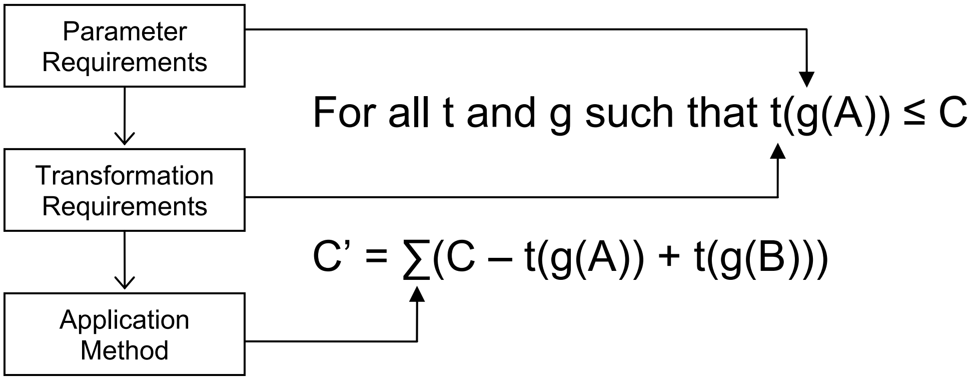
$$C' = \sum(C - t(g(A)) + t(g(B))) \qquad (4)$$

The application method phase determines how a set of subshapes should be applied to the drawing C. The options range from applying only one of the subshapes to applying all of the subshapes, which is equivalent to the parallel application of the rule as shown in figure 2.09.



2.09 Derivation showing the parallel application of rule 2.04 on a single square.

The three phases described thus far, explain the mechanics of applying any rule (figure 2.10). The formulas in the language only describe how to find a shape, erase it and add another shape. This process omits the decisions that need to be made before applying a rule. In order to apply a rule, a rule must first be selected. How is a rule chosen? Is the user free to select any rule in the grammar? Does the grammar restrict which rules can be used? Once a rule is selected, where can the rule apply? Can the rule apply to all parts of the drawing or are only certain portions applicable?


Parameter Requirements
Transformation Requirements
Application Method
For all t and g such that t(g(A)) ≤ C
C' = ∑(C – t(g(A)) + t(g(B)))

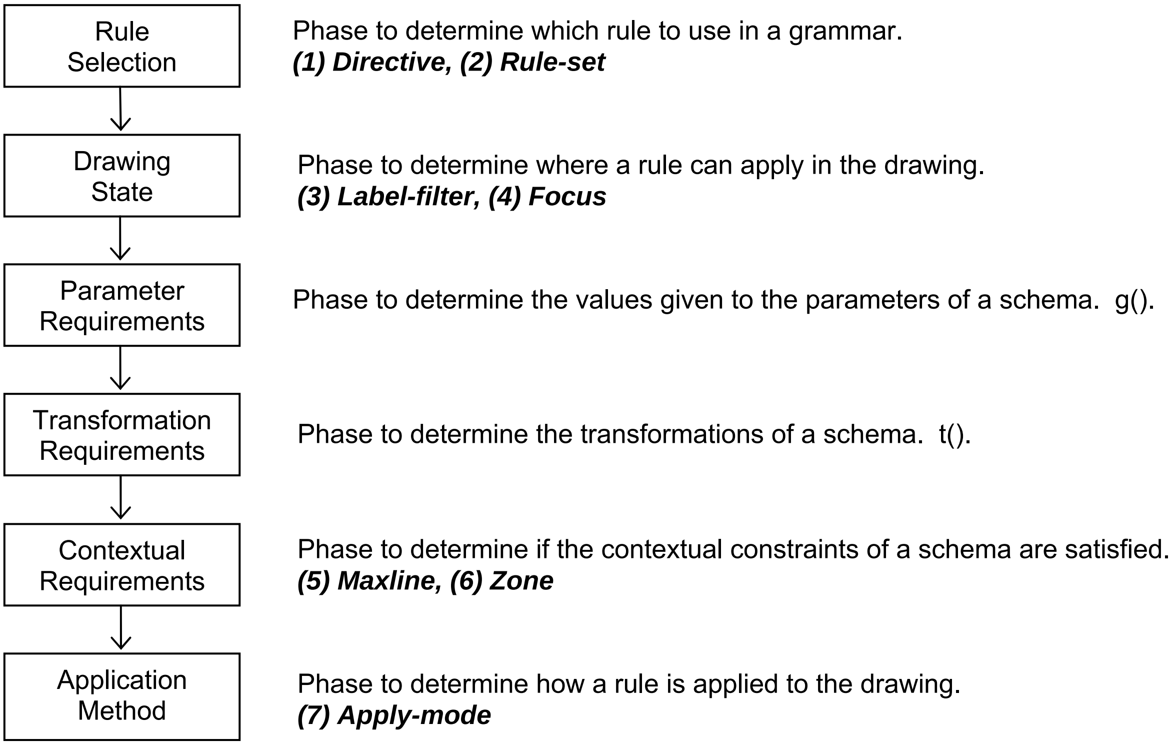
2.10 Diagram showing how the formulas apply to the parameter requirements, transformation requirements and application method phases.

These questions address the overall process of applying a rule in the shape grammar language. According to Chase (2002), there are three actions to apply a rule: determination of rule, determination of object, and determination of matching condition. To incorporate these actions into the rule application process, two phases are added to the process: rule selection and drawing state. The rule selection phase determines which rule to apply. The drawing state phase determines what portion of the drawing to apply a selected rule.

According to the shape grammar language, the matching conditions are composed of two phases: parameter requirements and transformation requirements. The SGML adds a third phase, contextual requirements, to the matching conditions. This phase determines if the context of the subshape fulfills the contextual constraints defined in the schema. In order to have a match, the requirements of all three phases must be fulfilled. The next section will describe all six phases and their corresponding meta-language descriptors.

3.0 Six phases of the rule application process and the meta-language descriptors

The six phases of the rule application process are: rule selection, drawing state, parameter requirements, transformation requirements, contextual requirements, and application method (figure 3.01). Three of these phases, parameter requirements, transformation requirements, and application method, are derived from the original shape grammar formulas. The three new phases are rule selection, drawing state and contextual requirements.



Phase to determine which rule to use in a grammar.
(1) Directive, (2) Rule-set

Phase to determine where a rule can apply in the drawing.
(3) Label-filter, (4) Focus

Phase to determine the values given to the parameters of a schema. g().

Phase to determine the transformations of a schema. t().

Phase to determine if the contextual constraints of a schema are satisfied.
(5) Maxline, (6) Zone

Phase to determine how a rule is applied to the drawing.
(7) Apply-mode

3.01 The six phases of the rule application process and their corresponding descriptors. The descriptor names are shown in bold italic. The sequence of the phases is determined by the order of the decisions a user makes in order to apply a rule.

The sequence of the phases is determined by the decisions a user makes in order to apply a rule. The first step in using a grammar is to determine which rule to use. This decision is made in the first phase named rule selection. Once a rule has been selected, the second phase, drawing state, determines what portion of the drawing to apply the selected rule. This is achieved by filtering information in the drawing.

The next three phases determine the matching conditions between the schema and the drawing. With a modified drawing from the drawing state phase, values are assigned to the parameters of the schema and transformations are established to get a subshape match in the drawing. This is executed in the parameter and transformation requirements phases. The third phase of the matching conditions is contextual requirements. In this phase, the contextual constraints of the schema are evaluated to determine if the context of the subshape is satisfied. All subshapes that fulfill the three requirements phases are grouped together to create a set. The final phase, application method, determines how this set of subshapes is applied to the drawing.

The SGML provides descriptors for every phase except the parameter requirements and transformation requirements phases. The other phases have one or more descriptors for a total of seven descriptors: directive, rule-set, label-filter, focus, maxline, zone, and apply-mode (figure 3.02). The remainder of this chapter will provide details of each phase and any corresponding descriptors from the shape grammar meta-language. The explanations will start in reverse order from the application method phase to the rule selection phase.

Phases	Descriptors	Options	Descriptions
Rule Selection	**Rule-set**	set-rule add-rule sub-rule	A parallel description that determines which rules are available for use by the user during the derivation of a grammar. A rule can modify the rule-set with three options: set-rule, add-rule, sub-rule. Set-rule sets which rules are in the rule-set. Add-rule adds rules to the rule-set and sub-rule subtracts rules from the rule-set.
	Directive	success failure	A control mechanism added to a rule that dictates which rule to apply next depending upon the success or failure of the given rule to apply to the drawing. If a given rule is embedded in the drawing then the success rule is applied next. If the given rule does not exist in the drawing then the failure rule is applied next.
Drawing State	**Label-Filter**	on off	An option to filter out any labeled shapes in the drawing that is not a labeled shape in the left-hand schema of a rule.
	Focus	-	A special set of labeled lines that control what areas of a drawing are available for rule application. The areas are defined as enclosed polygons.
Contextual requirements	**Maxline**	-	A specification used in the left-hand schema of a rule to specify that a line in the subshape must be a maximal line in the drawing.
	Zone	void exclude	A specification that associates an area of a schema with a predicate function. One commonly used predicate function is the void function which states that the specified area must be void of all shapes. Another predicate is exclude which states that the demarcated area can have any labeled shape except for those specified by the exclude function.
Application Method	**Apply-Mode**	single parallel random	A specification that controls how a set of subshapes is applied. Possible options are single, parallel and random. The single option allows the user to choose one subshape to apply to the drawing. The parallel option applies all subshapes to the drawing. And the random option randomly applies one subshape to the drawing.

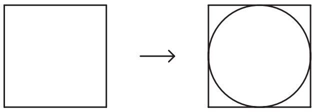
3.02 Table describing the seven meta-language descriptors and their associated rule application phase.

3.1 Application method phase

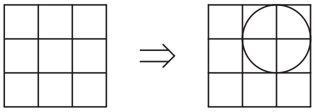
The application method phase determines how a set of subshapes is applied to a drawing. As shown in figure 2.08, there can be many possible different subshape matches between the left-hand schema of a rule and the drawing. Typically only one subshape is used but there are other possibilities such as a parallel application which would apply all subshapes at the same time. The shape grammar meta-language provides one descriptor, apply-mode, to allow the developer of a grammar to specify what type of application a rule should have.

3.1.1 Apply-mode

The apply-mode descriptor determines how a rule is applied to the drawing with three control options named single, parallel, and random. The single option allows the user to select one of the subshapes for application. Figure 3.04 shows the selection of a 2x2 square in the upper right hand corner using the rule in figure 3.03 (rule 3.03).

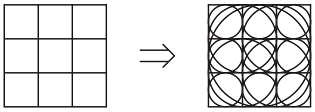


3.03 Schema rule that finds a square and places a circle in the middle of it.



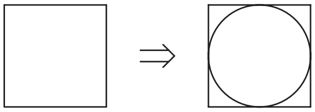
3.04 Derivation of rule 3.03 when the apply-mode is single. Here the user has selected the upper right-hand square.

The second option is parallel. This option will apply the rule to all possible subshapes in the drawing. In a 3x3 square grid, there are 14 different square subshapes as shown in figure 1.01. A derivation showing the effects of a parallel application using rule 3.03 is shown in figure 3.05.



3.05 Derivation of rule 3.03 when the apply-mode is parallel. A circle is placed in all 14 squares. An enumeration of the 14 different possible squares is shown in figure 1.01.

The third option is named random. With this option, any subshape can be applied to the drawing. The subshape can be randomly selected by the user or if used in conjunction with a shape grammar interpreter, a computer can randomly select the subshape (Chase 1989, 2002). This option can also be used when all subshapes are known to produce an equivalent shape in spite of differences in transformations and parameters. An example of this is shown in figure 3.06 where the application of rule 3.03 will produce the same result regardless of which transformation is selected.



3.06 Derivation of rule 3.03 when the apply-mode is random. All possible transformations and parameters produce the same result.

3.2 Contextual requirements phase

The matching conditions between a schema and the drawing are traditionally determined by the parameter function g() and the transformation function t() which correspond to the parameter requirements and transformation requirements phases. The shape grammar meta-language adds an additional phase, named contextual requirements, which has descriptors that provide additional matching constraints for the determination of a subshape based on the context of the subshape in the drawing. These three phases, in combination, determine the matching conditions of a schema.

One of the difficulties in using shape grammars is that the matching conditions between the schema and the drawing works only on the shapes found. It is not easy to define a schema that uses the surround conditions of the subshape as part of the matching constraint. For instance, suppose I wanted to define a schema that found rectangles that were clear of any shapes on the inside.

This is difficult to define in the language because there is no direct convention that states that the inside of a subshape found in the drawing must be empty of shapes. Instead such a condition could be defined in the language using a series of compound rules that combine shapes along with a parallel description grammar (Knight 2003). The function in the parallel description grammar is the algorithm that would determine if the inside of the rectangle is void of any shapes. Other techniques include, adding additional geometries or labels, constraining the transformations, or varying the parametric values in order to isolate the desired condition.

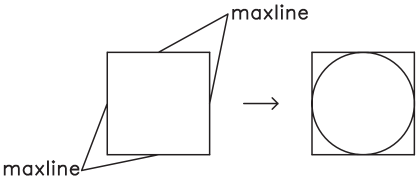
Notice that these techniques complicate the situation and deviate from the simplicity of the original condition which is succinctly stated as finding a rectangle that is clear of any shapes on the inside. Instead of relying on compound rules with parallel description grammars or using transformations and parameters restrictions, the descriptors in the contextual requirements phase rely on the visual properties of the subshapes. Determining if the inside of a rectangle subshape is empty is a visual property of the subshape relative to the drawing.

There are two meta-language descriptors in the contextual requirements phase: maxline and zone. Maxline constrains a line in the subshape to be a maximal line. The zone descriptor evaluates a predicate function against an area of the drawing defined relative to the subshape. This method allows subshapes in a drawing to assess the surrounding conditions. The zone descriptor can directly define a schema that finds rectangles with no shapes on the inside.

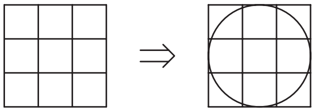
3.2.1 Maxline

The maxline descriptor adds an additional constraint that the matching subshape line must be a maximal line. By definition, a maximal is a line that can not be embedded in another line. The use of the maxline descriptor allows the developer to specify that a line in the subshape is a line that is not a smaller portion of a larger line in the drawing.

For example, figure 3.07 shows a rule where the left-hand schema is a square composed of maxline lines. By using this rule only one subshape, the outer square, can be found because of the restriction that all the lines in the square must be maximal lines (figure 3.08). A parallel application of the rule without the maxline description would result with a circle in all 14 possible squares as shown in figure 3.05. The use of the maxline descriptor therefore defines a schema that differentiates the larger outer square from the smaller inner squares in the drawing.



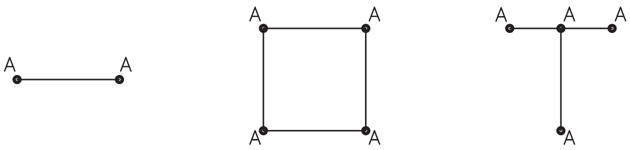
3.07 Rule where the left-hand shape is composed of lines with the maxline descriptor.



3.08 The parallel application of the rule in figure 3.07 on a 3x3 grid. Only the larger outer square is possible for selection.

Another key use of the maxline descriptor is to define schemata that are determinate. A determinate schema is one that has a finite number of subshapes in a drawing. An indeterminate schema has an infinite number of subshapes. A schema composed of only one line is indeterminate because there are an infinite number of lengths that can be embedded in a line. Often times, a determinate shape is desired to fix the number of possibilities. The maxline descriptor can be used to make a line a determinate shape.

Another method of finding maximal lines is to place labeled points at the endpoints of a line (figure 3.09). To find a maximal line in the drawing, the schema simply looks for a line with two endpoint labels. Although this sounds like an easy solution to identifying maximal lines, it is in fact not a reliable method because the notion of a maximal line is determined by the placement of labeled points instead of the visual property of the line.



3.09 Examples of how a maximal line can be defined by using labeled points at the endpoints of a line. The left image is a schema that looks for a line with the labeled point A at the endpoints of the line. The middle image shows how the schema is used to find the four maximal lines of a square. The right image shows why this method is not reliable. Although the schema from the first image is able to find the two maximal lines in the T shape, it is also able to find the shorter line segments along the top of the T which is not a maximal line.

For example, the right image in figure 3.09 shows two lines with labeled points at the endpoints of the line arranged to make a T shape. A schema looking for a line with labeled endpoints would find the maximal lines but would also find the shorter line segment along the top of the T which is not a maximal line. The maxline descriptor resolves this problem by using the visual information in the drawing to determine which lines are maximal.

The descriptors of the meta-language are based on the shape grammar language. Therefore, the same effect can be achieved without the use of the meta-language. The maxline descriptor constraints a line in the subshape to be a maximal line in the drawing. To describe a schema that finds a maximal line without the use of the maxline description requires thinking about how to specify the same conditions using the transformation and parameter components. The maxline descriptor is the equivalent to restricting the scaling factor of the line to be at 100%. In other words, the line must match the entire line.

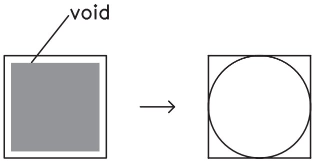
The descriptors in the contextual requirements can be considered short cuts for the equivalent definitions in the shape grammar language. But more importantly the descriptors describe a rule by emphasizing certain global conditions surrounding the subshape as opposed to emphasizing the descriptive methods of the shape grammar language. For the maxline descriptor it is the difference between seeing a line as a whole versus seeing the line as a scaling factor of 100%.

3.2.2 Zone

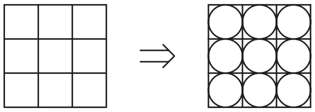
The zone descriptor adds an addition constraint on the matching conditions of a schema in the form of a predicate function. With the use of the zone descriptor, not only must the geometry of the schema be embedded in the drawing, the predicate must also be true. The predicate function evaluates a marked area of the schema to determine if the predicate is true or false. By using the zone descriptor, the schema can use the context of the drawing in which the subshape is found as part of the constraints.

A commonly used function is the void function which states that the demarcated area must be void of any shape. This function enables the schema to detect empty spaces relative to the subshapes found in the drawing. Other computational languages, such as structure grammars (Carlson and Woodbury 1992), have also used the concept of a void.

The void zone can be used to differentiate subshapes in a drawing as demonstrated in figures 3.10-3.17. Suppose the developer of a grammar wants a rule that picks out only the nine smaller squares in a 3x3 grid. Using the rule in figure 3.03 will find 14 different types of squares. In order to find only the nine smaller squares, the rule in figure 3.10 is used instead. This rule has a void zone to demarcate that the area inside the square must be empty. A parallel application of the rule is shown in figure 3.11.

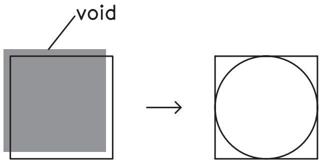


3.10 A rule that finds a square such that the inside of the square is void of shapes.

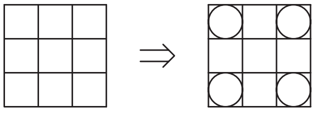


3.11 The parallel application of rule 3.10 on a 3x3 grid.

The use of the void zone therefore differentiates the smaller squares from the larger squares. This occurs because whenever a larger square is selected, the void zone constraint rejects the subshape since there are lines in the interior of the square. In other words, the void predicate function returns false. The void zone can also be used to distinguish other types of squares in the grid. The rule in figure 3.12 is used to define a schema that selects only the corner squares.



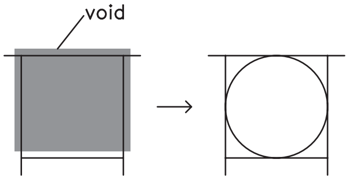
3.12 A rule that finds a square such that one corner of the square is void of shapes.



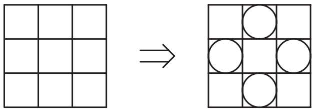
3.13 The parallel application of rule 3.12 on a 3x3 grid.

The difference between rule 3.10 and rule 3.12 is that the void zone in rule 3.12 has been enlarged to cover one corner of the square. A corner square has the characteristic that one corner of the square does not have any protruding lines. By expanding the void zone to cover one corner, the rule is guaranteed to select only the corner squares. A parallel application is shown in figure 3.13.

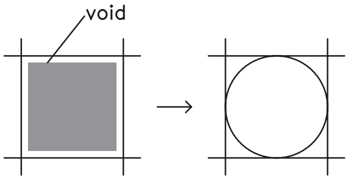
In a similar fashion, the void zone can also be used to distinguish squares along the edge of the grid and squares inside the middle of the grid (figures 3.14-3.17). These rules not only make use of the void zone but also the unique geometries of each square type. To define a schema that finds only squares along the edge of the grid, two changes are made to the original schema in rule 3.10. The first change is to add protruding lines to the geometry of the square to prevent the schema from picking a corner square (figure 3.14). The second change is to enlarge and shift the void zone so that only one side of the square does not have any protruding lines. The void zone prevents the schema from selecting a center square. A similar method is used to define a schema that finds only the center area squares. The schema to select center area squares has protruding lines on all four corners and the area inside the square must be void of any shapes (figure 3.16).



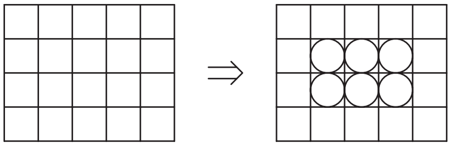
3.14 A rule that finds a square along the edge of the grid.



3.15 The parallel application of rule 3.14 on a 3x3 grid.



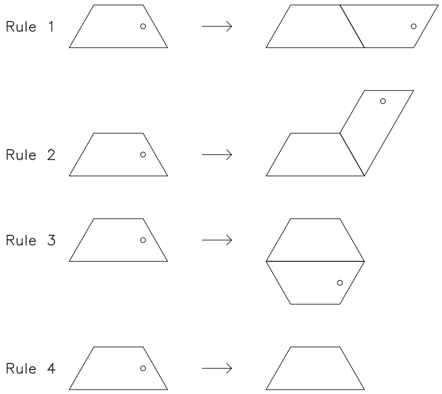
3.16 A rule that finds the center squares of a grid.



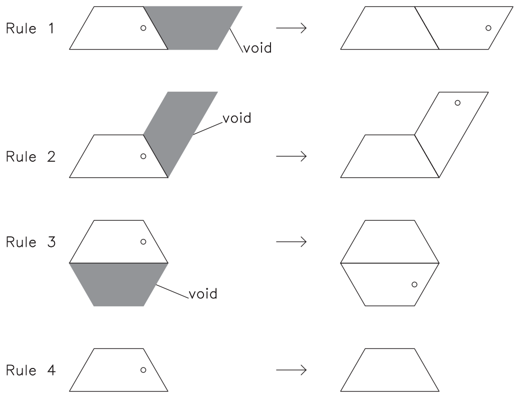
3.17 The parallel application of rule 3.16 on a 5x4 grid.

Another way of using the void zone in shape grammars is to use it as an overlap detector. Cagan and Mitchell (1993) show an example of the shape annealing design technique with the half hexagon grammar (figure 3.18). The objective in their example is to fill in a space with half-hexagons such that no half-hexagon overlaps another and all half-hexagons are within the boundaries of the space.

Their solution incorporates an overlap detector in the shape annealing algorithm that would test the result of each rule application to determine if the rule created an overlap or went out of bounds. This step in the algorithm could be avoided by using the void zone in the half-hexagon rules to insure that the area where the half-hexagon is being added is void of any pieces (figure 3.19). By incorporating the void restriction into the shape rules it is guaranteed that each rule will produce the appropriate result of not having any half-hexagons placed out of bounds or over another half-hexagon.



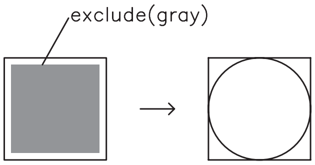
3.18 The original four rules of the half-hexagon grammar.



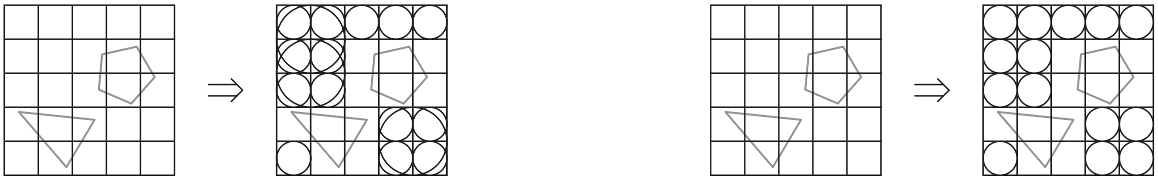
3.19 The four rules of the half-hexagon grammar incorporating the use of the void zone to guarantee that the path created by the rules will not overlap another half-hexagon or go beyond the boundaries of the space.

The void function is one commonly used predicate when using the zone descriptor. An example of another function is the exclude function which tests if the demarcated area does not contain a specific set of labeled shapes. Unlike the void zone where the designated area must be void of all shapes, the exclude zone can have any labeled shapes in the area, except for the ones specified.

Figures 3.20 and 3.21 demonstrate the effects of the exclude zone. The left-hand schema of figure 3.20 defines a square that does not have any gray labeled lines on the inside of the square. A sample derivation of schema rule 3.20 is shown on the left of figure 3.21. All possible squares that do not contain any gray labeled lines have a circle placed on the inside. The right derivation of figure 3.21 shows the effects of rule 3.10 where the void zone is used instead of the exclude zone. Notice that the larger 2x2 grid circles do not appear in the derivation.



3.20 A schema rule that makes use of the exclude zone. The rule looks for squares that do not have gray labeled shapes on the inside and adds a circle.



3.21 The left derivation shows the parallel application of schema rule 3.20. A circle is place in all squares that do not contain any gray labeled shapes. The right derivation shows the parallel application of schema rule 3.10 where the void zone is used instead of the exclude zone. Notice the larger 2x2 grid circles do not appear in the derivation.

The zone descriptor can be implemented in the shape grammar language using a sequence of compound rules with a parallel description grammar. Knight (2003) has demonstrated that such a combination can be used with emergent shapes to define grammars controlled by external functions. The zone descriptor is a special case of this situation where the external function is the predicate function of the zone. The rules could be written in the following form:

$A, 0 \rightarrow A', f(C, A, Z)$
$A', 1 \rightarrow B, 0$
$A', 0 \rightarrow A, 0$

The compound rules have two components. The first component is a shape rule and the second component is a rule for the parallel description grammar. The first rule looks for a shape A with the parallel description of 0 and transforms it to the shape A' with the result of the function *f* which corresponds to the predicate function of the zone descriptor. The inputs to the function are: C, the drawing, A, the schema, and Z, the defined zone. The function returns 1 if satisfactory and 0 if not. The second rule transforms the design if the function *f* returns 1. If not, then the third rule is applied to return the drawing back to its original state. If we apply these rules to the situation in figures 3.12 and 3.13 where the void zone is used to define corner squares of the 3x3 grid, the compound rules are as follows:

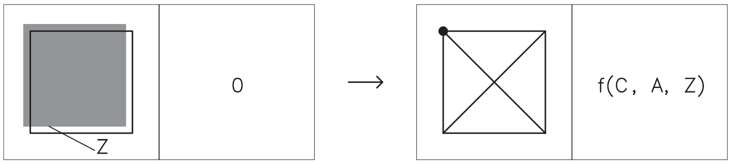


Figure 3.22 Rule A, 0 → A’, *f*(C, A, Z). The left side of the compound rule is the shape component. The right side is the parallel description grammar component.

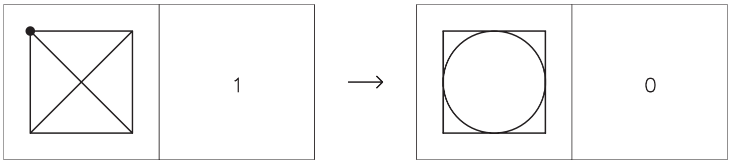


Figure 3.23 Rule A’, 1 → B, 0

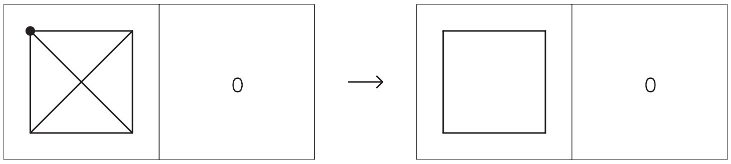


Figure 3.24 Rule A’, 0 → A, 0

A sample derivation using the three rules in figures 3.22-3.24 is shown in figure 3.25. In the first rule application, the user applies rule 3.22 and has picked a square where the function *f* returns 1. Rule 3.23 is applied next to finish the transformation. In the third rule application, the user has picked a square where function *f* returns 0. The next to rule to apply is 3.24 to return the drawing back to its original state.

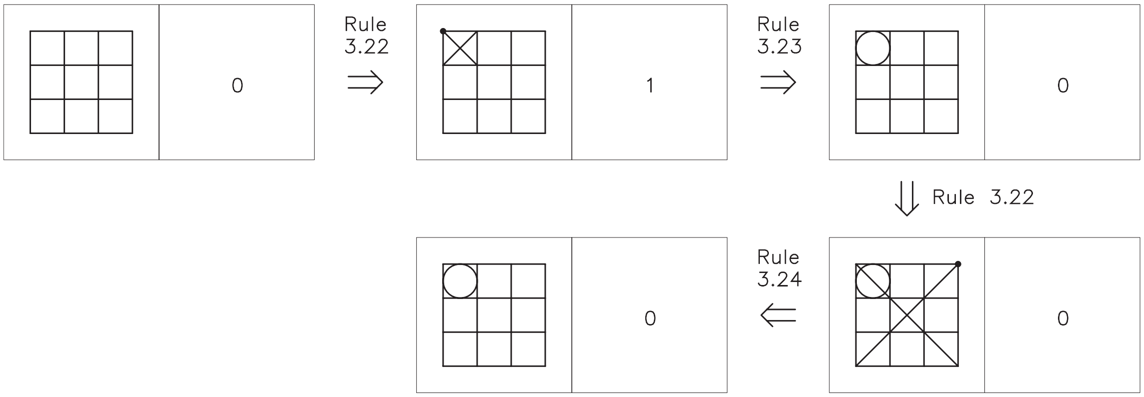


Figure 3.25 A sample derivation using the three rules in figures 3.22-3.24. The first two rule applications show the successful placement of a circle in a corner square. The second two rule applications show the unsuccessful application of rule 3.22.

The parallel description grammar (Stiny 1981) is a general mechanism which has been used, in this case, to evaluate an external function in order to restrict the application of rules. The zone descriptor, on the other hand, is a specific mechanism that simplifies the process of using an external function by eliminating the need for a sequence of compound rules used in conjunction with a parallel description grammar.

3.3 Transformation requirements phase

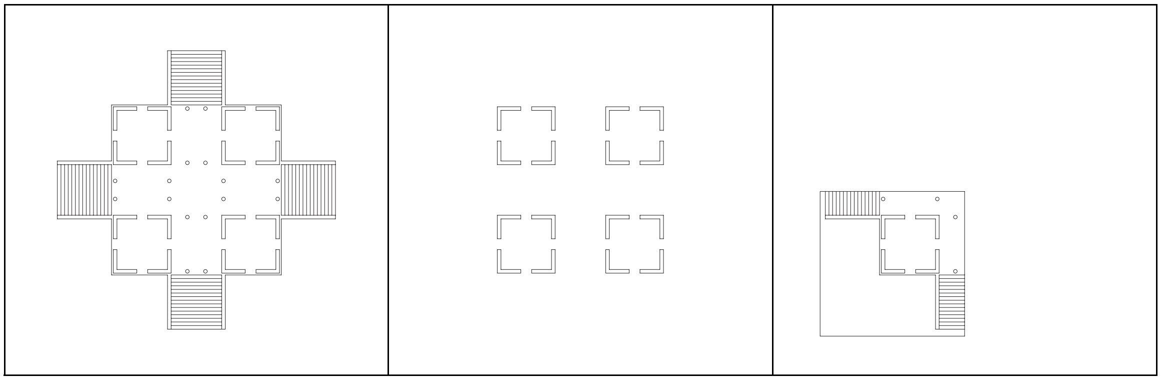
The transformation requirements phase determines what transformations are necessary in order to have a subshape match in the drawing. This phase corresponds with the t() function. There are no meta-language descriptors in this phase. The transformations include translation, rotation, reflection and scaling. Restrictions can be stipulated by specifying a value such as setting the scaling factor to be 100%, which is the equivalent of the maxline descriptor, or by specifying the type of transformations to use such as all transformations except for reflection.

3.4 Parameter requirements phase

The parameter requirements phase corresponds with the g() function and determines what values are given to the parameters of the schema in order to have a subshape match in the drawing. There are no meta-language descriptors in this phase. A common use of the g() function is to determine the parameters for the size of a shape. Restrictions can be placed on the parameters to match certain types of subshapes in the drawing.

3.5 Drawing state phase

The drawing state phase determines where a rule can and cannot apply. Typically, the entire drawing is used to look for a possible subshape match between the left-hand schema of a rule and the drawing. But sometimes this is not desirable. For instance, when affecting changes on a drawing that only affect the walls of the building, it might be desirable to alter the drawing so that only the walls are visible for manipulation. In another situation, maybe the design rules are concerned with affecting changes in only one room. It might then be desirable to alter the drawing so that only the one room is visible for manipulation (figure 3.26).



3.26 The original plan of a Durand building (left) with alternative views where only the walls are visible (middle) and where only one room and its surroundings is visible (right).

The examples above all alter or “see” the drawing in a different context. If we are concerned with only walls, we see only walls. If we are concerned with seeing a particular room, we see only that particular room. This process of seeing a drawing as something else is part of the design process. Schön and Wiggins (1992) has characterized this as a cyclical see-move-see process. The designer sees the drawing as something and evaluates the design to make a move. This in turn generates a new drawing which can be re-interpreted and re-evaluated to make new moves.

In a similar fashion, the modifications permitted by the drawing state phase gives a rule the ability to see the drawing as something else. What we see varies greatly and depends, in part, on the context in which the rule is applied. The ultimate goal of this phase is to view the drawing in such a way as to isolate the parts of the drawing that are pertinent to the rule. A drawing can get quite complex and the drawing state phase is a means to manage that complexity by removing parts of the drawing that are not of interest at the time.

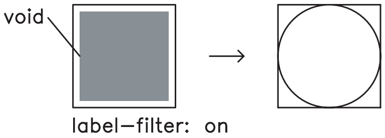
The shape grammar meta-language provides two generic methods for altering the drawing. The first method is information filtering which filters out unnecessary shapes from the drawing. What shapes to filter out is dependant on the labeled lines in the left-hand schema. This method has the same effect as looking at only a specific type of line, such as walls, in drawing (figure 3.26 middle). The SGML descriptor for this method is named label-filter.

The second method is based on visual attention. When a person pays attention to a particular object in a drawing, they produce what is commonly called the searchlight of attention (Posner 1980). The location and scope of the searchlight determines what the person is focused on. This effect is abstractly mimicked using the focus convention in

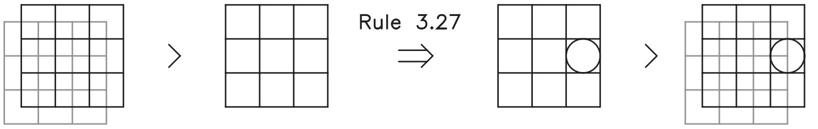
SGML. An area of focus is demarcated by placing a special labeled polygonal shape in the drawing. All areas outside of the polygonal shape are ignored. This method has the same effect as looking at a specific room in a drawing (figure 3.26 right).

3.5.1 Label-filter

The label-filter descriptor filters out any labeled shapes in the drawing that does not have the same labels as those used in the left-hand schema of the rule. For example, the rule in figure 3.27 uses a void zone with the label-filter option on so that only the smaller squares are selected along. The derivation of applying this rule to two overlapping 3x3 labeled grids is shown in figure 3.28.



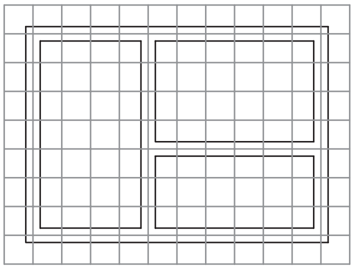
3.27. A rule that uses the label-filter to filter out irrelevant labeled shapes. The shaded area is the void zone.



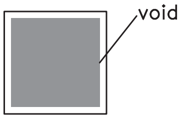
3.28 An example derivation showing the effects of the label-filter. The gray labeled lines are shown in gray and the black labeled lines are shown in black. The greater-than symbol signifies the effects of the label-filter descriptor. When the rule from figure 3.27 is applied, the label-filter removes all the gray labeled lines from the drawing temporarily. The user then selects the middle right square. After the rule is applied, the label-filter effect is removed which brings back the gray labeled lines.

The first image shows the overlapping 3x3 labeled grids. The gray labeled lines are shown in gray and the black labeled lines are shown in black. The second image shows the effects of the label-filter. Because the gray lines are not part of the left-hand schema, they are removed temporarily from the drawing. Now that the gray lines are removed, rule 3.27 can apply and the third image shows the user selecting the middle right square. The fourth and last image shows the final result of apply rule 3.27 where the gray lines are replaced. Applying rule 3.27, with the label-filter off, to the first image in figure 3.28 will not work because of the gray overlapping grid. The void zone will return false every time because of the gray grid pattern.

In an architectural setting, the label-filter descriptor could be used to filter out background information in a drawing. For instance, a drawing can consist of walls laid out on a square grid pattern (figure 3.29). To select an empty room, the schema in figure 3.30, which looks for a rectangular room that is void of shapes on the interior, could be used.



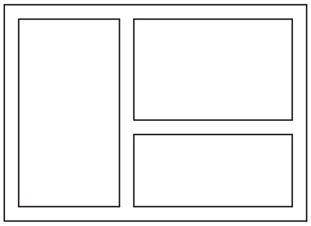
3.29 A sample drawing consisting of walls placed on a grid.



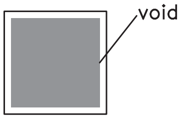
label–filter: off

3.30 A schema that looks for an empty rectangular room with the label-filter descriptor off.

Unfortunately this schema will find no rooms in figure 3.29 because the void zone will detect grid labeled lines whenever it finds a room. To fix this problem, the filter-label option in the schema is turned on (figure to 3.32). This has the effect of removing any labeled shapes in the drawing that are not part of the labeled shapes used in the schema. Since the schema does not have any grid labeled lines in it, the grid labeled lines are temporarily removed from the drawing before searching for an empty room (figure 3.31). With the grid lines removed, the three empty rooms can now be selected.



3.31 The effect of schema 3.28 on the same drawing as 3.25. Schema 3.28 has the label-filter descriptor on, which has the effect of filtering out the grid lines in the drawing.



label–filter: on

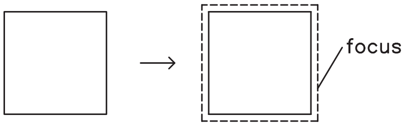
3.32 A schema that looks for an empty rectangular room with the label-filter descriptor on.

3.5.2 Focus

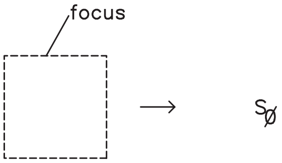
The focus descriptor controls what area or areas of the drawing can be used to find a subshape match. These areas are demarcated by enclosed polygons composed of a special labeled line named "focus". The focus lines can be altered and erased, just like any other labeled line and the enclosed polygons can be of any shape, concave or convex, as long as the lines do not intersect themselves or another focus polygon.

When a drawing contains an enclosed polygon composed of focus lines, all lines outside of the enclosed area are temporarily removed from the drawing. This temporary state is persistent until the focus lines are removed. Therefore, once a focus line is placed, all subsequent rules can only apply inside the demarcated areas. To apply a rule outside of the focus area, the focus lines have to be removed.

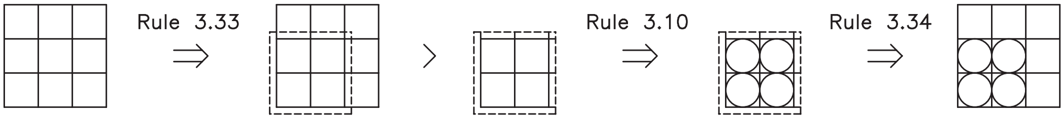
Figure 3.35 is a derivation that shows how the focus lines can be used to isolate an area of the drawing for subsequent rule applications. The derivation begins with the rule in figure 3.33, which draws a focus polygon, shown as dashed lines, around a square. In this case, the user has selected the lower left corner. After using this rule, any subsequent rule applied to the drawing will occur only in the lower left-hand corner of the drawing within the focus lines. The rule in figure 3.10 is applied next in parallel to place four circles inside the focus area. The last step of the derivation uses the rule in figure 3.34 to remove the focus lines so that the next rule used will apply to the entire drawing.



3.33 A rule that places a focus polygon around a square.

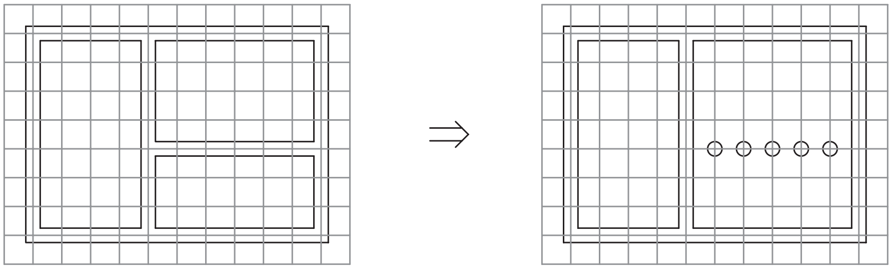


3.34 A rule that removes a square focus polygon.



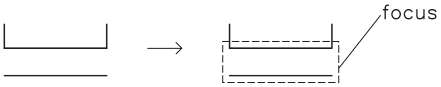
3.35 Example derivation showing the effects of the focus lines which are shown as dashed lines. The greater-than symbol signifies the effects of the focus lines. After rule 3.33 is applied, all areas outside of the focus polygon are temporarily removed. This has the effect of reducing the size of the drawing. Rule 3.10 is applied to place circles in each square within the focus polygon. Rule 3.34 is applied to remove the focus polygon so that subsequent rules can apply to the entire drawing.

In an architectural setting, the focus lines can be used to define specific areas where changes can occur. For instance, suppose the task is to replace a wall with a series of columns, at grid intersections (figure 3.36). The straightforward method is to write rules that transform walls of various sizes into a series of columns. The problem with this method is that you will need an infinite number of rules to accommodate the infinite number of wall and grid sizes.

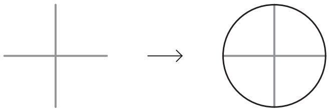


3.36 The derivation shows one wall of a 2D floor plan transformed into a series of columns where the columns are placed at the intersection of the grids.

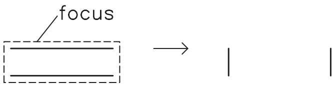
Another method is to develop a procedural series of rules that can vary according to the size of the wall and grid. This can be achieved in three steps (figure 3.40). First step is to define the area where the columns will be placed. In this particular case, it is a wall. We can use rule 3.37 to place a focus polygon around the desired wall. The next step is to place columns at the grid intersections. This can be achieved by applying rule 3.38 in parallel. If the focus rectangle did not exist, then the rule would have applied to all the grid intersections in the drawing as opposed to only the grid intersections inside the focus rectangle. The last step is to replace the walls and remove the focus polygon so that subsequent rule can apply to the entire drawing (figure 3.39).



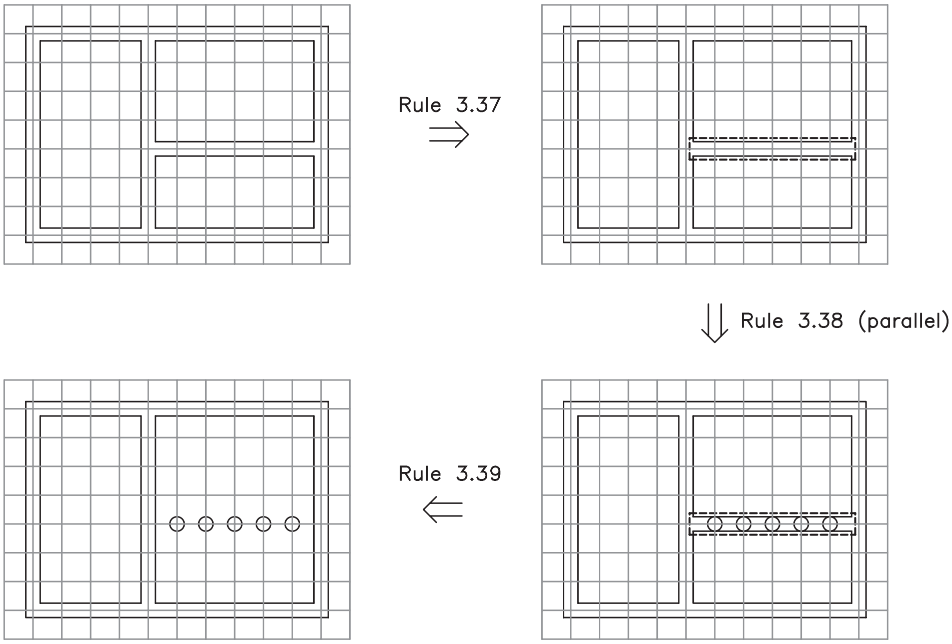
3.37 Rule to place a focus polygon around a wall.



3.38 Rule to place columns at the grid intersections.



3.39 Rule to remove the focus lines from the drawing and cap off the walls.

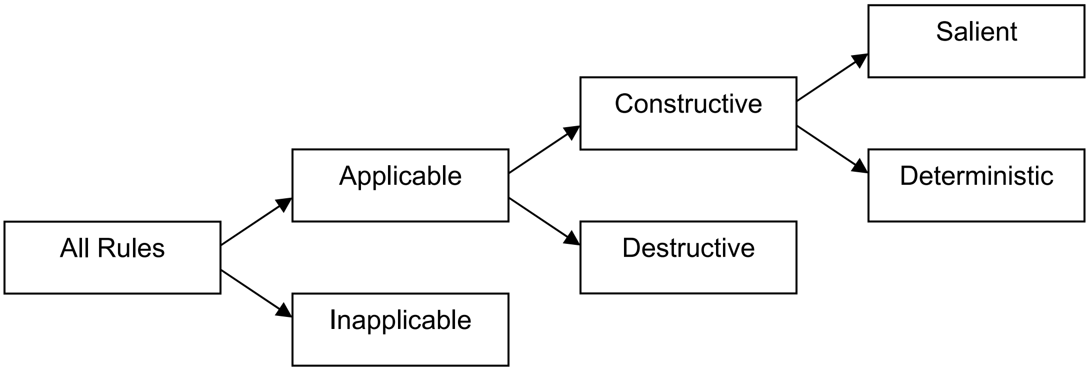


3.40 Derivation of replacing a wall with a series of columns. The derivation begins with the original plan. Rule 3.37 is applied first which adds a rectangular focus polygon around the selected wall. Rule 3.38 is applied in parallel to place a column at every grid intersection along the wall. The last rule applied is rule 3.39 which removes the wall and focus lines and caps off the ends of the wall.

The three step process I have just described requires that the rules are applied sequentially and in the correct order. Typically this is achieved by using state labels (Knight 1994). The next section will show some alternative methods for controlling the execution of a grammar provided by SGML.

3.6 Rule selection phase

The rule selection phase controls the derivation of the grammar by manipulating the availability and sequencing of rules. A grammar is composed of rules which can be placed into three general categories (figure 3.41). Of all the rules in the grammar, typically only a subset of the rules can be applied to the drawing at any point during the derivation. This is the first category: applicable vs. inapplicable. The applicable rules can be further divided into constructive or destructive rules. Just because a rule can affect changes in the drawing does not necessarily mean it is a constructive change for the design. The final category subdivides constructive rules into salient and deterministic rules. Salient rules provide the user of a grammar with design choices. Deterministic rules, on the other hand, are mechanistic rules used to complete a design transformation.

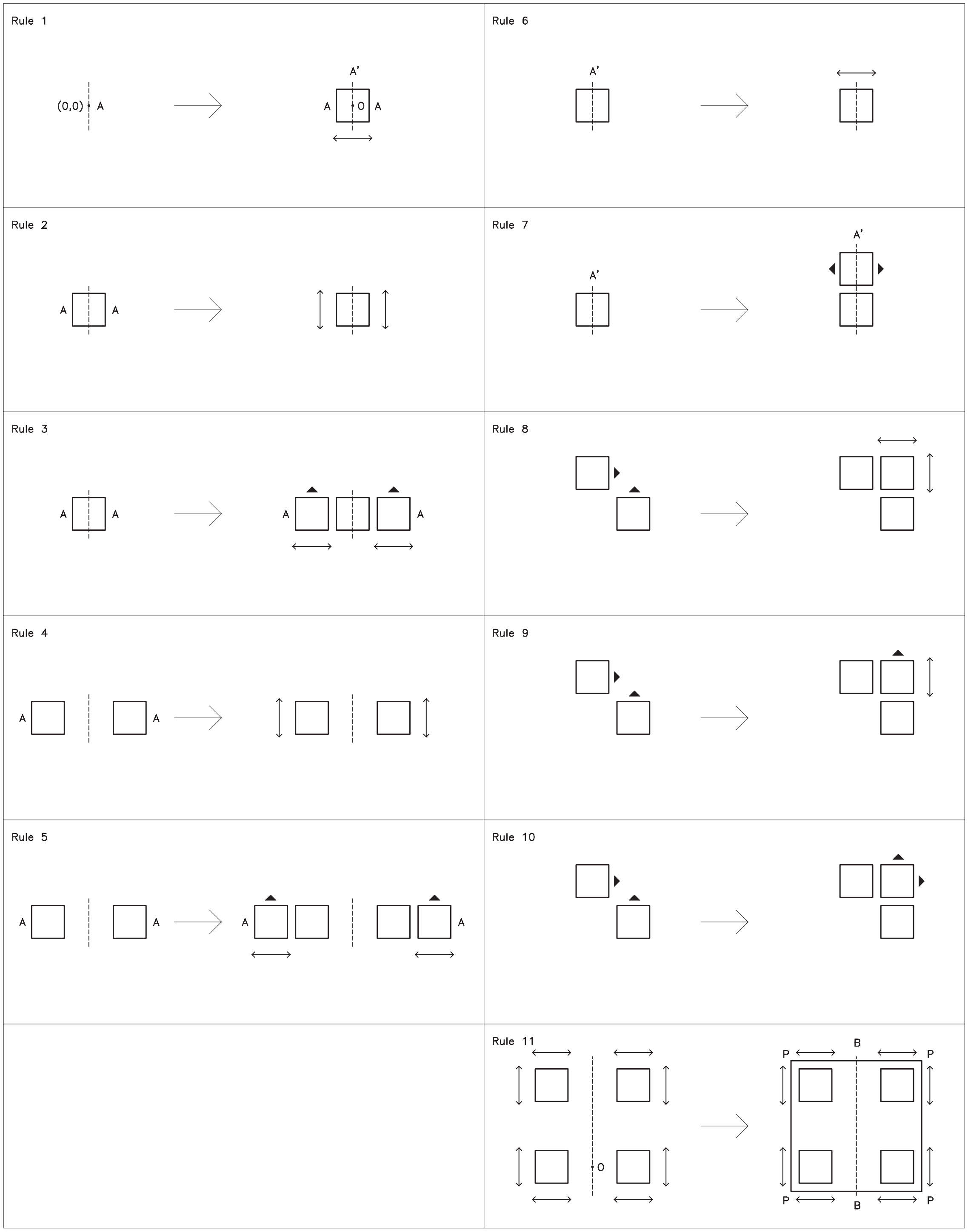


3.41 The three general categories of rules are applicable vs. inapplicable, constructive vs. destructive, and salient vs. deterministic.

Ideally, a grammar should provide the user with rules that are applicable, constructive and salient in nature. The SGML provides two descriptors, directive and rule-sets, towards achieving this goal. The directive is used to define an explicit sequence of rules which is called a macro. The sequence is dependant upon the success or failure of a given rule to apply to the drawing. The rule-set descriptor determines the availability of a set of rules at any point during the derivation of the grammar. Sets of rules are typically associated with stages of a grammar.

The sequencing of rules in a grammar is important. Should the rules of a grammar be executed out of order, the result could be a dead-end state. In this state, no subsequent rule application will bring the grammar to its conclusion. The Palladian grammar illustrates how a dead-end state can be created. The grammar is a parametric shape grammar that generates floor plans of Palladian villas. It is one of the first attempts to formalize a body of design work using shape grammars. The grammar generates two-dimensional floor plans in eight stages. The example presented here concentrates on the first stage of the grammar which determines the overall size of a bilaterally symmetrical grid. This grid also forms the basic skeleton for the plan.

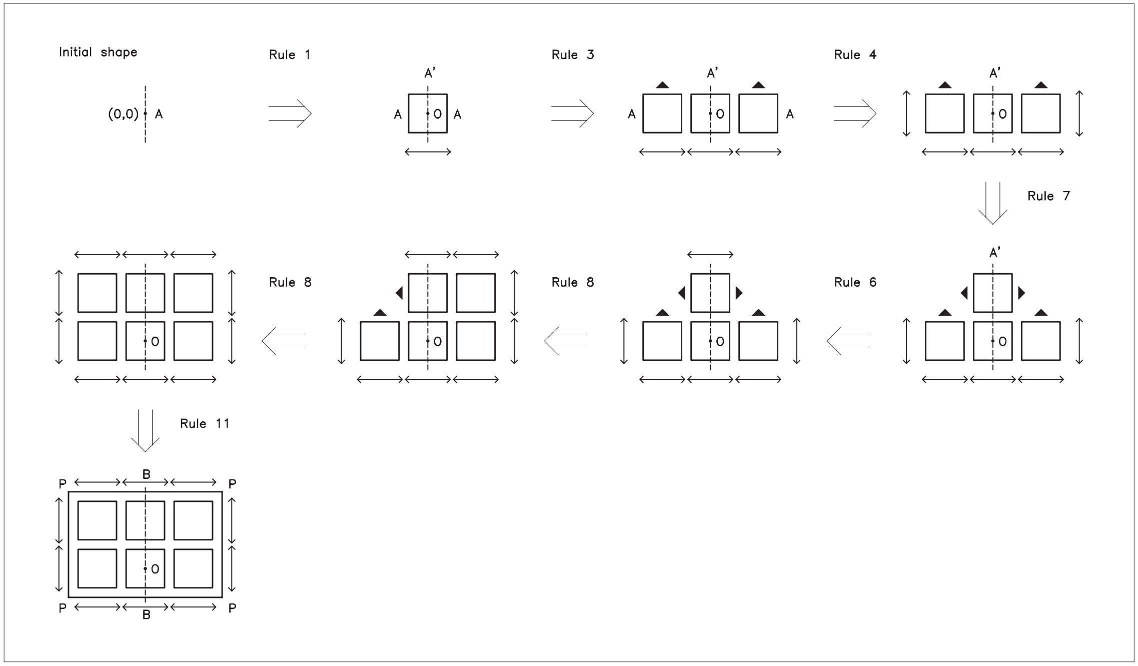
The rules for developing the bilaterally symmetrical grid in the Palladian grammar are show in figure 3.42. The rules can be organized into five basic groups. Rule 1 creates the first shape from the initial shape. Rules 2-5 deal with the expansion and termination of the overall width of the grid. Rules 6-7 controls the expansion and termination of the overall height of the grid. And rules 8-10 fill in the corners of the grid. Rule 11 modifies the labels so that the rules in the next stage can apply.



3.42 The rules from the first stage of the Palladian grammar which makes a bi-laterally symmetrical grid. Rule 1 creates the first shape from the initial shape. Rules 2-5 deal with the expansion and termination of the overall width of the grid. Rules 6 and 7 control the expansion and termination of the overall height of the grid. And rules 8-10 fill in the corners of the grid. Rule 11 modifies the labels so that the rules in the next stage can apply.

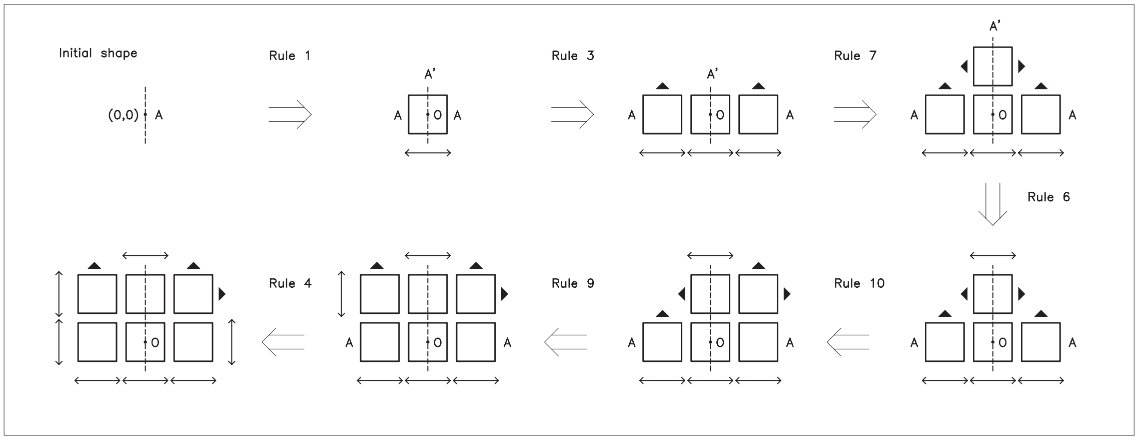
The goal of this stage of the grammar is to make a bilateral grid. The user should be given options to succinctly determine the size of the grid. But given the Palladian grammar rules however, it is not always clear what sequence of rule applications will produce a valid grid pattern. You will notice that some of the rules have the same left-hand schema such as rules 2-3, 4-5, 6-7 and 8-10. The similarity of some of the rules may cause confusion for the user making it difficult to determine which rule is appropriate. What sequence should the rules be applied to achieve a 3x5 grid? Should the width of the grid be set before the height? When is it appropriate to terminate the grid if the desired grid size is 3x5? Which rules terminate the grid? The grammar assumes the user will make the correct choices; it assumes the user understands the implicit ordering of the rules.

To help the user make the correct choices, a sample derivation is given to complement the rules (Mitchell 1990). Figure 3.43 shows one possible correct sequence to create a 3x5 grid. The user begins by expanding the grid horizontally and then vertically. At each step, the appropriate rule is selected so that all the labels are properly placed. The last step applies Rule 11 so that the next stage of rules can be used.



3.43 Correct derivation of the Palladian grammar to build a 3x5 bi-lateral grid. Rule 11 modifies the labels so that the next stage of rules can apply.

Since the grammar does not always make explicit what rule should be applied next, the user is free to apply any rule that is applicable. For instance, the user could apply the rules in a different sequence such as figure 3.44. As the derivation shows, instead of applying Rule 4 at the third step, rule 7 is applied. Instead of applying Rule 8 to fill the corners, Rule 10 and Rule 9 are used as an alternative. This sequence of rule applications also makes a 3x5 grid but the difference is that the labels are not appropriately placed. Although this sequence of rule applications for making a grid in the Palladian grammar is perfectly legal, the result is a dead-end state. Rule 11 can not be applied to the last step in the derivation because the labels are mismatched.



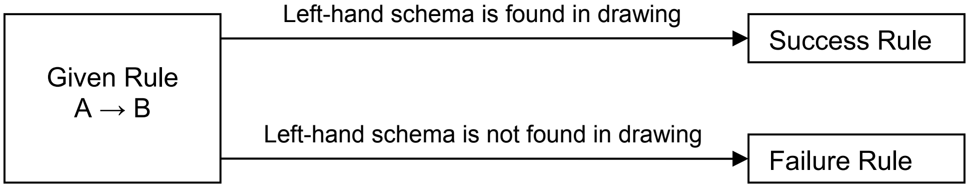
3.44 Legally correct derivation of the Palladian grammar to build a 3x5 bi-lateral grid but the end result is a dead-end state. No rule in the grammar can be applied after the last step.

In fact, the user is stuck in this dead-end state unless he starts over from the beginning because the grammar has no rules to allow him to undo the mistakes that were made in the beginning. The rules for designing a bilaterally symmetrical grid should be straightforward rules that succinctly determine the size of the grid. It should not give the user the ability to explore other design states, such as dead-end states, that are not fruitful for the rest of the design. If dead-end states do occur, the grammar should at least give the user the option of undoing what he did before.

The Palladian grammar demonstrates three problems: (1) the grammar has no explicit sequencing of rules; (2) it is possible to create a dead-end state; (3) lack of options to undo or adjust the effects of other rules. These problems can be resolved by re-designing the rules using the directive descriptor.

3.6.1 Directive

The directive descriptor adds an additional component to a rule that dictates which rule to apply next depending upon the success or failure of the current rule to apply. There are two options to the descriptor: success rule and failure rule (figure 3.45). The success rule determines which rule to apply next if the given rule was successfully applied. The failure rule determines which rule to apply next if the given rule fails to apply. This occurs when the left-hand schema of a rule does not exist in the drawing. The rule specified in the success rule or failure rule can be any rule in the grammar including itself. A rule that calls itself is a recursive rule.


Given Rule
A → B
Left-hand schema is found in drawing
Success Rule
Left-hand schema is not found in drawing
Failure Rule

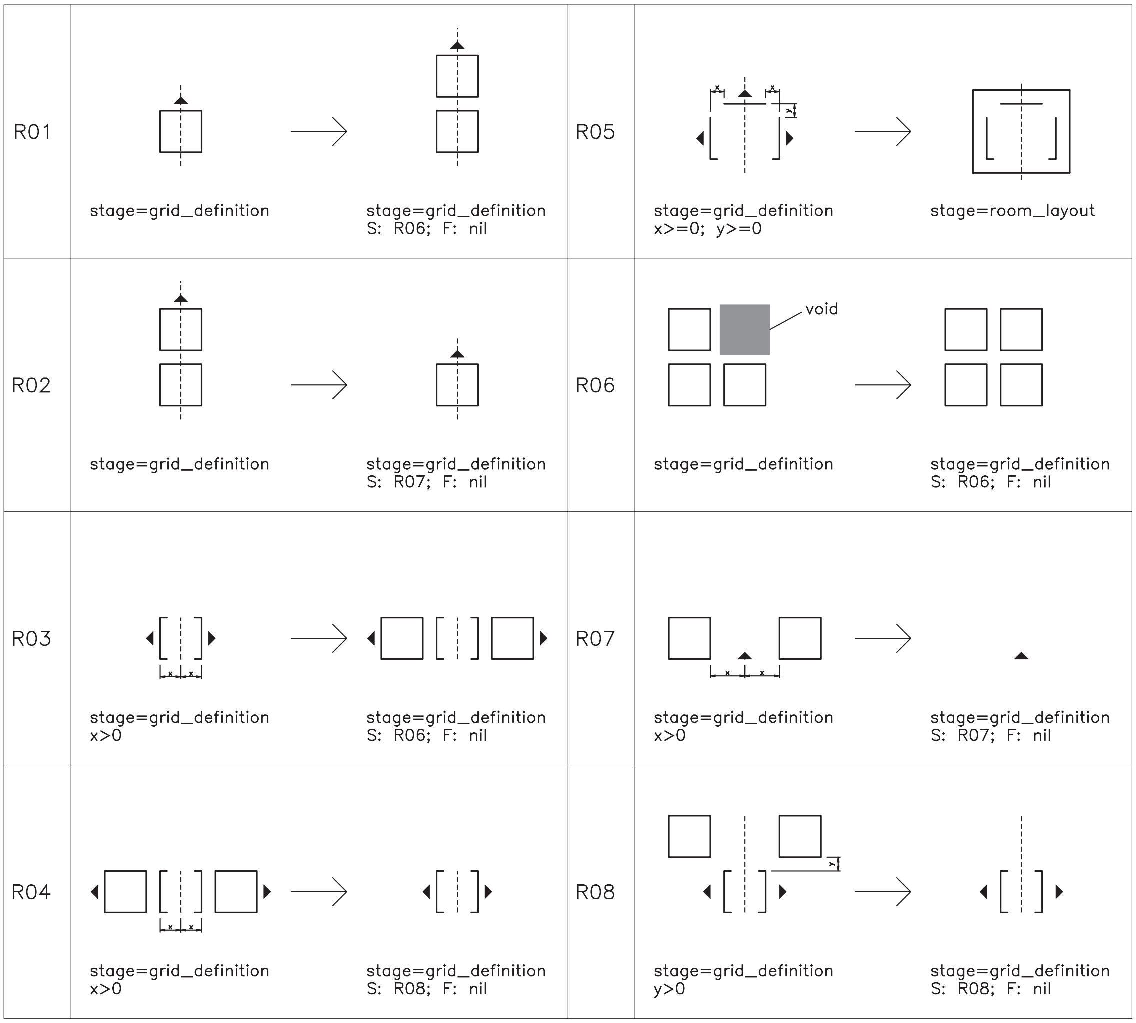
3.45 Diagram showing the success rule and failure rule of the directive descriptor.

By using the directive descriptor, the developer of a grammar is able to link a series of rules together to create a macro. A macro is composed of a primary rule and one or more secondary rules. The primary rule is the rule the user selects to trigger the use of the macro. The secondary rules are the rules that succeed the primary rule.

The Bilateral Grid grammar rewrites the rules from the first stage of the Palladian grammar to create a lucid set of rules for the user (figure 3.46). The grammar is composed of eight rules that give the user the ability to increase and decrease the size of the grid as they see fit. The directive descriptor is used to define macros or sequence of rules. Rule R01 and R06 defines one macro in this grammar. R01 is the primary rule and R06 is the secondary rule. Rule R02 and R07 define another macro as does rule R03 and R06. The last macro is defined by rules R04 and R08. R05 is only rule that does not make use of the directive convention. Rules R06, R07, and R08 are recursive rules because they call themselves using the directive.

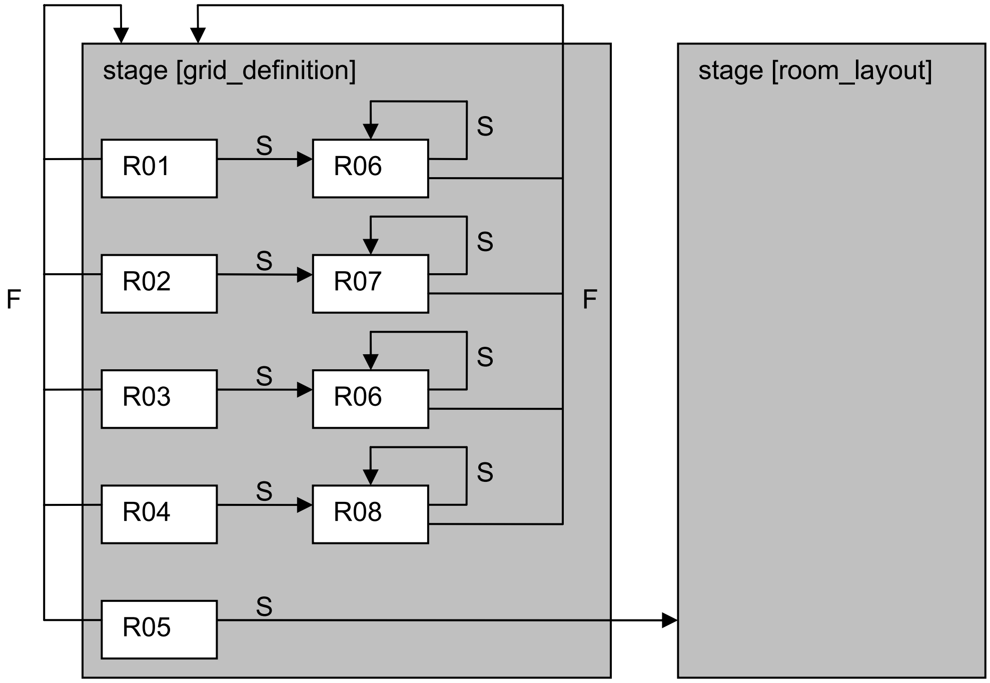
The use of macros makes the correct sequencing of rules explicit. For example, the macro R01+R06 links R01 with R06. R01 increases the height of the grid. After an increase in the grid height, the grammar ensures that all the other grid columns are increased as well by applying the recursive rule R06. The sequence of rules in a macro is not left up to the user but is embedded in the rules.

Rule 06 makes use of the zone descriptor to fill in the corners of the grid. The left-hand schema of the rule looks for three squares that forms an L-shape and uses the void zone to ensure that the remaining space inside the L is void of any shapes. Once that schema is found, the right-hand schema fills in the void space with another square. By repeatedly applying this rule all expansions of the grid will produce a rectilinear shape.



3.46 The Bilateral Grid Grammar is composed of eight rules which use the directive descriptor to define four macros. Macro R01+R06 and R02+R07 determine the overall height of the grid. Macro R03+R06 and R04+R08 determine the overall width of the grid and rule R05 changes the stage label to go to the next stage. The success rule of the directive is denoted with an “S” and the failure rule is denoted with an “F”.

Figure 3.47 is a diagram that shows the links created by the directive descriptor in the Bilateral Grid grammar. The gray boxes in the diagram represent stages of a grammar. In stage grid_definition, the user can select any of the eight rules (rule R06 is repeated). Success and failure rules are identified with the letter “S” and “F” respectively. If rule R01 is successfully applied, the next rule to apply is R06. Rule R06 will apply recursively until it fails. When that occurs, the user can again select any of the eight rules in this stage of the grammar because rule R06 does not specify a failure rule. Rule R05 changes the state label to go to the next stage which is room_layout.



3.47 Diagram showing the links between rules in the Bilateral Grid grammar. The diagram shows two stages represented as gray boxes. In stage grid_definition, the user can select any of the eight rules (rule R06 is repeated). Success and failure rules are identified with the letter “S” and “F” respectively. If rule R01 is successfully applied, the next rule to apply is R06. Rule R06 will apply recursively until it fails. When that occurs, the user can again select any of the eight rules in this stage of the grammar because rule R06 does not specify a failure rule.

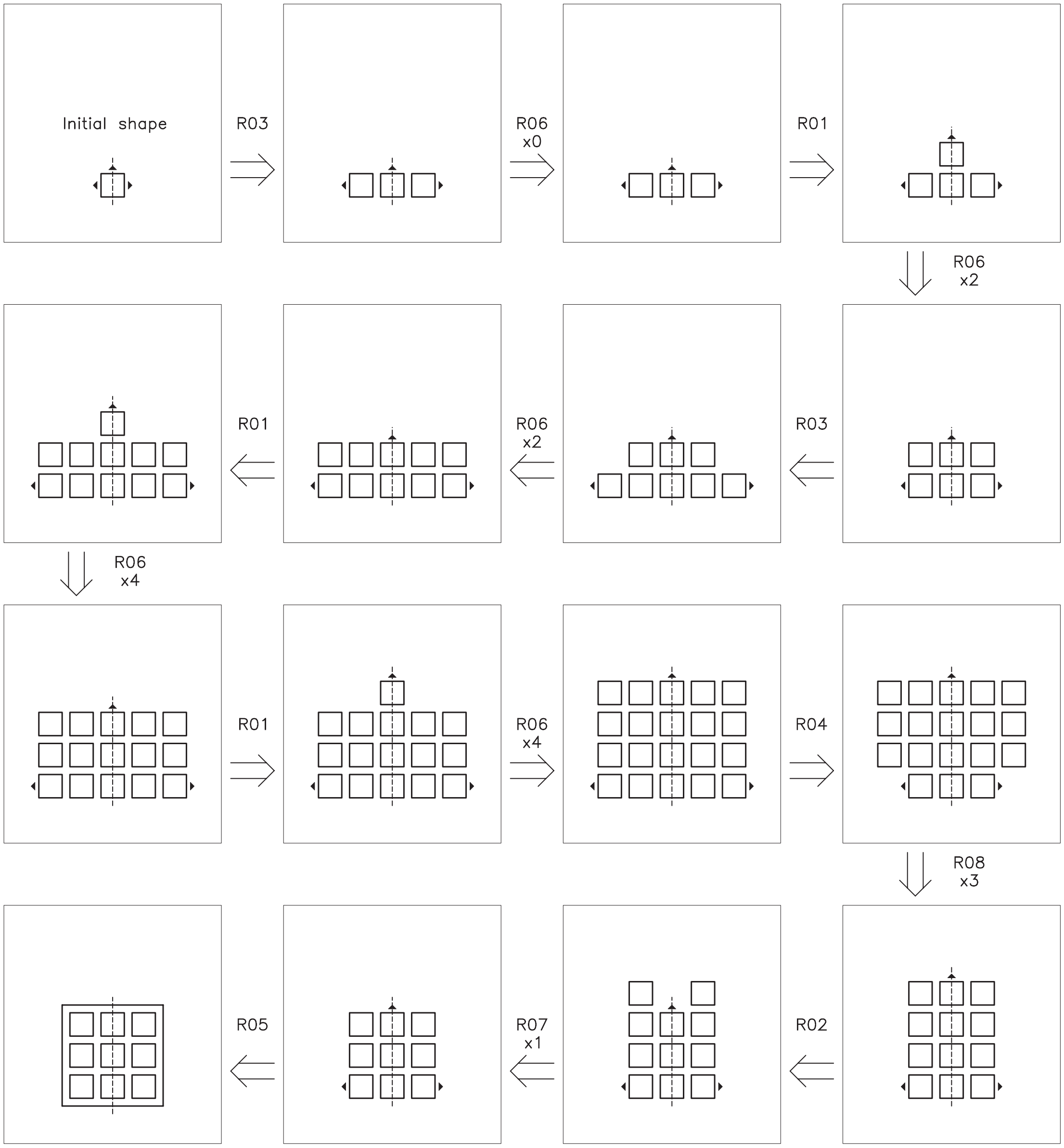
The rules in the Bilateral Grid grammar can be divided into three groups. The first group, R01+R06 and R02+R07, adjusts the overall height of the grid. R01+R06 increases the height whereas R02+R07 decreases the height. The second group, R03+06 and R04+R08, adjusts the overall width of the grid. R03+06 increases the width whereas R04+R08 decreases the width. The third group consists of only R05 which terminates this stage of the design. This set of rules gives the user the options to enlarge or reduce the size of the grid as they see fit. The design of these rules adds flexibility to the grammar by giving the user the ability to go back and adjust the design.

Figure 3.48 shows one possible derivation of the grammar to create a 3x3 grid. There are an infinite number of possible derivations since the grammar allows the user to modify the grid size until the user is ready for the next stage. Regardless of which rule is applied; there will never be a dead-end state.

The derivation begins by expanding the width of the grid through the application of rule R03. The directive in R03 requires that rule R06 be applied next but since the left-hand schema of R06 can not be found in the drawing, the rule does not apply. The second step in the derivation is to increase the height of the grid through the use of R01. R06 is applied immediately after R01 because R06 is linked, by the directive, with R01. R06 recursively applies itself twice to fill in the corners of the grid. The rule stops after two iterations because on the third iteration, the left-hand schema does not exist in the drawing which causes the failure rule to apply. In this case, there is no failure rule so control reverts back to the current stage where the user can pick any of the eight rules again. This directed process frees the user from having to determine what syntactic elements, such as labeled points and symbols, are required to achieve a valid design state. All the user needs to be concerned about is the design generated by the grammar which in this case is the overall size of the bilateral grid.

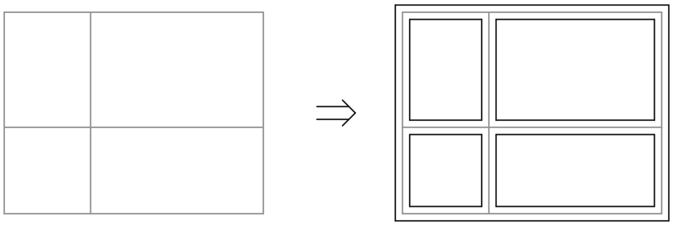
Should the grid be higher or shorter? Should the grid be wider or narrower? The derivation clearly shows that the Bilateral Grid grammar is flexible enough to allow the user to change his mind about the design. At step 11 of the derivation, the grid size is 5x4. The user at this stage decides that the grid is too big and applies rule R04 to reduce the size. After applying R04, R08 is recursively applied three times to reduce the width of all the other rows in the grid. The user decides the grid is still too big and reduces the size of the grid yet again with R02, which in turn applies R07, to arrive at the final size of a 3x3 grid. Finally, rule R05 is applied in order to move on to the next stage.

This process of evaluating the drawing to determine if the design has the desired consequences is an example of the see-move-see process. The user sees the bilateral grid and moves to alter the design until what he sees is satisfactory. The rules in the Bilateral Grid grammar give the user options for making the grid wider, narrower, taller or shorter. Through the see-move-see cycle the user has a conversation with the drawing (Schön 1983). The Palladian grammar, on the other hand, develops the bilateral grid in a linear sequence. The user can only make the grid larger. Should the user decide that the grid is too wide or too tall, there is no means to make the grid narrower or shorter. The Palladian grammar does not take into consideration the user’s experience in using the grammar. It is only concerned with being able to produce the appropriate design.



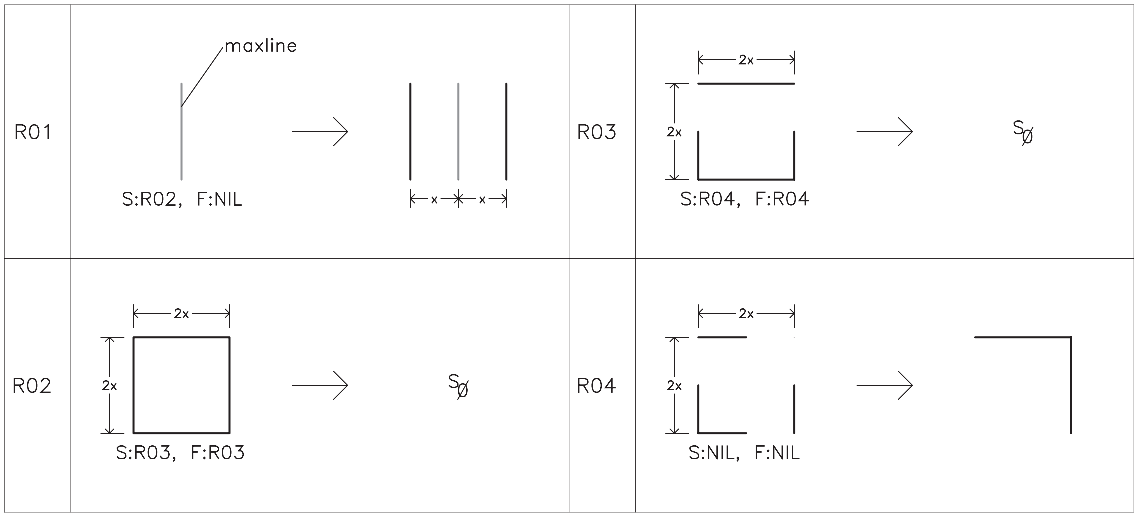
3.48 One possible derivation of the Bilateral Grid grammar. Because the grammar is designed to be flexible, there are many different possible derivations. The rules in the grammar make macros using the directive descriptor which explicitly sequences rules together. For example, after applying rule R01, R06 will always apply next.

In an architectural setting, the directive can be used to create macros that perform a single design transformation using several rules. We have already seen an example of this in section 3.5.2 where a single wall is transformed into a series of columns (figure 3.36). The directive can be used to link rules 3.37, 3.38 and 3.39 together to make one macro. Another example is shown in figure 3.49. Suppose the design task is to generate a series of walls using an underlying pattern as the centerline.



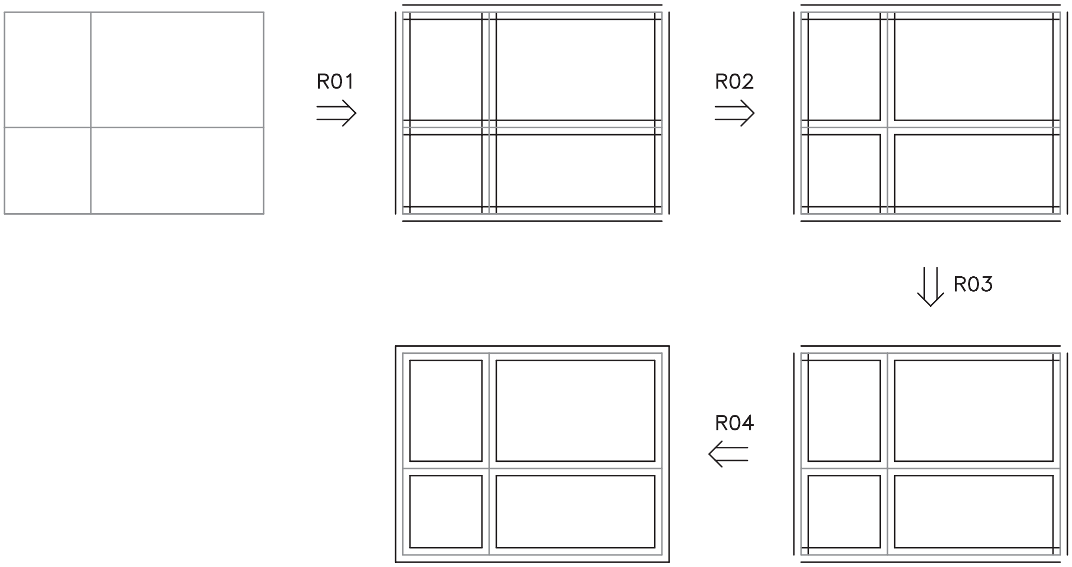
3.49 The derivation shows orthogonal centerlines used as a basis to generate the walls.

One method to achieve this effect is to offset the centerlines half the thickness of the walls and then trim off lines at the intersections where the walls join. The derivation for such a task is shown in figure 3.51. To produce this derivation, the four rules in figure 3.50 must be applied in order.



3.50 Four rules linked together using the directive description. Each rule is applied using the parallel apply-mode. Rule R01 offsets a pair of lines by distance x. R02 trims off the excess lines that occur in a cross intersection. R03 does the same for T intersections. And R04 trims and connects lines in an L intersection.

Each rule is linked to the next rule using the directive descriptor. Rule R01 is applied in parallel first then rule R02, R03, and R04. R01 is the primary rule and R02, R03, and R04 are the secondary rules. Since rule R03 and R04 are subshapes of R02, it is possible for rule R03 and R04 to apply in situations where rule R02 should have been applied instead. For this reason, if the rules are applied in a different order, the results could potentially be incorrect and produce a dead-end state. The appropriate derivation of the four rule macro is shown in figure 3.51.



3.51 Derivation showing the result of applying the macro as defined in figure 3.43. All rules applications are parallel applications.

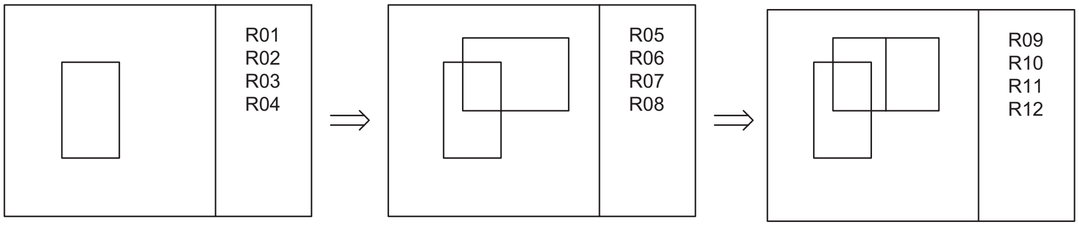
To achieve the same effect without the use of the directive descriptor would require the use of labeled points which act as state labels. Each rule would need to have a unique state label so that there is no confusion as to which rule to apply. For a grammar with few rules, this is a manageable task. But if there are hundreds of rules with hundreds of unique state labels, the grammar becomes daunting and unmanageable. The directive manages the complexity of having numerous state labels by placing the information of which rule to apply within the rule.

The main difference between using state labels and the directive descriptor are: 1) The mechanism for determining the next rule, when using the directive, is not in the drawing but in the rule. 2) The directive creates a macro which cannot be subverted by adding a rule with the same state label. 3) The directive has a failure component which allows an alternative rule to apply if a rule fails. The failure component can be used as a terminal case for recursive rules.

A macro is designed to start from the primary rule. One problem of using a macro, as defined by the directive descriptor, is that the macro will not necessarily work if the user picks a secondary rule to apply instead of the primary rule. Often times, the developer of a grammar wants to prevent the user from selecting the secondary rules of a macro because it can have negative effects on the design. One means of controlling this restriction can be accomplished by using the rule-set descriptor described in the next section.

3.6.2 Rule-set

The rule-set descriptor provides the user of a grammar with a set of rules at any point during the derivation of a grammar. This is achieved with a parallel description that contains a group of rules called a rule-set. A rule-set usually corresponds to one stage of a grammar. A change in the rule-set is the same as going from one stage of the grammar to another. A rule is able to modify the rule-set with three control options: set-rule, which defines the set of rules that are available, add-rule, which inserts additional rules into the rule-set and sub-rule, which removes rules that exist in the rule-set. An illustrative derivation is shown in figure 3.52 where the rule-set of each step is complete different from the previous step.

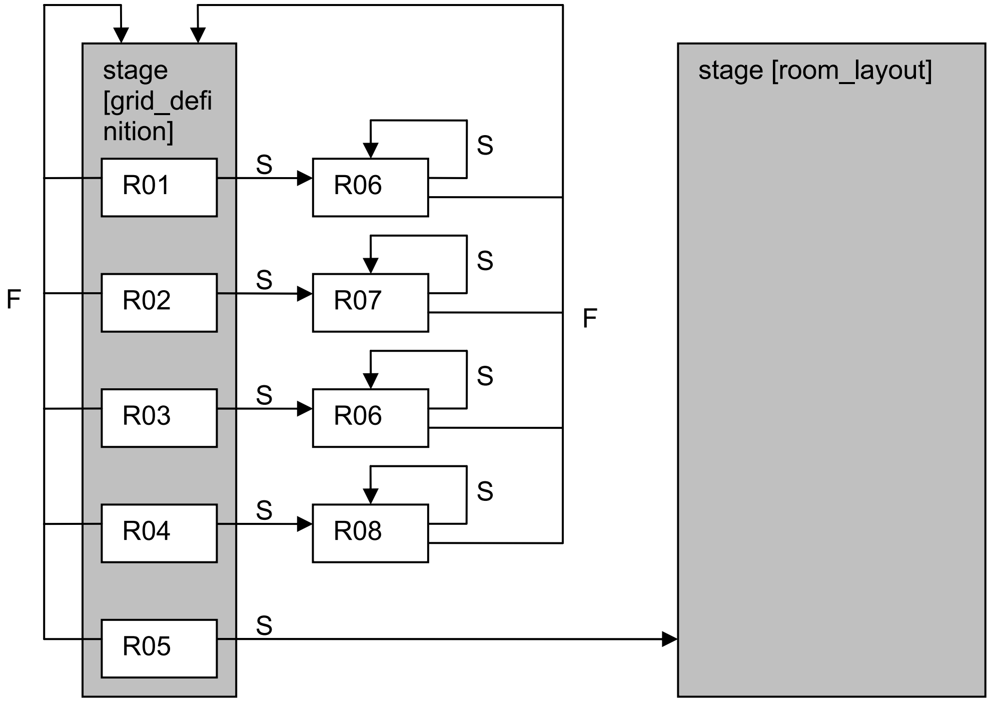


3.52 An illustrative derivation of a grammar showing show the rule-sets can change from step to step. Each step is composed of a drawing on the left and the rule-set on the right. In the first step, the rule-set is composed of R01, R02, R03, and R04. In the second step, the rule-set changes to R05, R06, R07 and R08.

The rule-set descriptor organizes a complex set of rules which aids the developer in writing a grammar. The appropriateness of the rules in the rule-set is determined by the developer. Ideally, the rule-set should include only rules that are applicable, constructive and salient. By presenting the user with all viable options, the rule-set mechanism alleviates the need for the user to determine which rules can and cannot apply (applicable vs. inapplicable rules).

The rule-set descriptor gives the developer the flexibility to use the same rule in different stages of the grammar. This is achieved by placing the same rules in different rule-sets. He can also prevent the user from using a rule, even if it is applicable, by not including the rule in the rule-set. This method is commonly used in conjunction with macros. By including only the primary rule of a macro in the rule-set, the developer of the grammar can prevent the user from selecting the secondary rules which may have negative effects on the drawing.

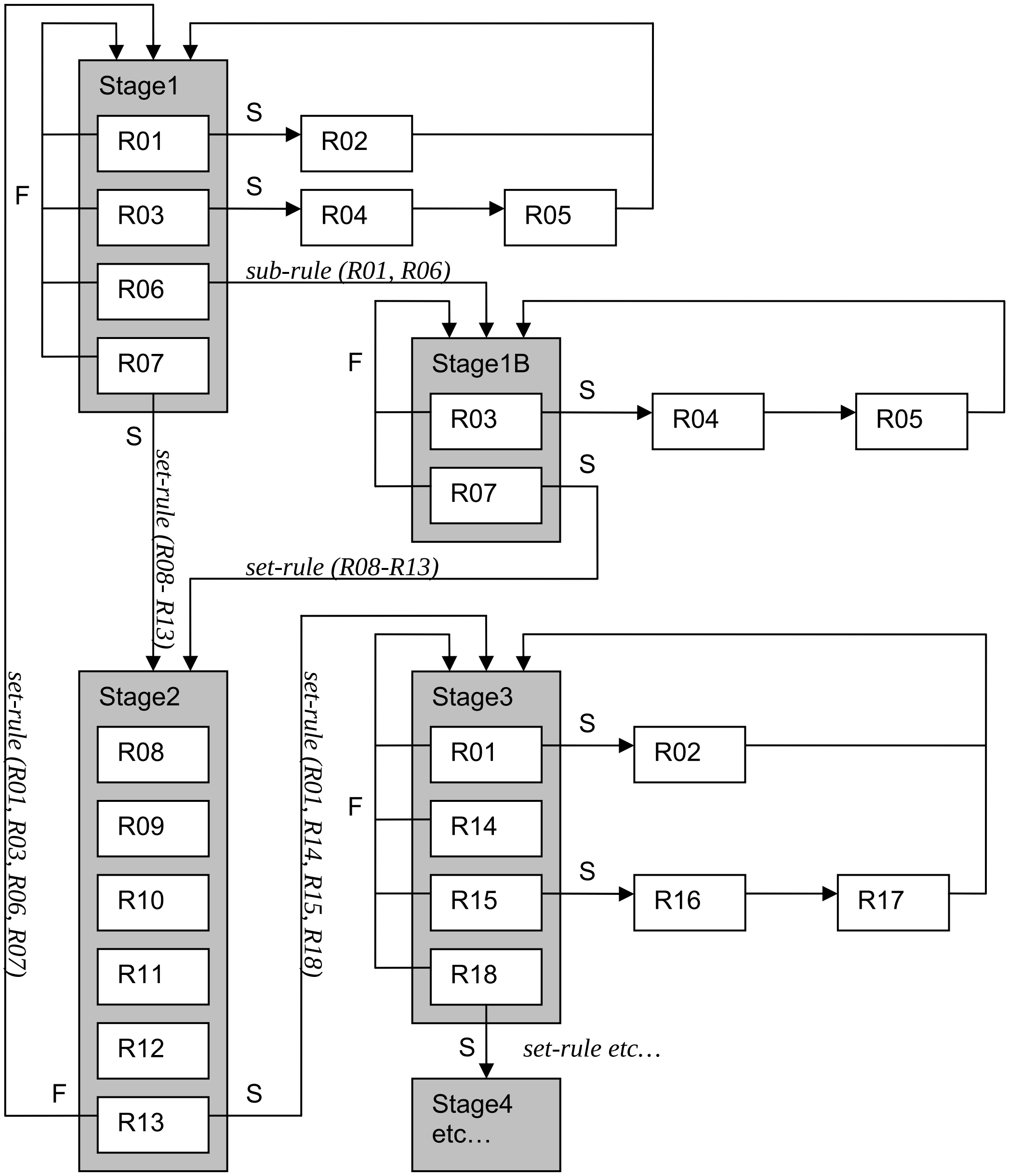
The diagram in figure 3.53 shows how the rule-set convention can be used to hide the secondary rules in the Bilateral Grid grammar. Instead of allowing the user to select from eight rules, the rule-set gives the user five salient rules to choose from. These rules correspond to the five basic options for manipulating the bilateral grid. The user can make the grid taller (R01) or shorter (R02), wider (R03) or narrower (R04), or go to the next stage (R05). The rule-set descriptor hides unnecessary details in order to manage a complex set of rules.



3.53 Diagram showing how the rule-set descriptor can be used in the Bilateral Grid grammar to make a salient set of rules for the user to choose from. In this case, the rule-set is used to hide the secondary rules of the macros. Stage grid_definition now has only five salient choices (rules R01-R05).

Figure 3.54 is a sample grammar that illustrates how rule-sets in combination with directives can be used to define and manipulate stages of a grammar. The illustration shows the first three stages of a sample grammar. In the first stage, the user has four options: R01, R03, R06, and R07. This stage also has two macros: R01+R02 and R03+R04+R05. Although the actual number of rules in the first stage is seven, the user can only select four of the rules. The rule-set descriptor acts as a filter to differentiate the salient rules from the deterministic rules of a macro.

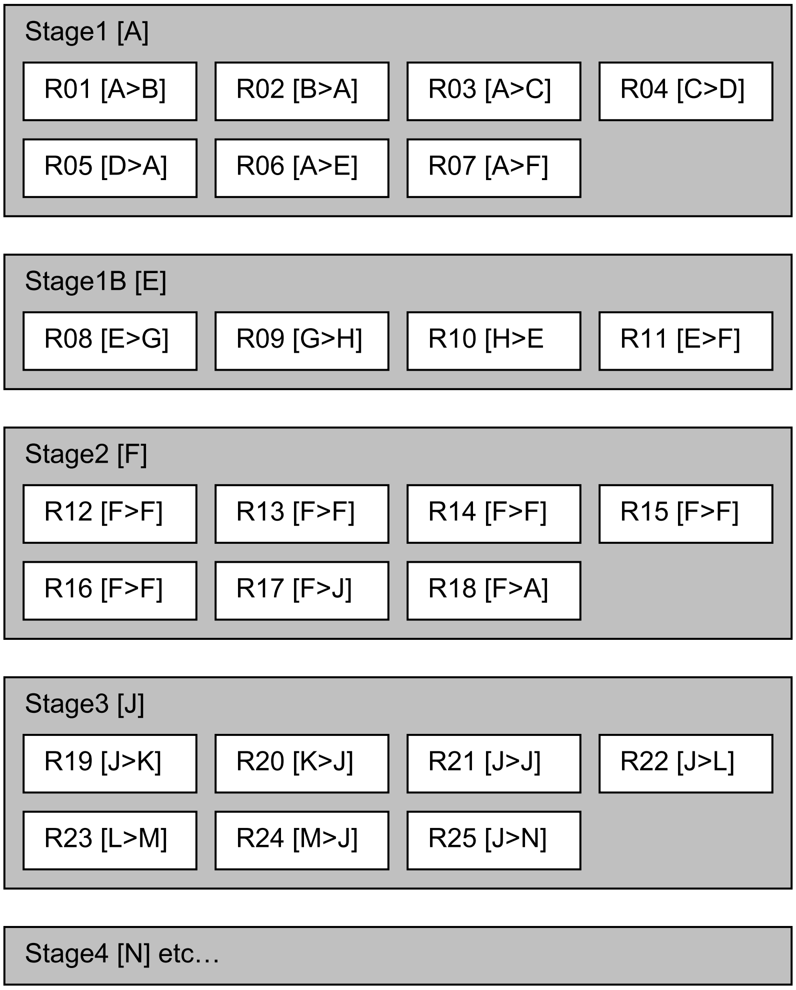
Two of the rules in the first stage change the rule-set with the rule-set controls. Rule R06 uses the sub-rule control to remove rule R01 and R06 from stage1. This might occur when a rule affects some changes to the drawing which makes other rules no longer useful. Rule R07 changes the rule-set to go to stage2 which is composed rules R08 through R13. This can be achieved by using set-rule or combining both the sub-rule and add-rule controls to define the new rule-set. Rule R13 changes the rule-set in stage2 to go to stage3 if it is applied successfully to the drawing. If rule R13 can not be applied to the drawing then the rule-set is changed to go back to stage1. Such a move might occur if certain conditions do not exist in the drawing and requires rules in another stage to implement the necessary changes.



3.54 Diagram showing how stages of grammar can be defined using the rule-set and directive descriptors. Each rule-set is represented by a gray box and the white rectangles inside are rules. The user can select any of the rules in the gray boxes. The lines connecting the rules together represent a link using the directive. A success rule is denoted by the letter “S” and a failure rule is denoted by the letter “F”. If there is no letter, then both the success and failure rules are the same. The rule-set controls used in the rules to change the rule-set are shown in italics.

The rule-set can also reuse rules from other stages of the grammar as shown in the stage3 of the diagram. Stage3 has seven rules but the rule-set only allows the user to select four of the rules. Two of the rules in the rule-set are the primary rules of a macro. One of the two macros, R01+R02, is a macro from stage1. The rule-set descriptor allows a rule to be used in any rule-set without having to modify the rule.

The rule-set descriptor is an alternative means of controlling the execution of a grammar. To generate the previous example without using the rule-set descriptor requires the use of state labels as shown in figure 3.55 where each stage corresponds to the same stage name in figure 3.54. Stage1 is composed of seven rules. Only four of the rules can apply at the initial state of this stage because they have the state label A. The other three rules have different state labels to correctly sequence a series of rules to behave like the directive descriptor.



Stage1 [A]

R01 [A>B] R02 [B>A] R03 [A>C] R04 [C>D]

R05 [D>A] R06 [A>E] R07 [A>F]

Stage1B [E]

R08 [E>G] R09 [G>H] R10 [H>E R11 [E>F]

Stage2 [F]

R12 [F>F] R13 [F>F] R14 [F>F] R15 [F>F]

R16 [F>F] R17 [F>J] R18 [F>A]

Stage3 [J]

R19 [J>K] R20 [K>J] R21 [J>J] R22 [J>L]

R23 [L>M] R24 [M>J] R25 [J>N]

Stage4 [N] etc…

3.55 Diagram showing how stages of the grammar shown in figure 3.54 can be defined using only state labels. The before and after states of each rule is shown in brackets next to the rule name.

To achieve the same effect as the directive, each rule must have a unique state label to insure that the correct rule is the only rule that can apply. For example, R01 changes the state label from A to B. The next rule that applies has to be R02 since it is the only rule in the entire grammar that has a state label B. Rules R03, R04, and R05 are implemented in the similar manner.

When using state labels, each stage is identified by its unique state label. Adding a rule into a stage requires modifying the state label of the rule to match the stage. For instance, in stage1B rules R08, R09, and R10 are the same rules as R03, R04, and R05. The only difference between these rules is the change in state labels. Instead of being able to use the same rules as those used in stage1, new rules are made that have the same schema but the state labels in the rules are changed to match stage1B's state label. Stage1 uses the state label A, stage 1B uses the state label E, and stage2 uses the state label F.

To implement the failure component of rule R13, as shown in figure 3.54, requires the addition of another rule that changes the state label from F to A (figure 3.55 R18). In stage3 of the grammar, rule R19 and R20 are the same as R01 and R02 except for the state label. Rules R01 and R02 have to be modified to accommodate the state label for stage3 which is J.

The rule-set descriptor simplifies the process of modifying state labels in rules by providing a separate mechanism that explicitly groups rules together. The key difference between using the rule-set descriptor and using state labels is the parallel description in rule-sets that succinctly displays for the user what rules are available for application. Both methods are control flow mechanisms and may be combined together to obtain even more detailed control over the execution of a grammar.

3.7 Summary of the meta-language descriptors

The shape grammar meta-language is composed of seven descriptors (figure 3.56) which are derived from the six phases of the rule application process. The first phase is the rule selection phase. In this phase, the user determines which rule to apply. There are two descriptors in the rule selection phase: rule-set and directive. Rule-sets provide a parallel description which contains all the rules that are applicable at each step of the grammar. The directive is another control mechanism associated with the rule that determines what rule to apply next depending upon the success or failure of the given rule to apply to the drawing.

The next phase is the drawing state phase which determines what portions of the drawing can be used for rule application. There are two descriptors in this phase: label-filter and focus. Label-filter is an option that modifies the drawing by filtering out all labeled shapes in the drawing that is not a labeled shape in the left-hand schema of a rule. The filtering process in effect alters what labeled shapes can be used to find a match between the schema and the drawing. Another means of manipulating where a rule can apply is by using focus lines which are special labeled lines. If the drawing contains an enclosed polygon composed of focus lines, then only the area inside the polygon can be used to find a match between the schema and the drawing. The focus lines control what areas a rule can apply whereas label-filters control what elements can apply.

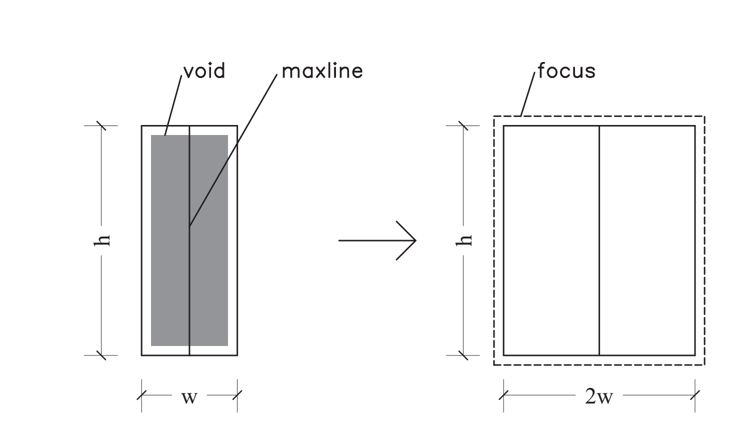
The next three phases, parameter requirements, transformation requirements, and contextual requirements, determine the matching conditions between a schema and the drawing. The requirements of all three phases must be fulfilled in order for the schema to have a subshape match in the drawing. The parameter requirements phase determines what values are given to the parameters of the schema in order to have a subshape match. The transformation requirements phase determines what transformations are needed to have a subshape match. There are no new meta-language descriptors in either one of these phases.

The contextual requirements phase determines if the context of the subshape is satisfactory. This phase has two descriptors: maxline and zone. The maxline descriptor states that a line in the subshape must be a maximal line in the drawing. If it is not, then the subshape is not a match. The zone descriptor associates an area of the schema with a predicate function. Not only must the schema be embedded in the drawing, the predicate, when applied relative to the subshape in the drawing, must also be true. One commonly used predicate function is the void function which states that the defined area must be void of any shapes. Another predicate function is exclude which returns true if specified labeled shapes are not in the defined area.

The last phase is the application method phase. This phase determines how a set of subshapes is applied to the drawing. Apply-mode is the only descriptor in application method phase. There are three possible options for the apply-mode descriptor. The first option is single, where the user can select one subshape from all possible subshape matches. The second option is parallel, where all the subshape matches are used. And the third option is random where a randomly chosen subshape is selected.

Figure 3.57 illustrates a sample rule that has all the components of the shape grammar meta-language. The components of the meta-language that are used in the schema are focus, zone and maxline. In this example, the left-hand schema makes use of the void zone and the maxline descriptor to find a rectangle that is void of shapes except for a maximal line placed in the center. The right-hand schema draws a rectangle twice as wide as the original and places a focus polygon around it in the drawing.
The remaining components of the meta-language are shown on the right-hand side of the rule. The first item, description, gives a general description of the rule. The next two items, constraint and transform, correspond to the parameter and transformation phases of the rule application process. The fourth item turns on or off the label-filter descriptor. The fifth item determines the apply-mode for the rule. The possible options are single, parallel or random. The success and failure items define the success and failure rules used in the directive descriptor. The last item defines any changes to the rule-set. The rule-set controls are set-rule, add-rule, and sub-rule.

Sample Rule



DESCRIPTION:	Sample rule illustrating the different conventions in the shape grammar meta-language.
CONSTRAINT:	g()
TRANSFORM:	t()
LABEL-FILTER:	[on, off]
APPLY-MODE:	[single, parallel, random]
SUCCESS:	[any rule]
FAILURE:	[any rule]
RULE-SET:	[set-rule (a list of rules)]
	[add-rule (a list of rules)]
	[sub-rule (a list of rules)]

3.57 A sample rule illustrating all the different descriptors in SGML.

The next section will describe how the new phases of the rule application process alter the original shape grammar formulas.

4.0 Implications of the meta-language

I have described characteristics of the six phases in the rule application process and their corresponding meta-language descriptors. Three of the phases, parameter requirements, transformation requirements, and application method are based on the original formulas (1-4) which are concerned with the mechanics of applying a rule. The new phases, rule selection, drawing state, and contextual requirements complement the original formula by addressing the decision making process when applying a rule. These phases can be incorporated together to generate a new set of formulas.

Traditionally, the matching conditions between a schema and the drawing are determined by the transformations, parameters, and parts relation (1). The meta-language introduces an additional matching constraint based on a predicate function. In order for a schema to have a subshape match in a drawing, all three components must be true. The transformation and parameters must produce a shape that is embedded in the drawing and the predicate function must be true. This changes the original formula as follows:

$$p(t(g(A))) \leq C \qquad (5)$$

Here the p() function is the additional predicate function of the contextual requirements phase. The meta-language has two descriptors in this phase: maxline and zone. The maxline descriptor tests to make sure the subshape line used for embedding are maximal lines. The zone descriptor tests a predicate function against a demarcated area of the subshape in the drawing. One commonly used predicate is void which checks if an area of the drawing is void of all shapes.

The contextual requirements phase modifies the matching condition between the schema and the drawing by altering the schema. The drawing state phase, on the other hand, modifies the matching condition by altering the drawing. The goal of the drawing state phase is to isolate portions of the drawing for rule application. This is achieved by hiding elements or areas of the drawing. This phase is composed of functions that “see” the drawing in a different context. The “see” function can be incorporated into the original formula in the follow manner:

$$t(g(A)) \leq s(C) \qquad (6)$$

Here the s() function is the function that changes what the rule sees in the drawing. The meta-language has two descriptors in this phase: label-filter and focus. The label-filter function determines what elements of a drawing can be used for a subshape match. When the label-filter descriptor is used, all labeled shapes in the drawing that are not in the left-hand schema of the rule will be removed. The focus descriptor, on the other hand, determines what areas of the drawing can be used for a subshape match. The areas that can be used for a subshape match are marked by enclosed polygons that use a special focus labeled line. All areas outside of the polygon cannot be used for a subshape match.

Putting the two modifications together we get the following formulas:

$$p(t(g(A))) \leq s(C) \quad (7)$$

$$C' = s(C) - p(t(g(A))) + t(g(B)) \quad (8)$$

Here the p() function is associated with the contextual requirements phase and the s() function is associated with the drawing state phase. And finally, to have a set of all possible subshapes:

$$\text{For all t and g such that } p(t(g(A))) \leq s(C) \quad (9)$$

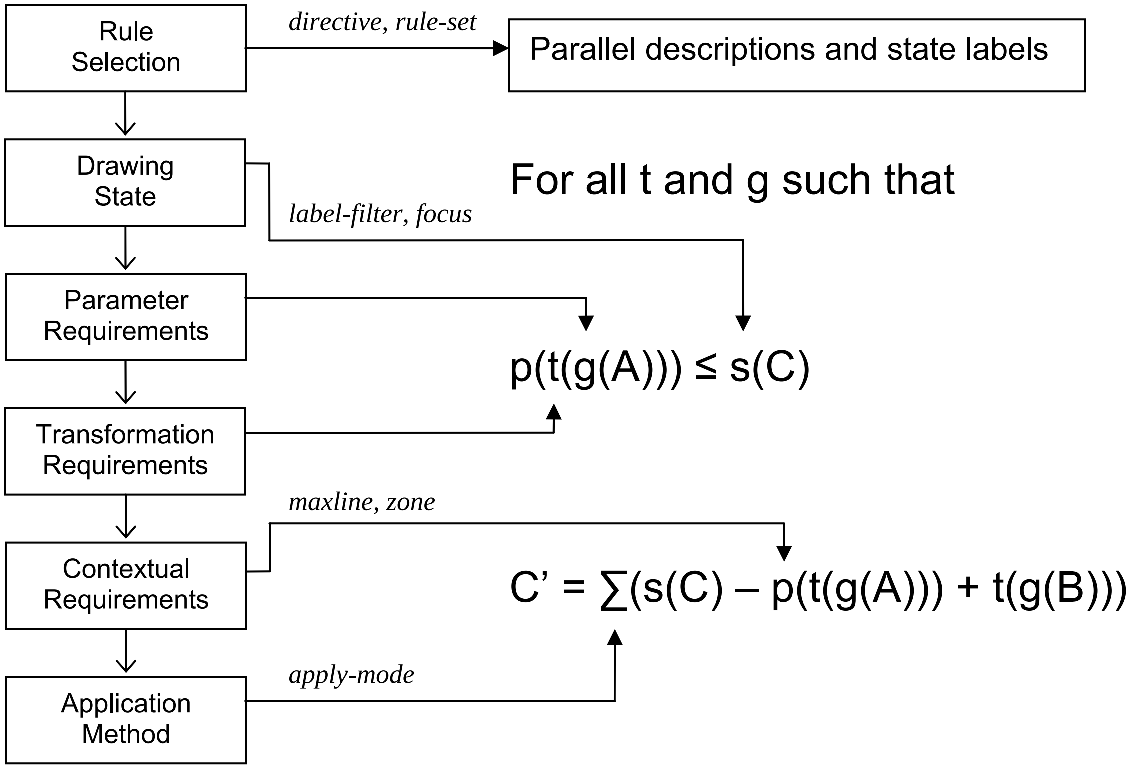
And to apply the entire set of subshapes to drawing C:

$$C' = \sum (s(C) - p(t(g(A))) + t(g(B))) \quad (10)$$

The formulas mentioned so far deal with the mechanics of the rule application process. This includes the drawing state, transformation requirements, parameter requirements, contextual requirements and application method phases. The rule selection phase does not apply to the above formulas because it deals with the overall control of the grammar.

The two descriptors in the rule selection phase are rule-set and directive. Rule-sets are an explicit mechanism for grouping rules in a grammar. The descriptor uses a parallel description to show the user which rules are available to choose from at different stages in the grammar. The directive is another control mechanism that dictates which rule to apply next depending upon the success or failure of a given rule to apply. Both descriptors are alternatives to the use of state labels which is the traditional method used in shape grammars to control the execution of the grammar.

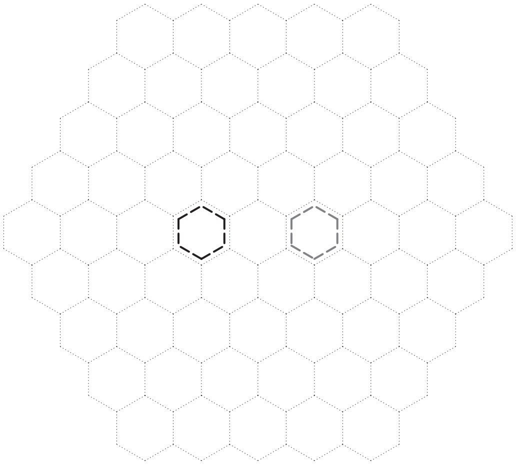
The shape grammar meta-language is a customization of the shape grammar language based on the six phases of the rule application process (figure 4.01). The six phases breaks down the decision process necessary to apply a rule. The descriptors in each phase provide an alternative means to work with shape grammars that emulates a specific method of design. The next two sections demonstrate how SGML can be used to describe and generate designs.



4.01 Diagram showing the relationship between the six phases of the rule application process and the shape grammar language. The seven descriptors of the meta-language are shown in italics.

5.0 Hexagon Path grammar

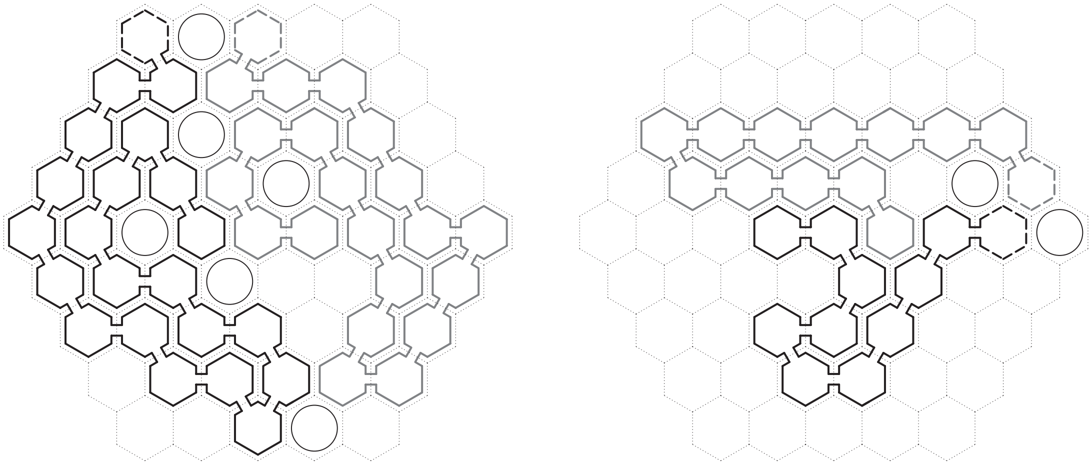
The Hexagon Path grammar demonstrates how the shape grammar meta-language can be used to play a simple board game. The rules of the grammar encode the algorithm for playing the game. The game is played between two players on a hexagonal grid composed of 61 hexagons (figure 5.01). The objective of the game is to design a path around your opponent so that he is unable to make a move. The first player that can not make a move loses the game. Each player begins with one hexagon shaped head piece near the center of the board.



5.01 The initial shape for the Hexagon Path grammar. Player1's head piece is represented by a black dashed line. Player2's head piece is represented by a gray dashed line.

There are three types of pieces in the game: head, tail, and block. The head piece is at the tip of the path and is the only piece that can place another head piece on the board. Once a new head piece is placed, the old head piece becomes part of the tail which is all the pieces in the path except for the head. The block piece acts as an obstacle and is represented as a circle. Although the Hexagon Path grammar is described as a game, it can also be considered as a methodology to generate a design on a hexagon grid using two paths. It just so happens that this method of design alternates between two different paths and stops when one path can not be extended.

The two players take turns making a move. During each player's turn, the player can either place a new head piece to extend the path or place a block piece. These pieces can only be placed in an empty hexagon grid adjacent to the head piece. If there are no empty hexagon grids surrounding the head piece, that player has lost the game (figure 5.02).

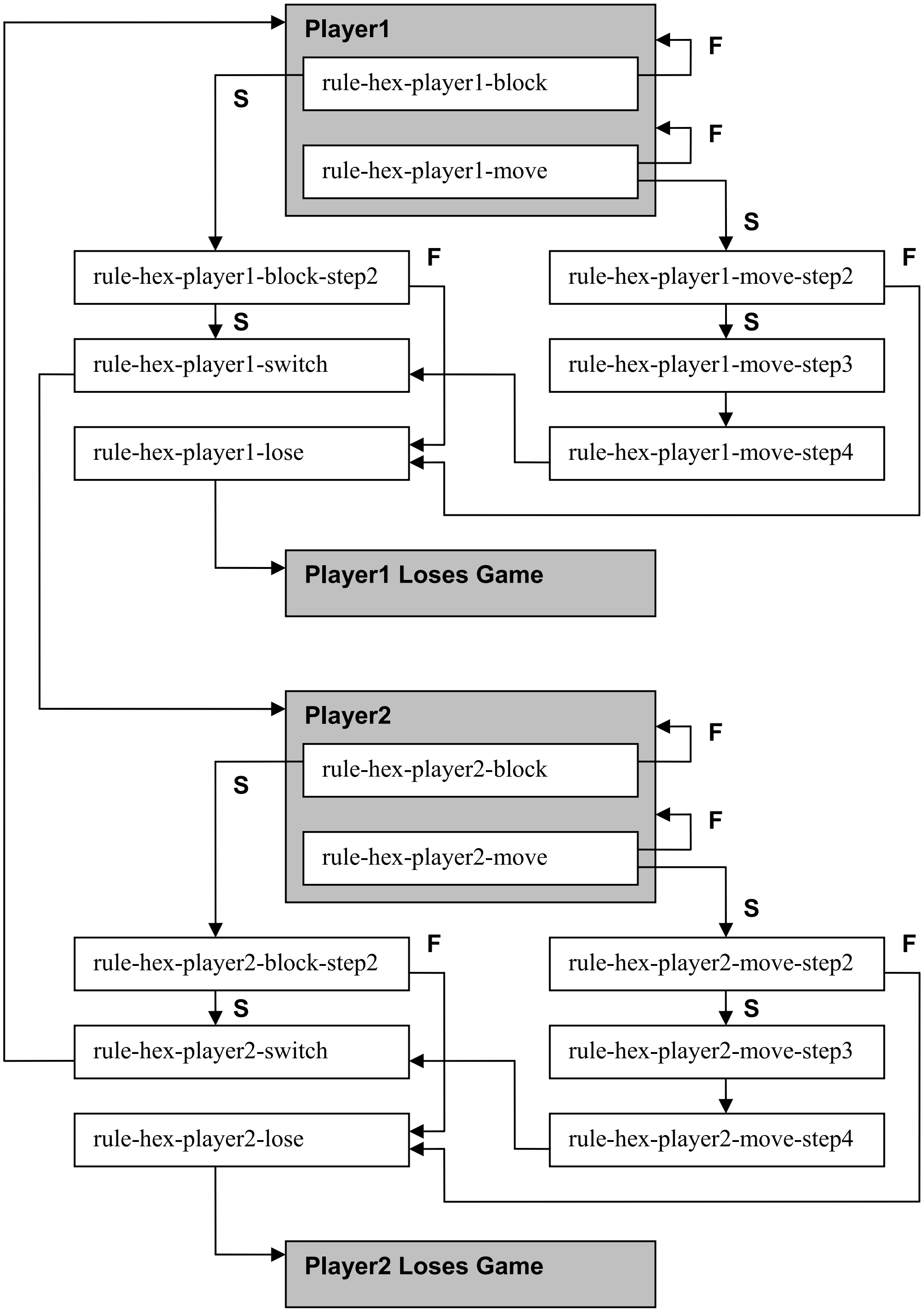


5.02 Two sample games of the Hexagon Path grammar. In the left game board, the first player (black lines) loses the game. In the right game board, the first player wins the game.

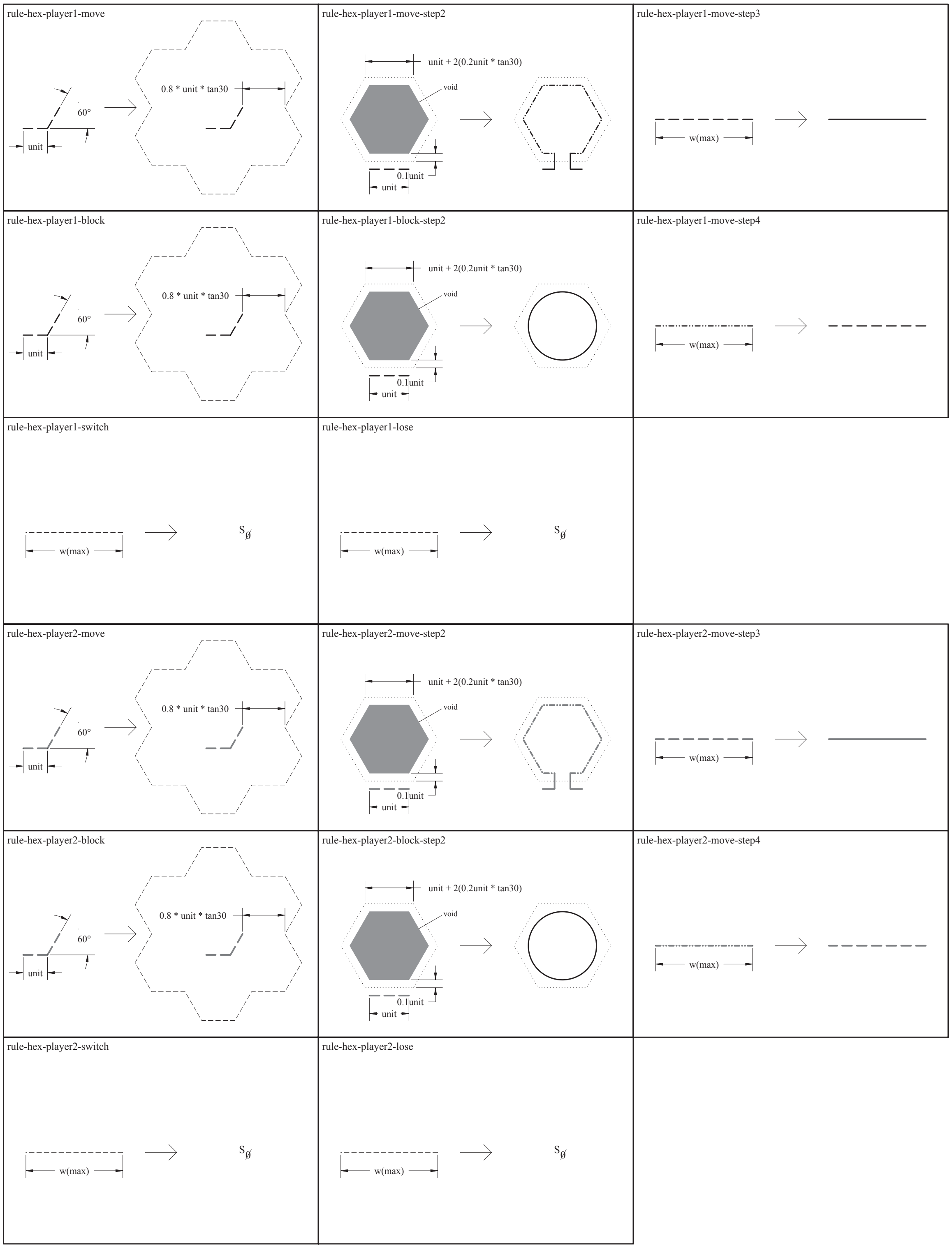
In order for the Hexagon Path grammar to emulate the game, the grammar must do two things. First it must be able to determine what legal moves a player can make. Second it must also be able to alternate control between the two players. These two tasks can be achieved using the SGML descriptors.

There are two possible legal moves a player can make. He can either place a new head piece or place a block piece. In either case, the space in which the piece will be placed must be an empty hexagon adjacent to the current head piece. If there are no empty hexagon spaces, then that player has lost the game. The Hexagon Path grammar uses the void zone to determine if a hexagon grid is empty or not. The directive is used to control which rule to apply next depending on whether or not there are any empty spaces adjacent to the head piece. If the player can make a move, the rule-set is modified so that the other player makes the next move. If the player can not make a move, all the rules are removed from the rule-set to signify that he has lost the game.

Figure 5.03 shows the two stages of the Hexagon Path grammar. The grammar begins with stage Player1. At this stage, the first player has two rules. One rule places a block piece and the other places a new head piece. If the selected rule is applied successfully to the drawing, then the rule-set is changed so that the next player can make a move. If a player fails to be able to place a new head piece or block piece, then that player has lost the game. The key rule that makes this decision is *rule-hex-player1-block-step2* or *rule-hex-player1-move-step2*. These rules use the directive to make a branch in the decision tree. If the rule can be applied, then apply the rule and switch sides. If the rule can not be applied then the game is over for that player. The options in stage Player2 are the same as those in stage Player1 except that they are made for the second player. An abridge version of the rules is shown in figure 5.04.



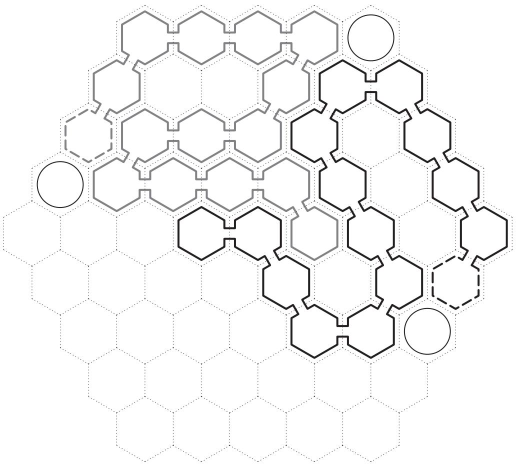
5.03 Diagram showing the two stages, Player1 and Player2, and the sequence of rules in the Hexagon Path grammar. The gray boxes are stages and the white rectangles are rules. The arrows represent a link to the next rule or stage. Success and failure rules are identified with the letter "S" and "F" respectively. If there is no letter next to the arrow then both the success rule and the failure rule are the same.



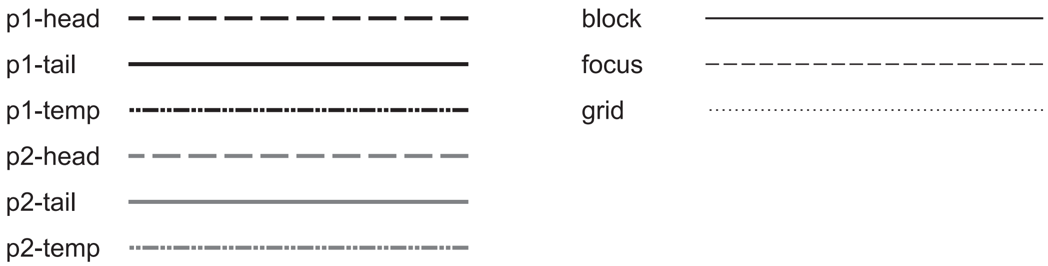
5.04 The abridged version of the rules for the Hexagon Path grammar. The first eight rules are for player1 and the second eight rules are for player2. For a detailed view of the rules see section 5.2.

5.1 Derivations of the Hexagon Path grammar

A sample game of the Hexagon Path grammar, as shown in figure 5.05, will be explained in this section. An overview of the process is shown in figure 5.07. Different linetypes and lineweights are used to differentiate the nine labeled lines used in the grammar (figure 5.06). Lines used by player1 begin with “p1” and are black in color. Lines used by player2 begin with “p2” and are gray in color. Each player has three types of labeled lines: head, tail, and temp. The head label is for the head piece, the tail label is for the tail pieces, and the temp label temporarily holds the position of the new head piece. A block labeled line represents block pieces. A grid labeled line represents the grid of the game board. And a focus labeled line represents the focus descriptor.

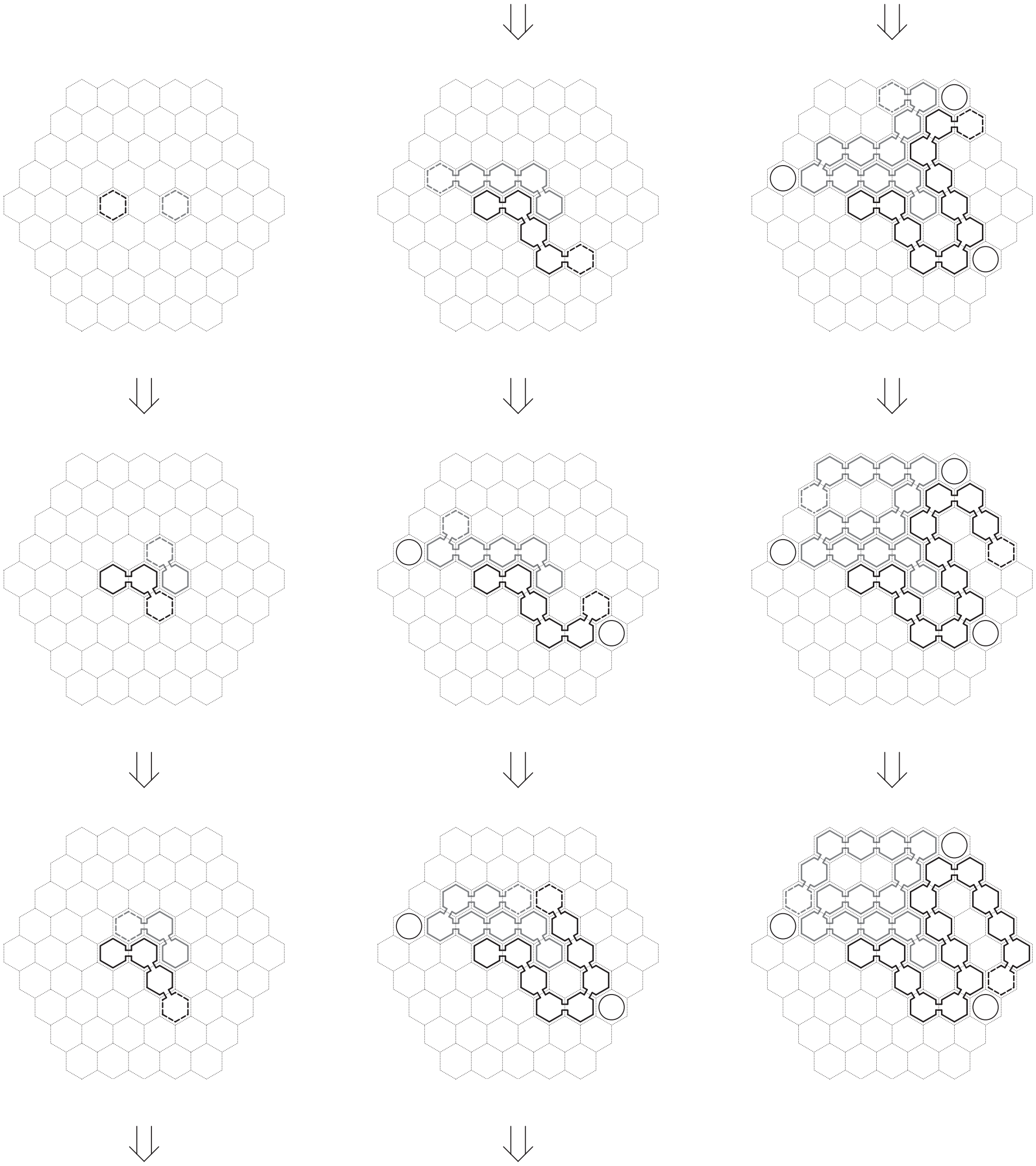


5.05 A sample game using the Hexagon Path grammar.



5.06 The linetype chart showing the nine different linetypes used in the Hexagon Path grammar.

What follows is a step by step derivation of the Hexagon Path grammar. Each frame is composed of two parts: the drawing and its corresponding rule-set parallel description. Between each frame, next to the arrow, is the rule selected to go to the next frame. There are a total of 41 steps in this derivation. The game begins with the initial state of the game board is shown in frame 1 of figure 5.08.



5.07 Overview derivation for the Hexagon Path grammar.

Frame 1: A hexagon grid composed of 61 hexagons is the game board for the game. One head piece from each player is placed near the center as shown. The initial shape also sets the variable “unit” which is the distance of one side of the hexagon head piece.

Frame 2: Player1 decides to place a head piece by selecting *rule-hex-player1-move* which is the primary rule of a macro. The rule looks for two head piece lines that are at a 60° angle and places a focus polygon that surrounds all the hexagon grids adjacent to the head piece. Because of the directive, the next rule applied in the macro must be *rule-hex-player1-move-step2*.

Frame 3: *Rule-hex-player1-move-step2* uses the void zone to determine if any of the hexagon grids are empty. Because of the focus polygon, this rule by default will check all the adjacent hexagon grids. If an empty hexagon is found, a p1-temp hexagon is placed in the hexagon grid and the p1-head lines are changed to p1-tail lines. At this step in the derivation, player1 selects the East empty hexagon and therefore *rule-hex-player1-move-step3* is applied next. If an empty hexagon grid is not found, then player1 has lost the game and the failure rule, *rule-hex-player1-lose*, is applied instead.

Frame 4: The p1-temp label is used to temporarily hold the position of the new head piece. *Rule-hex-player1-move-step3* replaces all the p1-head lines with p1-tail lines. Because of the focus polygon, this rule will affect only the lines within the polygon. This rule effectively changes the old head piece into a tail piece. *Rule-hex-player1-move-step4* is applied next because of the directive.

Frame 5: *Rule-hex-player1-move-step4* replaces all the p1-temp lines with p1-head lines. This creates the new head piece. The directive states that *rule-hex-player1-switch* is applied next.

Frame 6: The last rule in the macro is *rule-hex-player1-switch*. This rule removes the focus polygon and changes the rule-set so that player2 can make a move after the macro has finished.

Frame 7: Player2 has the same rules as player1 except they work on player2 pieces as opposed to player1 pieces. Player2 makes a path in the North-west direction.

Frame 8: Player1 moves to the South-east position.

Frame 9: Player2 moves in the West direction.

Frame 10: Player1 moves in the South-east direction.

Frame 11: Player2 moves in the West direction.

Frame 12: Player1 moves in the East direction.

Frame 13: Player2 moves in the West direction.

Frame 14: Player1 decides to place a block piece by selecting *rule-hex-player1-block* which is a primary rule of a macro. A focus polygon is placed around all the hexagon grids adjacent to the head piece. *Rule-hex-player1-block-step2* is applied next because of the directive.

Frame 15: *Rule-hex-player1-block-step2* looks for an empty hexagon in order to place a block piece. Player1 decides to place a block piece in the East direction.

Frame 16: The last step in the macro is to remove the focus polygon and change the rule-set so that player2 can make a move after the macro has finished. This step uses the same rule, *rule-hex-player1-switch*, as the one used in the macro *rule-hex-player1-move*. This is clearly shown figure 5.03.

Frame 17: Player2 places a block piece in the West direction. This move effectively divides the playing field into two halves. The top half has 27 hexagon spaces while the bottom half only has 22 hexagon spaces.

Frame 18: Player1 moves in the North-east direction. Player1 moves into the top portion of the board because there are more hexagon spaces than the bottom half.

Frame 19: Player2 moves in the North-east direction. Player2 also moves into the top half of the game board. If player2 had decided to go into the bottom half, he would have automatically lost the game because the maximum number of move he can make is 22 while player1 can make up to 27 moves.

Frame 20: Player1 moves in the North-west direction. The strategy for player1 is to reduce the amount of space player2 has to maneuver. To this effect, player1 moves toward player2.

Frame 21: Player2 moves in the East direction. Player2 has the same strategy and moves toward player1.

Frame 22: Player1 moves in the North-west direction.

Frame 23: Player2 moves in the East direction.

Frame 24: Player1 moves in the North-west direction.

Frame 25: Player2 moves in the North-east direction.

Frame 26: Player1 moves in the North-east direction.

Frame 27: Player2 moves in the North-east direction.

Frame 28: Player1 places a block piece in the North-east direction. This effectively ends the game for player2. The maximum number of moves player2 can make is 7 while player1 can make 9 moves.

Frame 29: Player2 moves in the West direction. This is the only move player2 can make.

Frame 30: Player1 moves in the East direction.

Frame 31: Player2 moves in the West direction.

Frame 32: Player1 moves in the South-east direction.

Frame 33: Player2 moves in the West direction.

Frame 34: Player1 moves in the South-east direction.

Frame 35: Player2 moves in the South-west direction.

Frame 36: Player1 moves in the South-east direction.

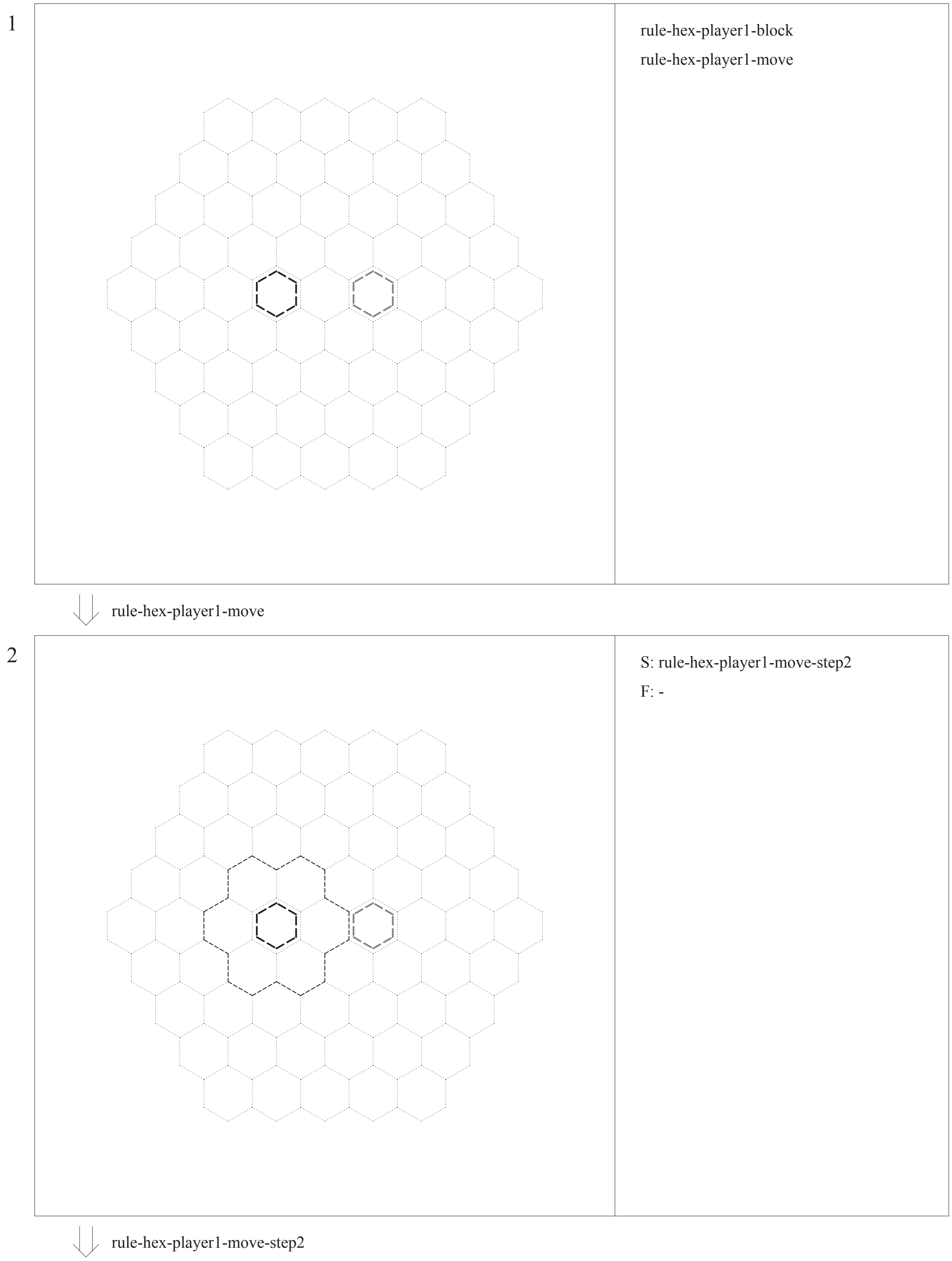
Frame 37: Player2 moves in the South-west direction.

Frame 38: Player1 moves in the South-west direction.

Frame 39: Player2 tries to make a move using *rule-hex-player2-move*. A focus polygon is placed around all adjacent hexagon spaces.

Frame 40: *Rule-hex-player2-move-step2* fails to apply in this situation. The failure rule, *rule-hex-player2-lose*, is applied next which terminates the game for player2.

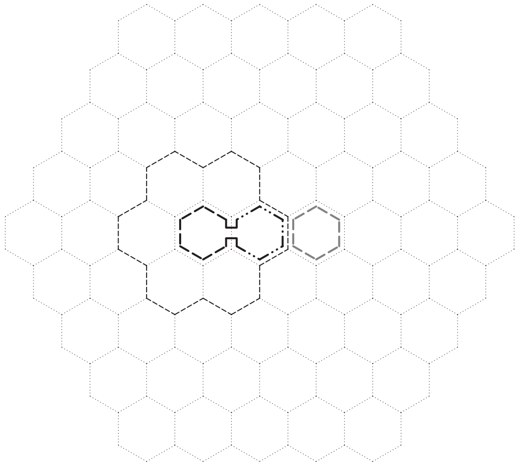
Frame 41: All the rules have been removed from the rule-set which signifies that Player2 has lost the game.



5.08 Derivation of the Hexagon Path grammar.

⇓ rule-hex-player1-move-step2

3

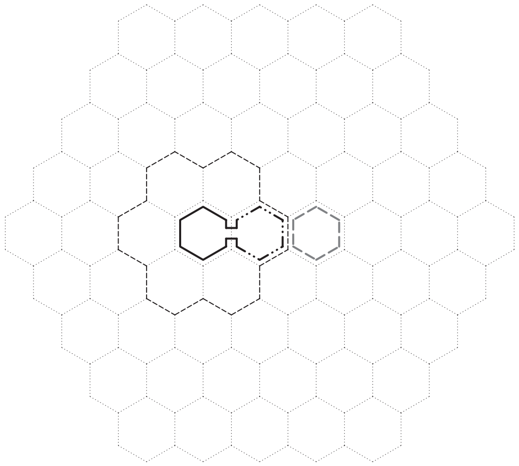


S: rule-hex-player1-move-step3

F: rule-hex-player1-lose

⇓ rule-hex-player1-move-step3

4



S: rule-hex-player1-move-step4

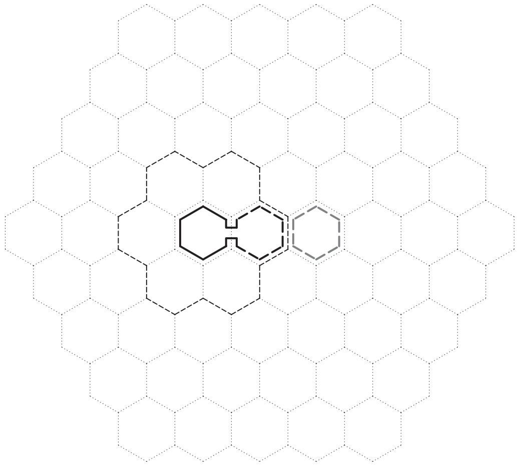
F: rule-hex-player1-move-step4

⇓ rule-hex-player1-move-step4

5.08 Derivation of the Hexagon Path grammar continued.

↓ rule-hex-player1-move-step4

5

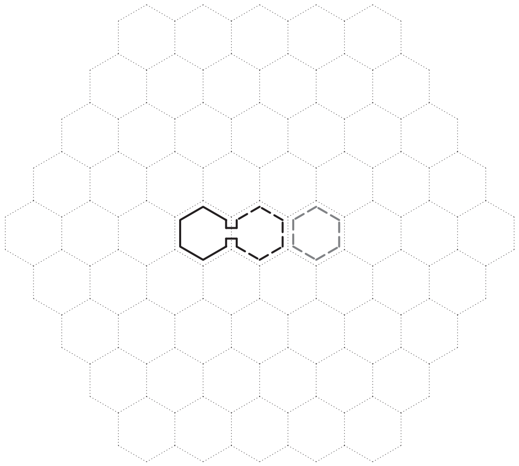


S: rule-hex-player1-switch

F: rule-hex-player1-switch

↓ rule-hex-player1-switch

6



rule-hex-player2-block

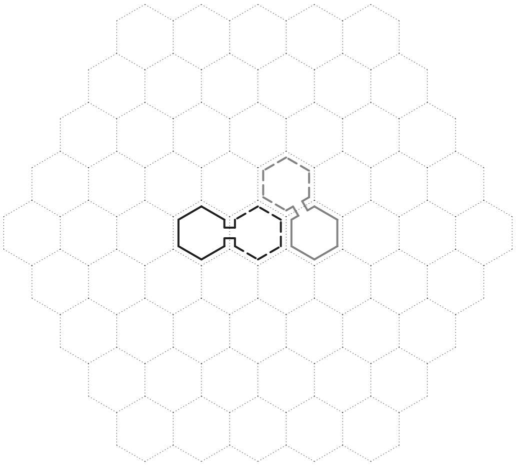
rule-hex-player2-move

↓ rule-hex-player2-move, -move-step2, -move-step3, -move-step4, -switch

5.08 Derivation of the Hexagon Path grammar continued.

↓ rule-hex-player2-move, -move-step2, -move-step3, -move-step4, -switch

7

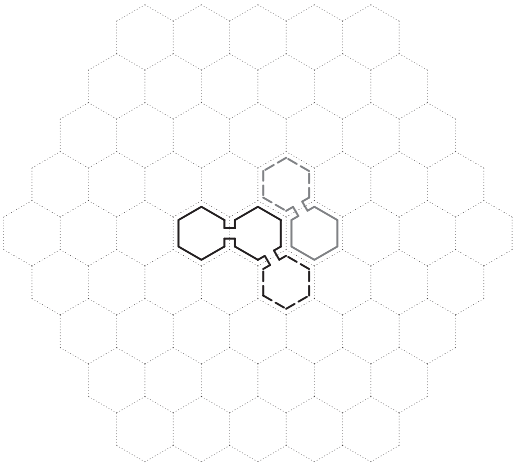


rule-hex-player1-block

rule-hex-player1-move

↓ rule-hex-player1-move, -move-step2, -move-step3, -move-step4, -switch

8



rule-hex-player2-block

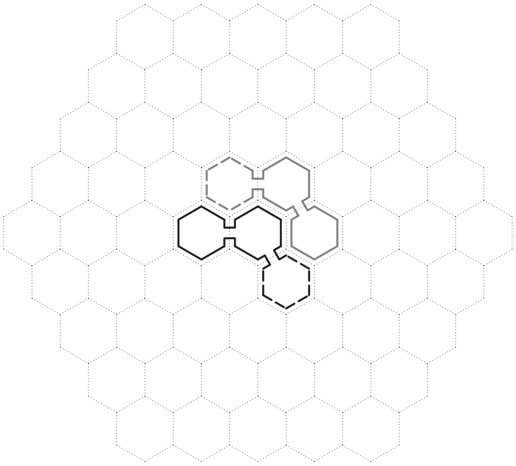
rule-hex-player2-move

↓ rule-hex-player2-move, -move-step2, -move-step3, -move-step4, -switch

5.08 Derivation of the Hexagon Path grammar continued.

rule-hex-player2-move, -move-step2, -move-step3, -move-step4, -switch

9

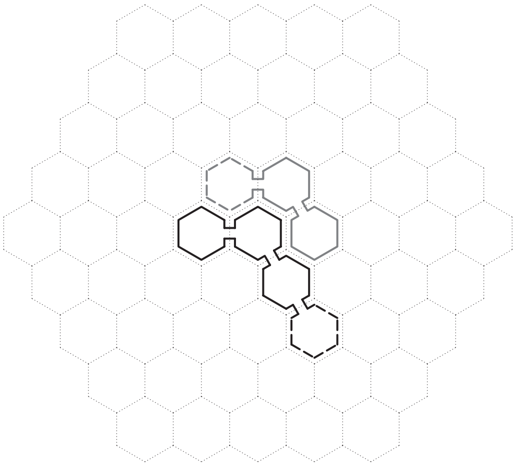


rule-hex-player1-block

rule-hex-player1-move

rule-hex-player1-move, -move-step2, -move-step3, -move-step4, -switch

10



rule-hex-player2-block

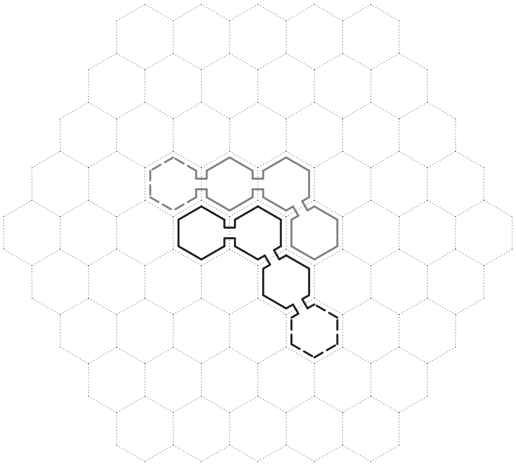
rule-hex-player2-move

rule-hex-player2-move, -move-step2, -move-step3, -move-step4, -switch

5.08 Derivation of the Hexagon Path grammar continued.

rule-hex-player2-move, -move-step2, -move-step3, -move-step4, -switch

11

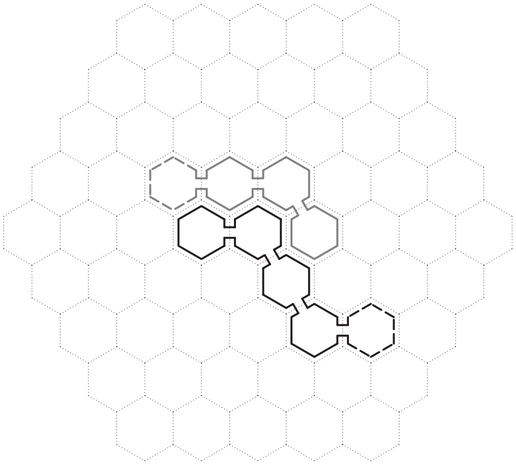


rule-hex-player1-block

rule-hex-player1-move

rule-hex-player1-move, -move-step2, -move-step3, -move-step4, -switch

12



rule-hex-player2-block

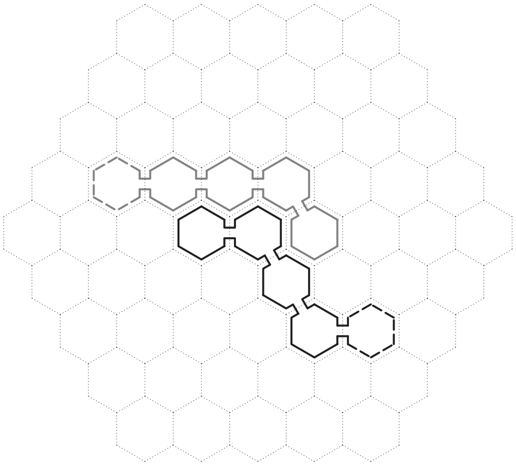
rule-hex-player2-move

rule-hex-player2-move, -move-step2, -move-step3, -move-step4, -switch

5.08 Derivation of the Hexagon Path grammar continued.

↓ rule-hex-player2-move, -move-step2, -move-step3, -move-step4, -switch

13

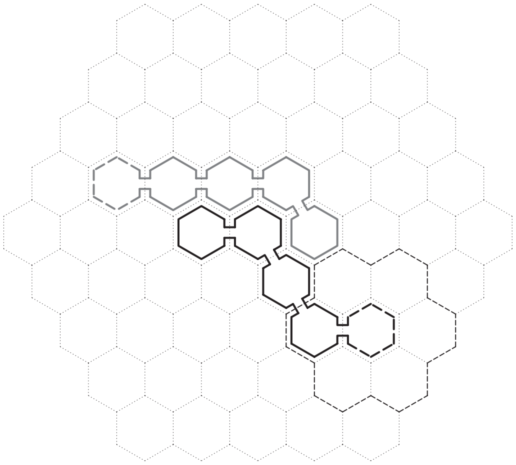


rule-hex-player1-block

rule-hex-player1-move

↓ rule-hex-player1-block

14



S: rule-hex-player2-block-step2

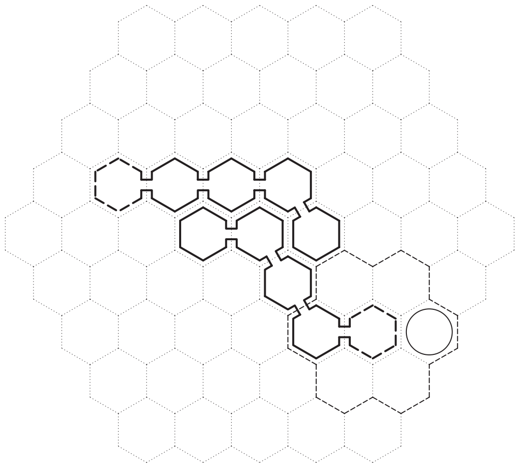
F: -

↓ rule-hex-player2-block-step2

5.08 Derivation of the Hexagon Path grammar continued.

rule-hex-player2-block-step2

15

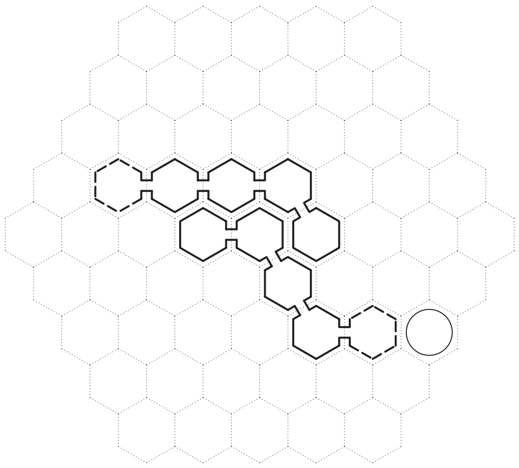


S: rule-hex-player1-switch

F: rule-hex-player1-move

rule-hex-player1-switch

16



rule-hex-player2-block

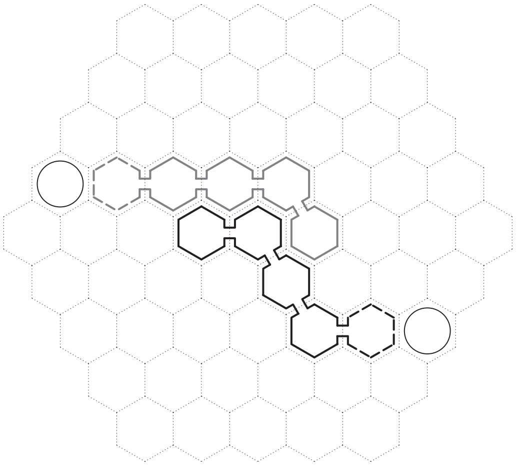
rule-hex-player2-move

rule-hex-player2-block, -block-step2, -switch

5.08 Derivation of the Hexagon Path grammar continued.

⇓ rule-hex-player2-block, -block-step2, -switch

17

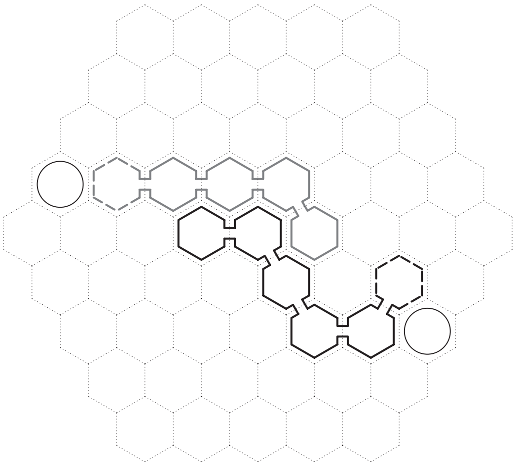


rule-hex-player1-block

rule-hex-player1-move

⇓ rule-hex-player1-move, -move-step2, -move-step3, -move-step4, -switch

18



rule-hex-player2-block

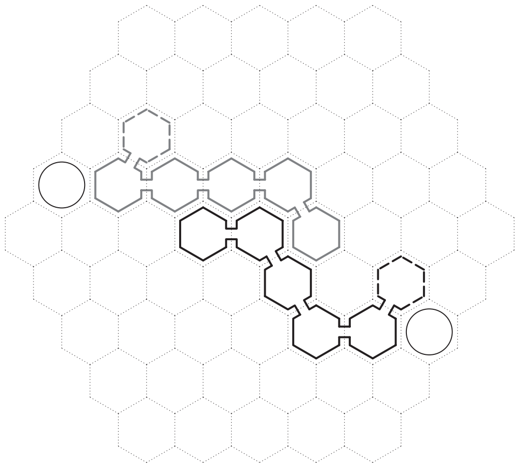
rule-hex-player2-move

⇓ rule-hex-player2-move, -move-step2, -move-step3, -move-step4, -switch

5.08 Derivation of the Hexagon Path grammar continued.

↓ rule-hex-player2-move, -move-step2, -move-step3, -move-step4, -switch

19

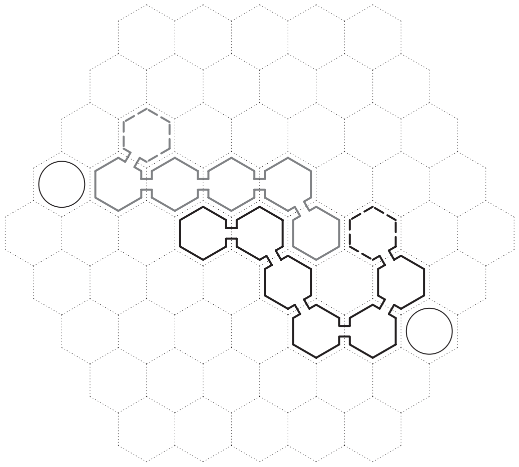


rule-hex-player1-block

rule-hex-player1-move

↓ rule-hex-player1-move, -move-step2, -move-step3, -move-step4, -switch

20



rule-hex-player2-block

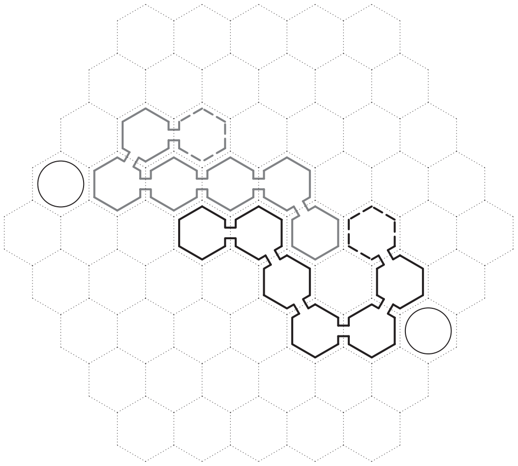
rule-hex-player2-move

↓ rule-hex-player2-move, -move-step2, -move-step3, -move-step4, -switch

5.08 Derivation of the Hexagon Path grammar continued.

↓ rule-hex-player2-move, -move-step2, -move-step3, -move-step4, -switch

21

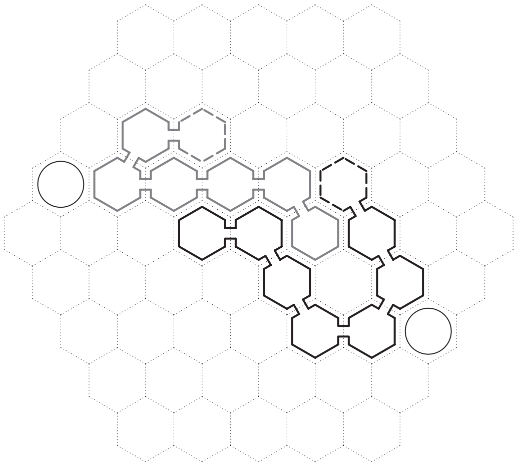


rule-hex-player1-block

rule-hex-player1-move

↓ rule-hex-player1-move, -move-step2, -move-step3, -move-step4, -switch

22



rule-hex-player2-block

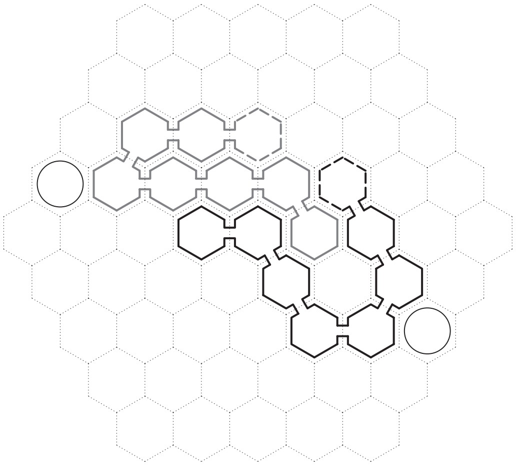
rule-hex-player2-move

↓ rule-hex-player2-move, -move-step2, -move-step3, -move-step4, -switch

5.08 Derivation of the Hexagon Path grammar continued.

rule-hex-player2-move, -move-step2, -move-step3, -move-step4, -switch

23

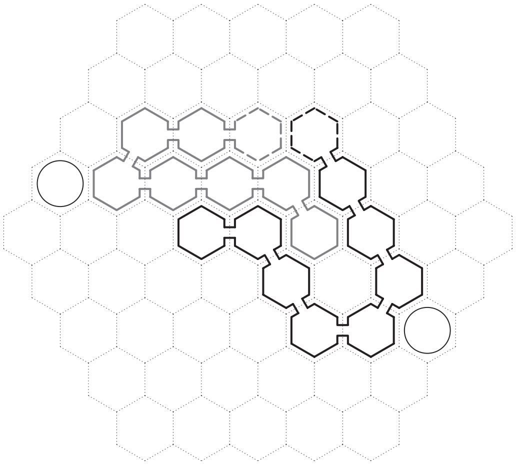


rule-hex-player1-block

rule-hex-player1-move

rule-hex-player1-move, -move-step2, -move-step3, -move-step4, -switch

24



rule-hex-player2-block

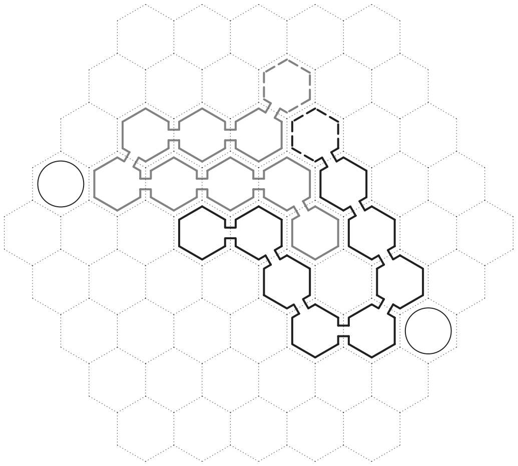
rule-hex-player2-move

rule-hex-player2-move, -move-step2, -move-step3, -move-step4, -switch

5.08 Derivation of the Hexagon Path grammar continued.

⇓ rule-hex-player2-move, -move-step2, -move-step3, -move-step4, -switch

25

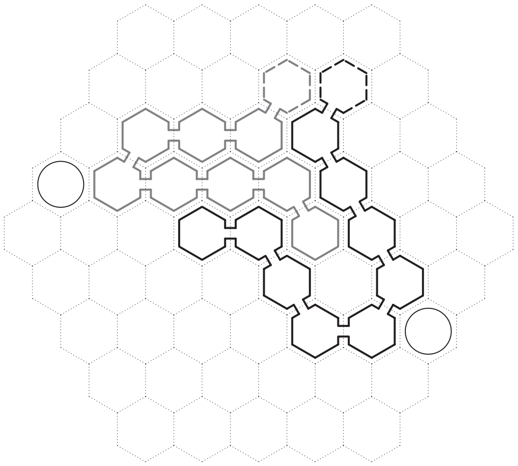


rule-hex-player1-block

rule-hex-player1-move

⇓ rule-hex-player1-move, -move-step2, -move-step3, -move-step4, -switch

26



rule-hex-player2-block

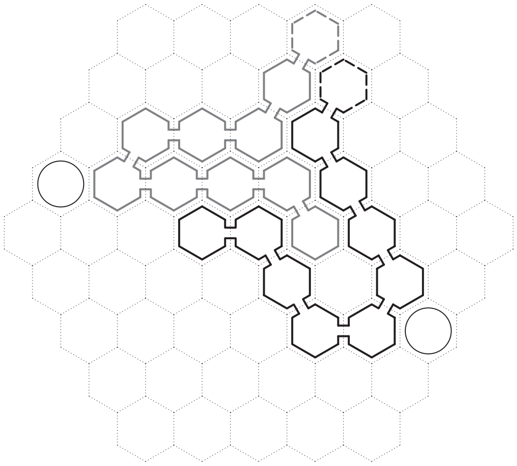
rule-hex-player2-move

⇓ rule-hex-player2-move, -move-step2, -move-step3, -move-step4, -switch

5.08 Derivation of the Hexagon Path grammar continued.

⇓ rule-hex-player2-move, -move-step2, -move-step3, -move-step4, -switch

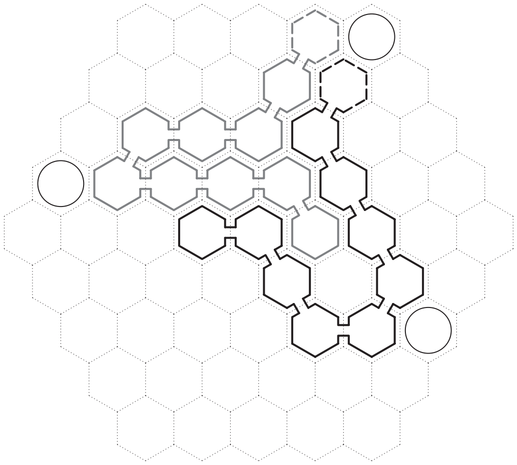
27



rule-hex-player1-block
rule-hex-player1-move

⇓ rule-hex-player1-block, -block-step2, -switch

28



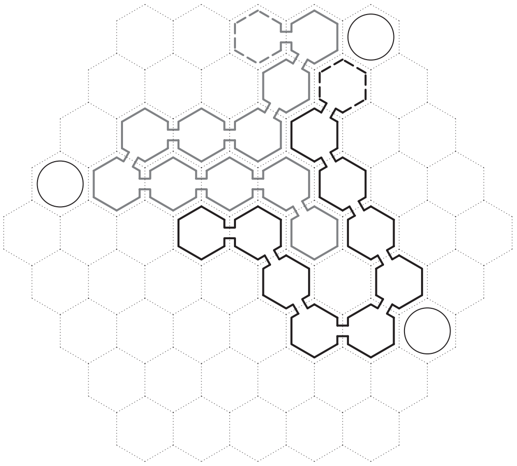
rule-hex-player2-block
rule-hex-player2-move

⇓ rule-hex-player2-move, -move-step2, -move-step3, -move-step4, -switch

5.08 Derivation of the Hexagon Path grammar continued.

rule-hex-player2-move, -move-step2, -move-step3, -move-step4, -switch

29

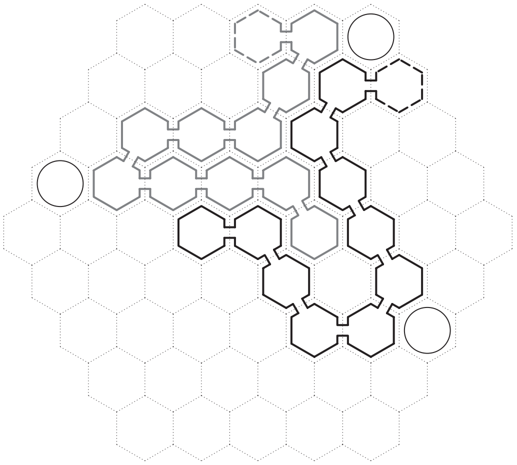


rule-hex-player1-block

rule-hex-player1-move

rule-hex-player1-move, -move-step2, -move-step3, -move-step4, -switch

30



rule-hex-player2-block

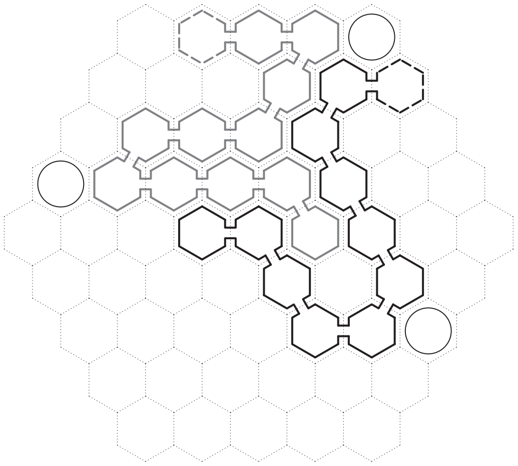
rule-hex-player2-move

rule-hex-player2-move, -move-step2, -move-step3, -move-step4, -switch

5.08 Derivation of the Hexagon Path grammar continued.

⇓ rule-hex-player2-move, -move-step2, -move-step3, -move-step4, -switch

31

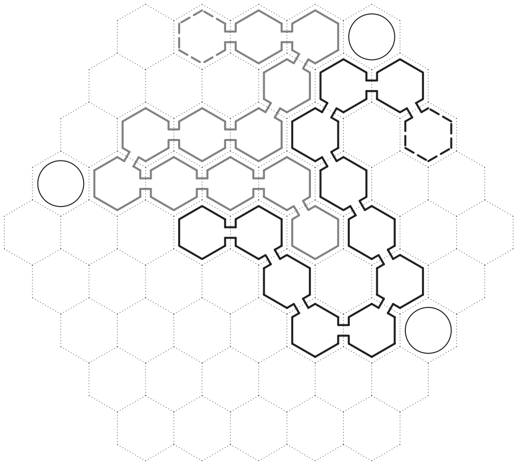


rule-hex-player1-block

rule-hex-player1-move

⇓ rule-hex-player1-move, -move-step2, -move-step3, -move-step4, -switch

32



rule-hex-player2-block

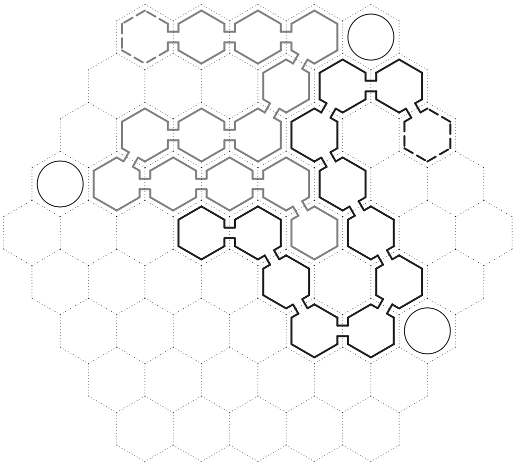
rule-hex-player2-move

⇓ rule-hex-player2-move, -move-step2, -move-step3, -move-step4, -switch

5.08 Derivation of the Hexagon Path grammar continued.

⇓ rule-hex-player2-move, -move-step2, -move-step3, -move-step4, -switch

33

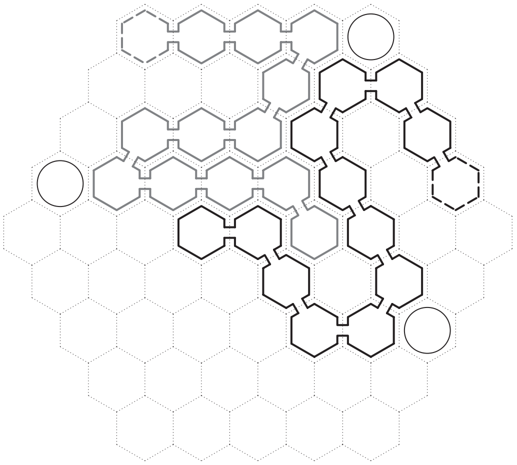


rule-hex-player1-block

rule-hex-player1-move

⇓ rule-hex-player1-move, -move-step2, -move-step3, -move-step4, -switch

34



rule-hex-player2-block

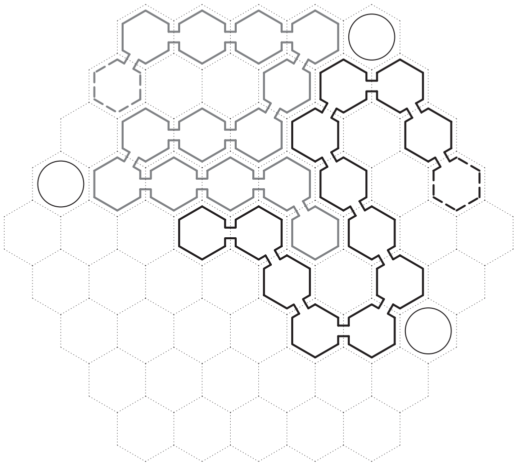
rule-hex-player2-move

⇓ rule-hex-player2-move, -move-step2, -move-step3, -move-step4, -switch

5.08 Derivation of the Hexagon Path grammar continued.

⇓ rule-hex-player2-move, -move-step2, -move-step3, -move-step4, -switch

35

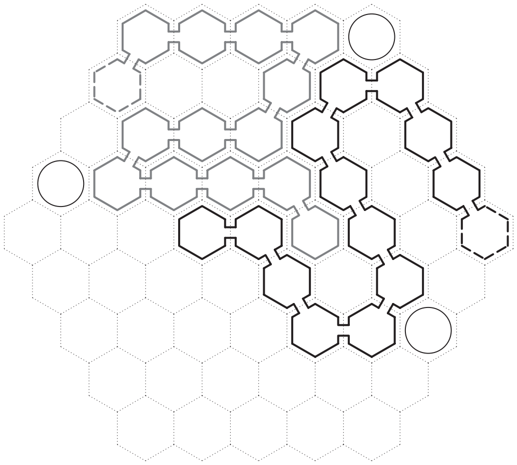


rule-hex-player1-block

rule-hex-player1-move

⇓ rule-hex-player1-move, -move-step2, -move-step3, -move-step4, -switch

36



rule-hex-player2-block

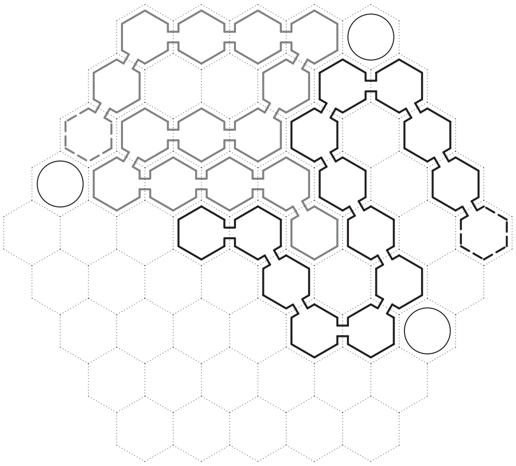
rule-hex-player2-move

⇓ rule-hex-player2-move, -move-step2, -move-step3, -move-step4, -switch

5.08 Derivation of the Hexagon Path grammar continued.

⇓ rule-hex-player2-move, -move-step2, -move-step3, -move-step4, -switch

37

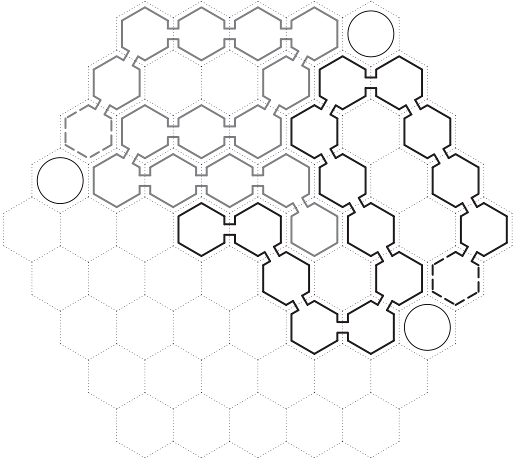


rule-hex-player1-block

rule-hex-player1-move

⇓ rule-hex-player1-move, -move-step2, -move-step3, -move-step4, -switch

38



rule-hex-player2-block

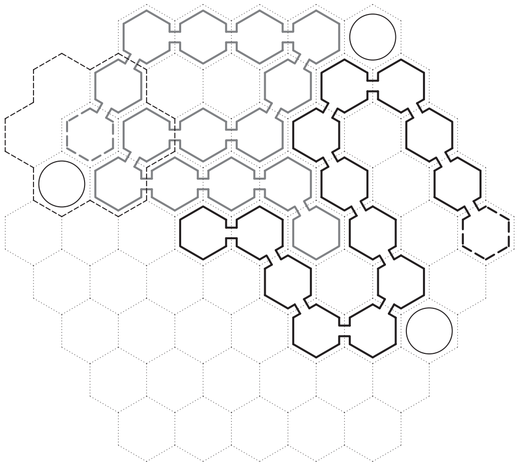
rule-hex-player2-move

⇓ rule-hex-player2-move

5.08 Derivation of the Hexagon Path grammar continued.

rule-hex-player2-move

39

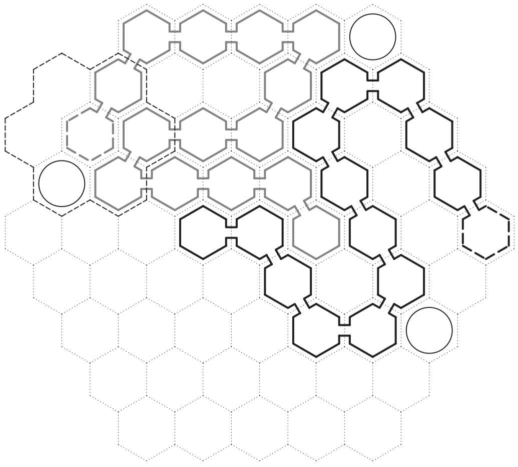


S: rule-hex-player1-move-step2

F: -

rule-hex-player2-move-step2

40

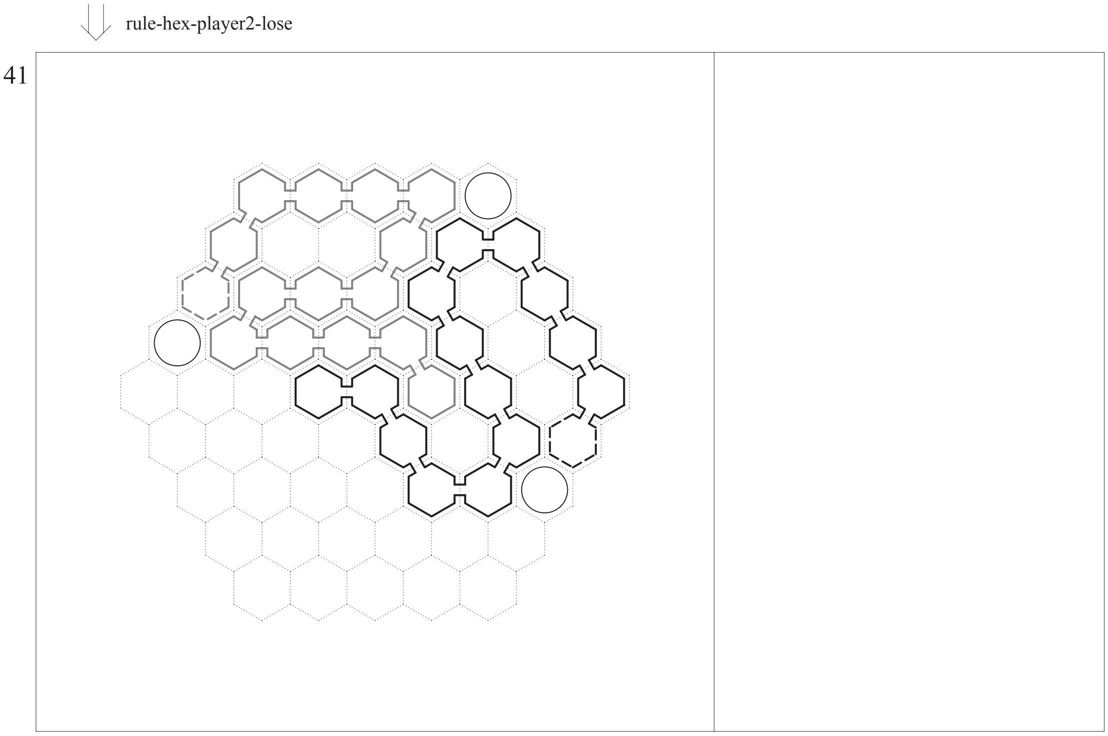


S: rule-hex-player2-switch

F: rule-hex-player2-lose

rule-hex-player2-lose

5.08 Derivation of the Hexagon Path grammar continued.



5.08 Derivation of the Hexagon Path grammar continued.

5.2 Rules of the Hexagon Path grammar

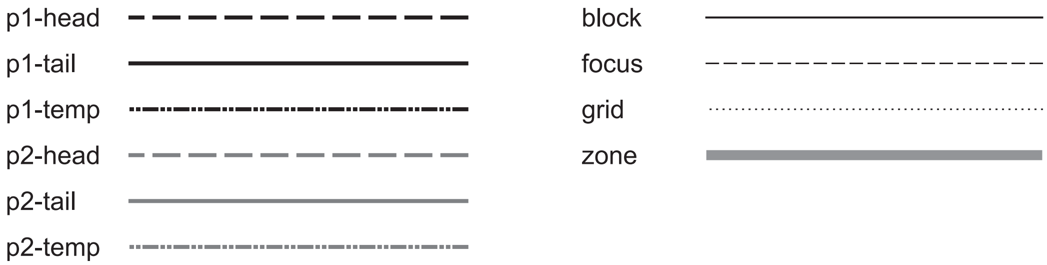
There are a total of 16 rules in the Hexagon Path grammar (figure 5.09). The rules are listed in this section in alphabetical order (figure 5.11). Each rule is divided into two parts. The left half of the frame gives the name of the rule along with the geometric representation of the schema. The pertinent parameters are noted by dimension lines. The right half of the frame lists the components of a rule. It begins with a general description of the rule and includes the parameters g() and the components for the meta-language descriptors. For all the rules, the transformation t() has been omitted since there are no transformation restrictions unless otherwise noted. The variable "unit" stores the length of one side of the hexagon pieces and is set at the beginning of the grammar.

rule-hex-player1-block
rule-hex-player1-block-step2
rule-hex-player1-lose
rule-hex-player1-move
rule-hex-player1-move-step2
rule-hex-player1-move-step3
rule-hex-player1-move-step4
rule-hex-player1-switch

rule-hex-player2-block
rule-hex-player2-block-step2
rule-hex-player2-lose
rule-hex-player2-move
rule-hex-player2-move-step2
rule-hex-player2-move-step3
rule-hex-player2-move-step4
rule-hex-player2-switch

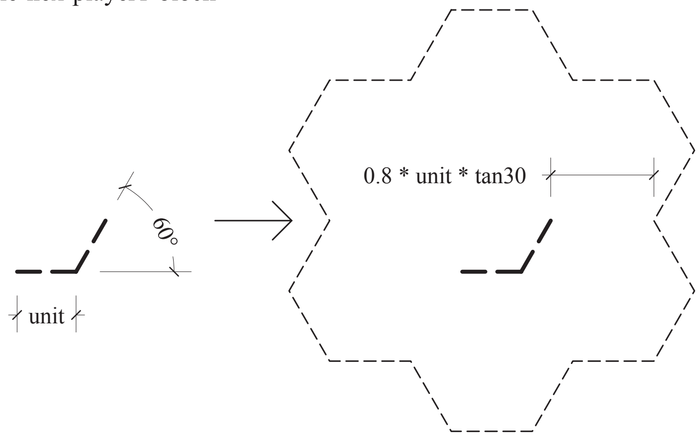
5.09 The 16 rules of the Hexagon Path grammar. Player1 and player2 have the same set of rules except they work on different labeled lines representing the two different players.

The linetypes used in the rules are the same as those used in the derivation of the Hexagon Path grammar (figure 5.06). The different linetypes correspond to the different labeled lines used in the grammar. The only change is the addition of a gray shaded area to represent the zone descriptor (figure 5.10).



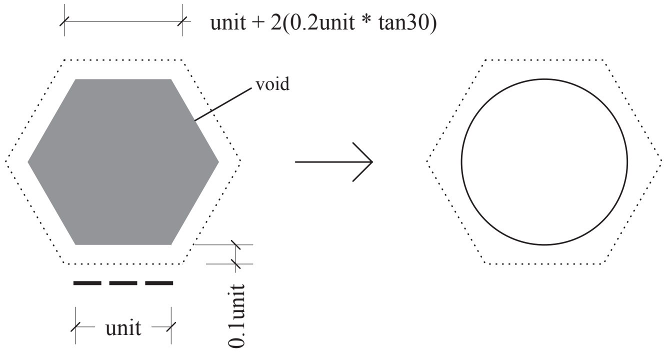
5.10 The linetype chart for the Hexagon Path grammar.

rule-hex-player1-block



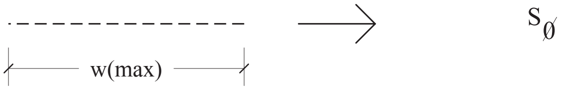
DESCRIPTION:	The primary rule of a macro that places a block for player1. this rule places a focus polygon around the head of the path.
CONSTRAINT:	nil
LABEL-FILTER:	on
APPLY-MODE:	random
SUCCESS:	rule-hex-player1-block-step2
FAILURE:	nil
RULE-SET:	nil

rule-hex-player1-block-step2



DESCRIPTION:	Rule that looks for an empty hexagon and fills it in with block piece.
CONSTRAINT:	nil
LABEL-FILTER:	on
APPLY-MODE:	single
SUCCESS:	rule-hex-player1-switch
FAILURE:	rule-hex-player1-lose
RULE-SET:	nil

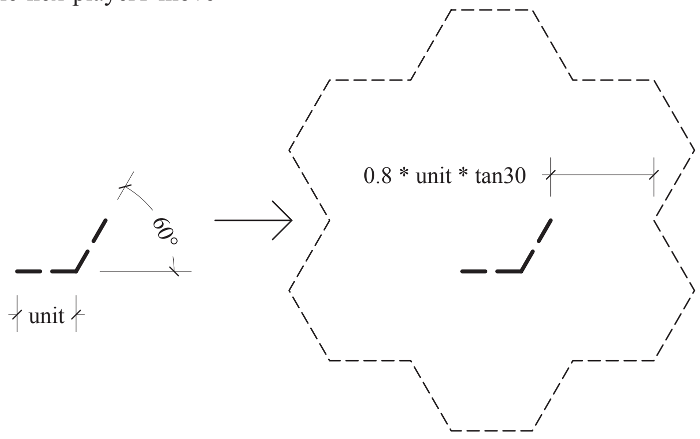
rule-hex-player1-lose



DESCRIPTION:	Rule that erases all focus lines and ends the game for player1 by removing all the rules from the ruleset.
CONSTRAINT:	w > 0
LABEL-FILTER:	on
APPLY-MODE:	parallel
SUCCESS:	nil
FAILURE:	nil
RULE-SET:	(sub-rule: rule-hex-player1-move rule-hex-player1-block)

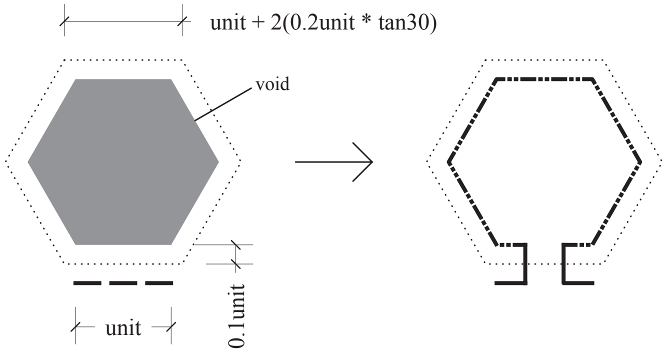
5.11 Rules for the Hexagon Path grammar.

rule-hex-player1-move



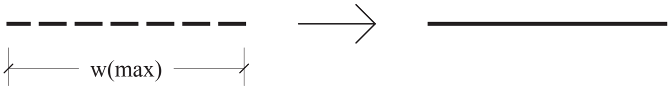
DESCRIPTION:	The primary rule of a macro that places a piece for player1. this rule places a focus polygon around the head of the path.
CONSTRAINT:	nil
LABEL-FILTER:	on
APPLY-MODE:	random
SUCCESS:	rule-hex-player1-move-step2
FAILURE:	nil
RULE-SET:	nil

rule-hex-player1-move-step2



DESCRIPTION:	Rule that looks for an empty hexagon and fills it in with player1's piece.
CONSTRAINT:	nil
LABEL-FILTER:	on
APPLY-MODE:	single
SUCCESS:	rule-hex-player1-move-step3
FAILURE:	rule-hex-player1-lose
RULE-SET:	nil

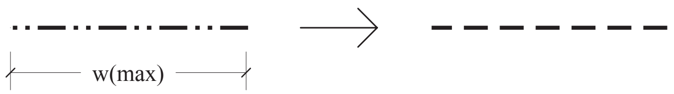
rule-hex-player1-move-step3



DESCRIPTION:	Rule that converts all p1-head lines to p1-tail lines.
CONSTRAINT:	nil
LABEL-FILTER:	on
APPLY-MODE:	parallel
SUCCESS:	rule-hex-player1-move-step4
FAILURE:	rule-hex-player1-move-step4
RULE-SET:	nil

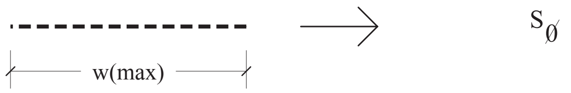
5.11 Rules for the Hexagon Path grammar continued.

rule-hex-player1-move-step4



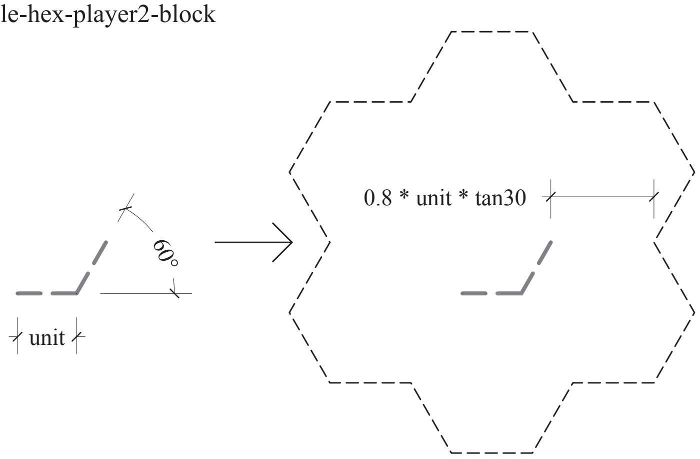
DESCRIPTION:	Rule that converts all p1-temp lines to p1-head lines.
CONSTRAINT:	nil
LABEL-FILTER:	on
APPLY-MODE:	parallel
SUCCESS:	rule-hex-player1-switch
FAILURE:	rule-hex-player1-switch
RULE-SET:	nil

rule-hex-player1-switch



DESCRIPTION:	Rule that erases all focus lines and modifies the rule-set so that player2 moves next.
CONSTRAINT:	w > 0
LABEL-FILTER:	on
APPLY-MODE:	parallel
SUCCESS:	nil
FAILURE:	nil
RULE-SET:	(set-rule: rule-hex-player2-move rule-hex-player2-block)

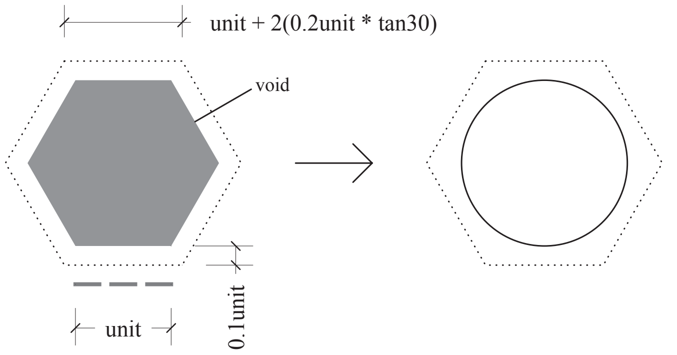
rule-hex-player2-block



DESCRIPTION:	The primary rule of a macro that places a block for player2. this rule places a focus polygo n around the head of the path.
CONSTRAINT:	nil
LABEL-FILTER:	on
APPLY-MODE:	random
SUCCESS:	rule-hex-player2-block-step2
FAILURE:	nil
RULE-SET:	nil

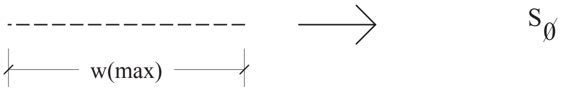
5.11 Rules for the Hexagon Path grammar continued.

rule-hex-player2-block-step2



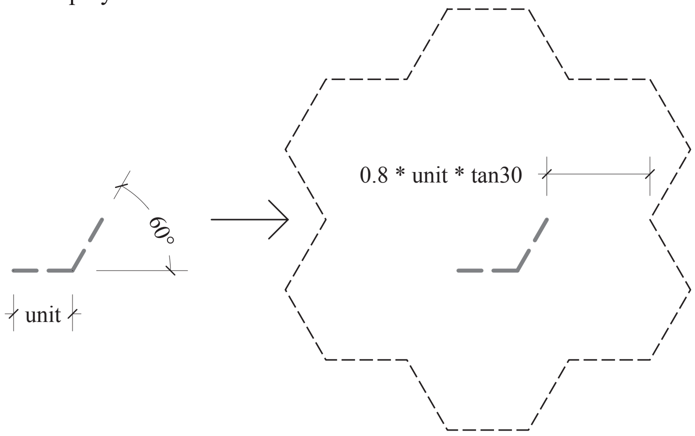
DESCRIPTION:	Rule that looks for an empty hexagon and fills it in with block piece.
CONSTRAINT:	nil
LABEL-FILTER:	on
APPLY-MODE:	single
SUCCESS:	rule-hex-player2-switch
FAILURE:	rule-hex-player2-lose
RULE-SET:	nil

rule-hex-player2-lose



DESCRIPTION:	Rule that erases all focus lines and ends the game for player2 by removing all the rules from the ruleset.
CONSTRAINT:	w > 0
LABEL-FILTER:	on
APPLY-MODE:	parallel
SUCCESS:	nil
FAILURE:	nil
RULE-SET:	(sub-rule: rule-hex-player2-move rule-hex-player2-block)

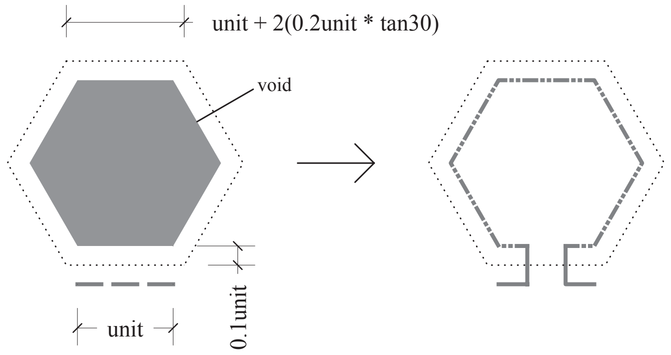
rule-hex-player2-move



DESCRIPTION:	The primary rule of a macro that places a piece for player2. this rule places a focus polygon around the head of the path.
CONSTRAINT:	nil
LABEL-FILTER:	on
APPLY-MODE:	random
SUCCESS:	rule-hex-player2-move-step2
FAILURE:	nil
RULE-SET:	nil

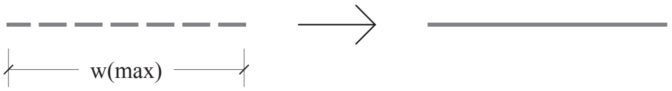
5.11 Rules for the Hexagon Path grammar continued.

rule-hex-player2-move-step2



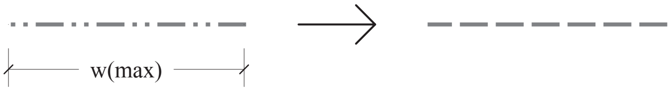
DESCRIPTION:	Rule that looks for an empty hexagon and fills it in with player2's piece.
CONSTRAINT:	nil
LABEL-FILTER:	on
APPLY-MODE:	single
SUCCESS:	rule-hex-player2-move-step3
FAILURE:	rule-hex-player2-lose
RULE-SET:	nil

rule-hex-player2-move-step3



DESCRIPTION:	Rule that converts all p2-head lines to p2-tail lines.
CONSTRAINT:	nil
LABEL-FILTER:	on
APPLY-MODE:	parallel
SUCCESS:	rule-hex-player2-move-step4
FAILURE:	rule-hex-player2-move-step4
RULE-SET:	nil

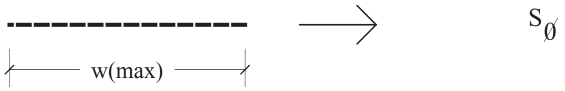
rule-hex-player2-move-step4



DESCRIPTION:	Rule that converts all p2-temp lines to p2-head lines.
CONSTRAINT:	nil
LABEL-FILTER:	on
APPLY-MODE:	parallel
SUCCESS:	rule-hex-player1-switch
FAILURE:	rule-hex-player1-switch
RULE-SET:	nil

5.11 Rules for the Hexagon Path grammar continued.

rule-hex-player2-switch



DESCRIPTION:	Rule that erases all focus lines and modifies the rule-set so that player1 moves next.
CONSTRAINT:	w > 0
LABEL-FILTER:	on
APPLY-MODE:	parallel
SUCCESS:	nil
FAILURE:	nil
RULE-SET:	(set-rule: rule-hex-player1-move rule-hex-player1-block)

5.11 Rules for the Hexagon Path grammar continued.

6.0 Durand grammar

The Durand grammar demonstrates how the descriptors in the meta-language can be used to emulate a process of design to generate architectural floor plans. The grammar is based on *Précis of the Lectures on Architecture* written in the early 1800's by Jean Nicolas Louis Durand and was the most significant treatise on architecture of the first half of the nineteenth century (Kruft 1994). The book instructed engineering students at the École Polytechnique in France about the basics of architecture by systematically describing how to generate architectural designs.

Durand divides architectural education into four basic areas: building materials, building elements, building parts, and building composition. He begins by describing the basic materials of architecture such as stone, brick, tile, and wood. These materials are then used to manufacture architectural elements such as walls, columns, and floors. Using these elements, he describes a process for assembling them together to produce building parts such as porches, vestibules, stairs, rooms and courtyards. In the end, he shows how different building parts can be combined together to generate an entire building composition.

There are two processes at work here. One process is bottom-up which builds up the design from the basic elements. The other process is top-down where the entire composition of the building is arranged first and then the architectural elements are used to fill in the composition (Villari 1990). In regards to these two processes, Durand says,

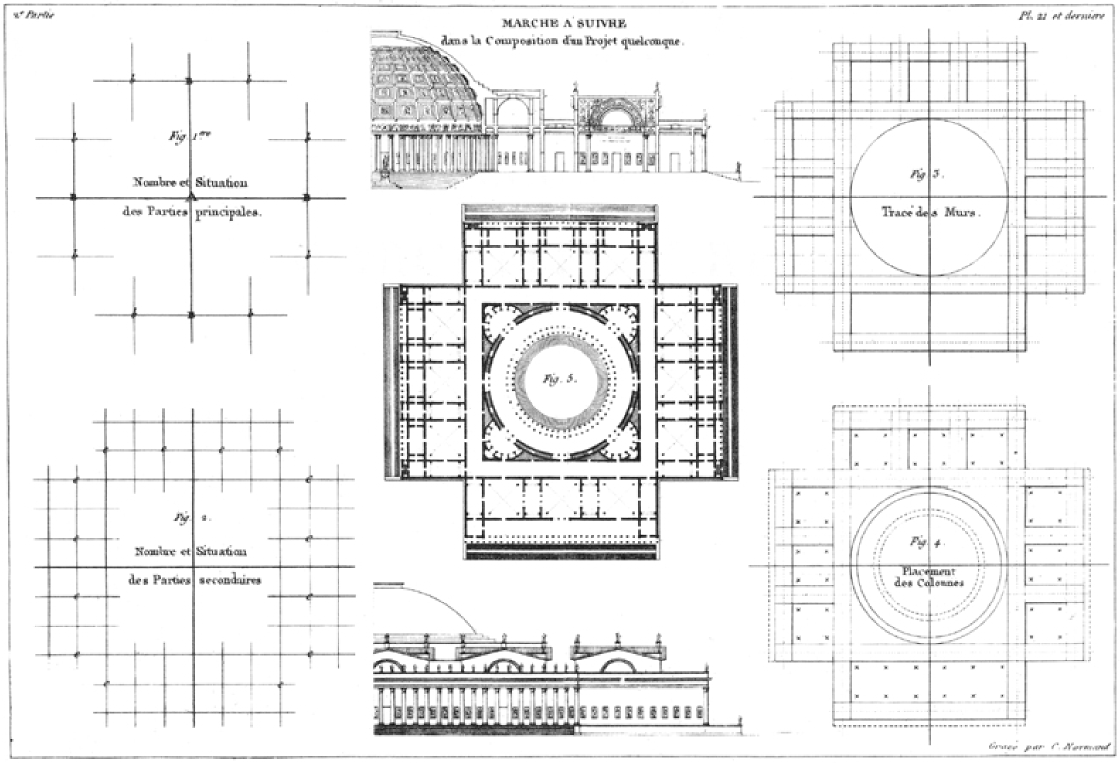
> "To combine the different elements, then to proceed to the different parts of buildings, and from those parts to the whole: such is the natural sequence that should be observed in learning to compose. By contrast, when you come to compose for yourself, you must begin with the whole, proceed to the parts, and finish with the details." (page 180)

The Durand grammar will focus only on the top-down design process described in developing the composition of a building. The goal of the grammar is to demonstrate how the meta-language descriptors can be used to describe the design process outlined in *Précis of the Lectures on Architecture.*

The grammar can produce a subset of the designs shown in his book. The user is also free to produce designs that are not part of Durand's vocabulary. This is mainly due to the fact that there are no symmetrical restrictions in the grammar. Durand states that one of the responsibilities of an architect is to design an economical building and a symmetrical building is the most economical design. Instead of incorporating symmetry into the rules, I have given the user the option of designing an asymmetrical and therefore non-economical building. This freedom gives the user the ability to design more modernistic plans using Durand's process of design. The grammar is meant to show how the process of design can be encoded into a grammar in the form of rules to generate a variety of designs.

6.1 Durand’s process of design

One of the key drawings that illustrate Durand’s design process is shown in figure 6.01. The French caption for the drawing is entitled “Procedure to be followed in the composition of any project”. The process begins with the layout of principal axes (top left). Then secondary axes are placed (bottom left). Walls are then placed in relation to the axes (top right). The columns are then placed in relation to the walls (bottom right). And finally other architectural elements, such as doors, windows, and stairs, are added to finish the building (center).



6.01 Illustration showing Durand’s design process. The French caption for the drawing is entitled “Procedure to be followed in the composition of any project”. The process begins with the layout of principal axes (top left). Then secondary axes are placed (bottom left). Walls are then placed in relation to the axes (top right). The columns are then placed in relation to the walls (bottom right). And finally other architectural elements, such as doors, windows, and stairs, are added to finish the building (center).

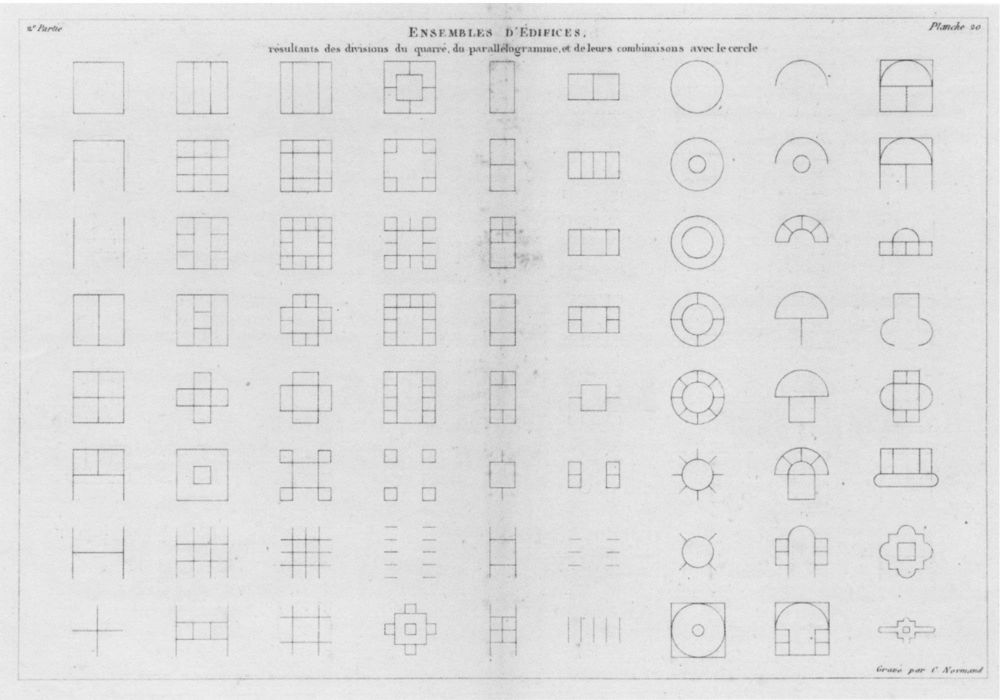
This method of design is a top-down process where the larger elements are recursively refined to generate the details. The process described by Durand so far is ambiguous. He describes the gist of the process but leaves out many of the details. How are the axes placed? Is there an organizing principle to determine their placement? What is the relationship between the parti and the walls? What is the relationship between the columns and the walls? How should the rooms be arranged? What conditions determine where a door is placed? These questions are answered to some extent in his writings about his design process. Durand writes:

> "Once we have set out parallel and equidistant axes, and drawn other axes at identical intervals to intersect them at right angles, the walls are set down on the axes, as many interaxes apart as is considered appropriate, and the columns, pilaster and so on, are placed on the intersections of those same axes; then the interaxes are bisected to create new axes that serve to define the positions of the doors, windows, arcades, and so on." (page 120)

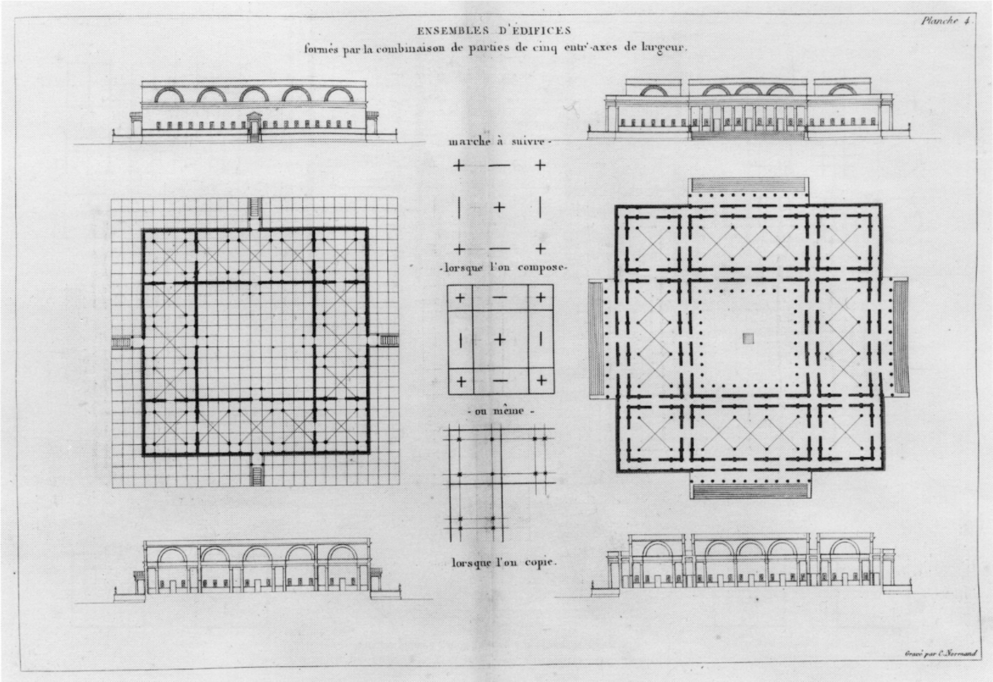
What Durand describes as parallel and equidistant axes at identical intervals is a grid. The grid is the key starting point for Durand's process. All the architectural elements are placed in relation to the grid. The walls are placed on center with the grid. The columns are placed at intersections of the grid. The position of doors and windows are placed on axes that bisect the grid. This standardization of design was the precursor to prefabricated construction although Durand himself did not envisage this possibility (Kruft 1994).

The first step in Durand's process is to place the primary and secondary axes such that they bisect the spacing of the grid. The placement of the axes lines are typically done in a symmetrical fashion. Durand includes a table of possible axis line placements as a guide (figure 6.02). The next step is to develop a parti based on the axis and the underlying grid. The size of the parti is dependant upon the designer. Throughout his book, Durand gives examples of appropriate parti arrangements. The center column of drawings of figure 6.03 shows one example. The two larger floor plans demonstrate how the same parti can generate different floor plans.

After the placement of the parti, the walls are defined. The parti is used as the centerline for the placement of the walls. From here, other architectural elements can be inserted into drawing. Columns are placed at the intersection of the grid in relation to the rooms generated from the walls. Doors and windows are placed on the walls that intersect the primary and secondary axes. Stairs and platforms are placed in relation to the walls as well as the grid and axes but are not specifically mentioned in his writings.


2e Partie
Ensembles d'Édifices,
résultants des divisions du quarré, du parallélogramme, et de leurs combinaisons avec le cercle
Planche 20
Gravé par C. Normand

6.02 Examples of axis line placements from Durand. The heading in French translates to "Building composition: Results of the division of the square, the parallelogram, and their combination with the circle."



6.03 An example of how one parti arrangement can be used to create two different floor plans. The heading in French translates to "Building composition: Formed by the combination of parts that are five interaxes in width."

Durand's process of design uses the grid and axes to create a control structure that determines how the architectural elements will be placed. This method of first creating a control structure to be the guiding principle for the rest of the design is a commonly used technique to generate architectural design (Lawson 1997, Crowe 1995).

Architects throughout the ages have used this method for design. In the 15th century, Leon Battista Alberti describes a similar concept when he wrote of using lineamenta to determine the placement of walls and columns in a building (Rykwert, Leach, and Tavernor 1988). According to Tzonis and Lefaivre (1986), classical architecture organized designs around a taxis which is a hierarchical grid that determined the placement of architectural elements. Louis Sullivan, an American architect in the 19th Century, used axes lines and energy points to organically generate ornamental designs on his buildings (Twombly and Menocal 2000). Modernist architect Le Corbusier used regulating lines to develop designs with certain proportions, like the Golden Section (Rowe 1976).

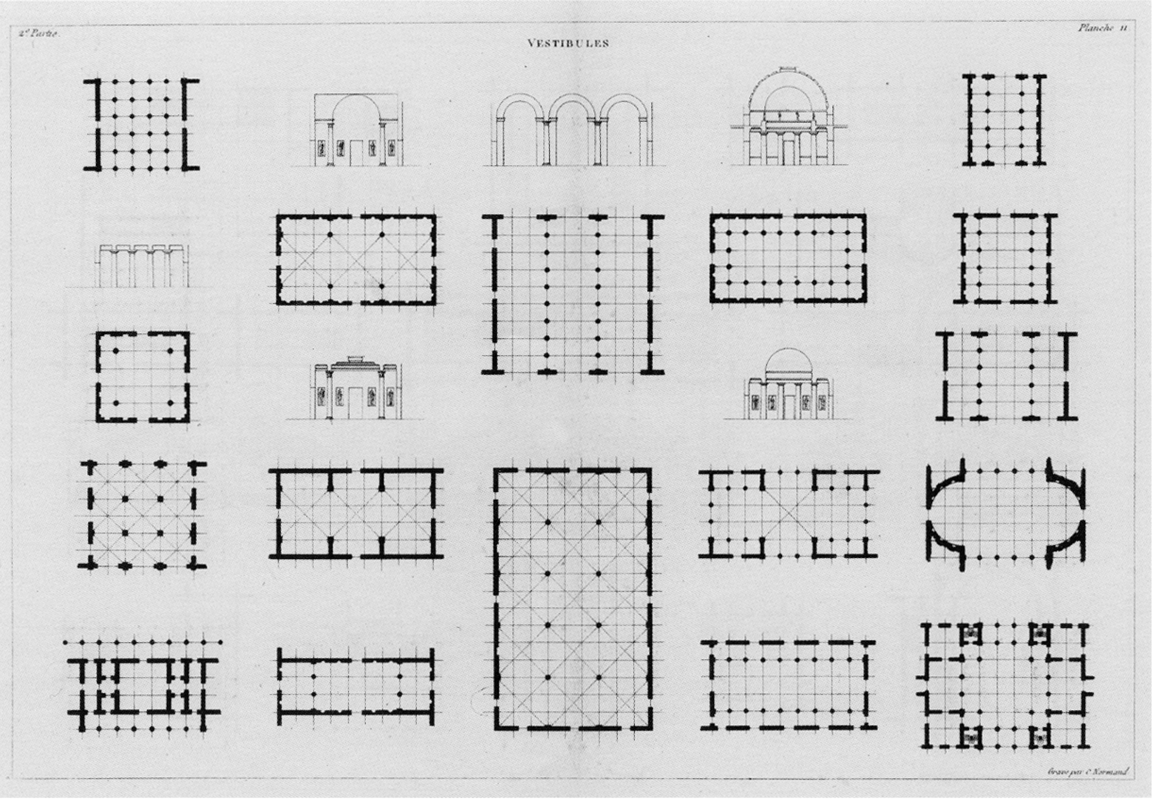
Durand's process naturally divides into a series of distinct steps. The first step is to create the axes. The second step is to generate the parti. The third step is to create the walls and so forth. These steps can be translated into stages to produce the Durand grammar.

6.2 Stages of the Durand grammar

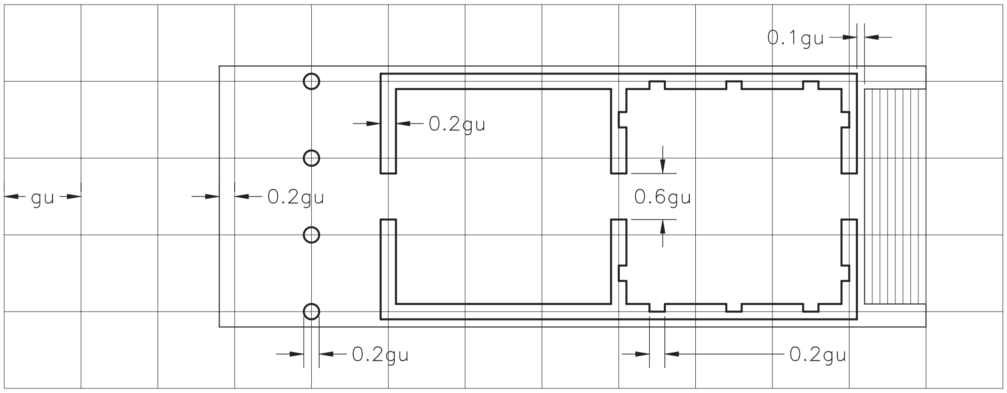
The Durand grammar presented here produces only a portion of the floor plans in the *Précis of the Lectures on Architecture.* Specifically it produces only floors plans that are rectilinear with walls, columns, niches, openings, base, and exterior stairs. Other elements such as interior stairs, semi-circular walls, buttressing, and vaulted roofs have been omitted. But since the grammar is designed as a collection of stages, the rules for producing the omitted elements can be inserted into the grammar as additional stages at a later time.

In *Précis of the Lectures on Architecture,* Durand specifically mentions how to arrange axis, parti, walls, columns and openings such as doors and windows (figure 6.04). Other elements such as stairs, niches and the base are not directly mentioned. But by examining the floor plans presented in the book, one can interpolate that there are also rules for the placement of exterior and interior stairs, the making of niches and the generation of a base.

The size of the architectural elements in the Durand grammar is fixed as specific ratios to the unit size of the grid (figure 6.05). For example, the thickness of the wall is 20% of the grid unit. The openings in the walls are 60% of the grid unit. The width of the niches in the walls is 20% of the grid unit. These ratios are fixed into the rules. The grammar also uses a variety of labeled lines to distinguish between the different elements of the design. For example, axis lines are represented by lines in the V_{12} algebra with the label axis. The same goes for grids, walls, stairs, etc… Each element has its own label.

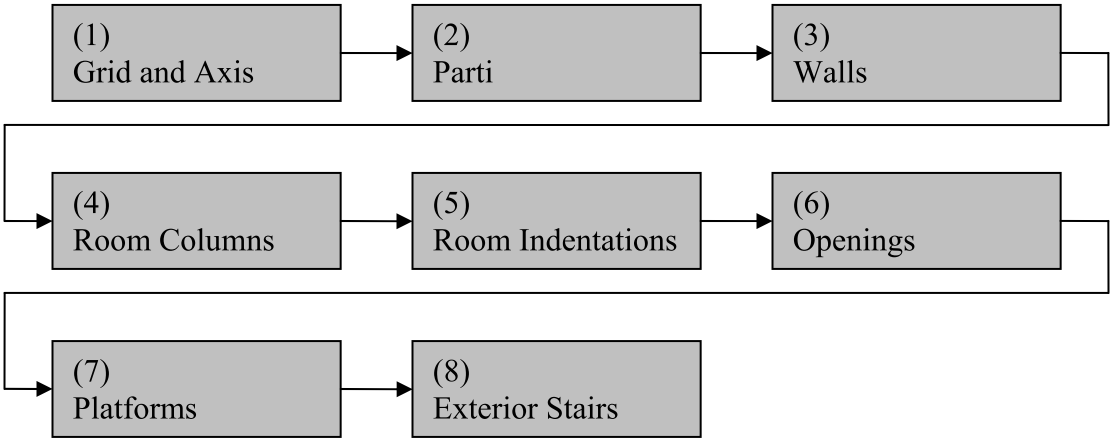


6.04 Examples of relationships between the elements of a building and the underlying grid; the walls are always placed on center to the grid lines; the columns are always placed at the intersection of a grid; and the openings are made between grid lines, never at an intersection. In this particular plate, the designs are for vestibules.



6.05 A sample design showing the relationship between the architectural elements and the size of the grid in the Durand grammar. The grid size is represented by the variable “gu”. All elements in the design are placed in reference to the grid size. For instance, the diameter of the columns is 20% of the variable gu, the size of the openings are 60% of gu, and the minimum distance between a platform and the exterior staircase is 10% of gu.

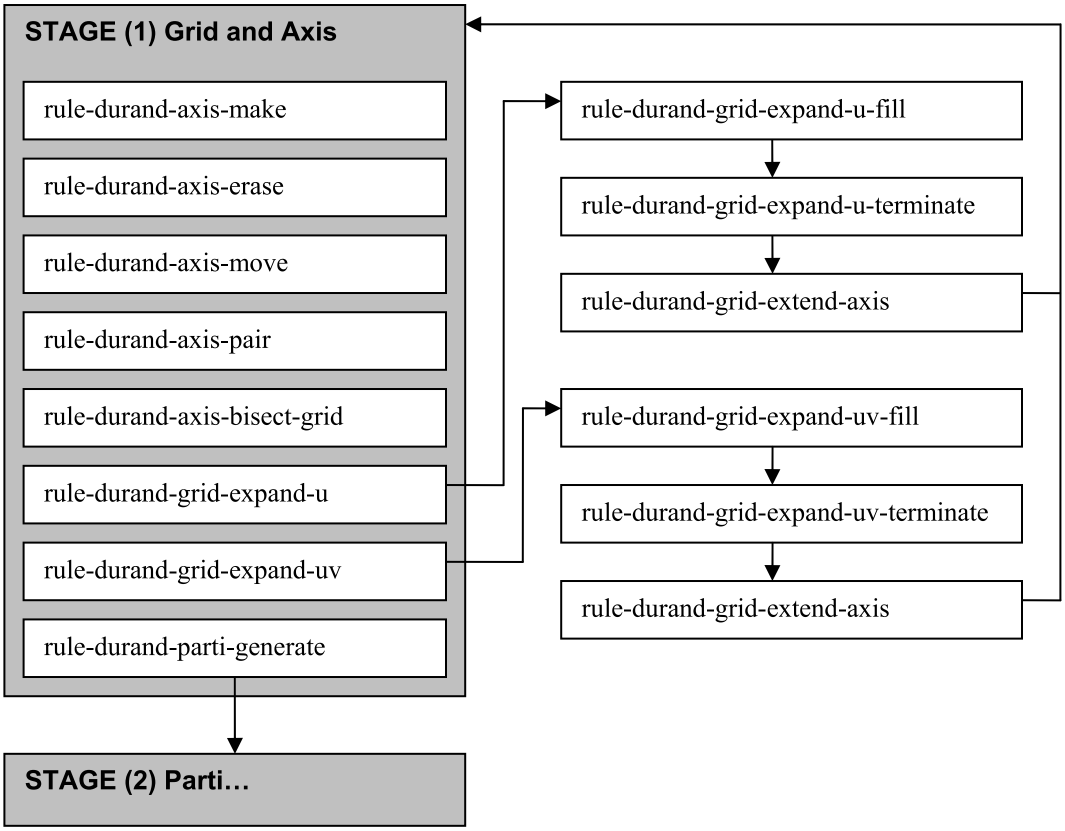
The initial shape for the Durand grammar is a 9x9 square grid. The grammar is divided into eight major stages of design: grid and axis, parti, walls, room columns, room indentations, openings, platforms, and exterior stairs (figure 6.06). (In the grammar, the niches in the walls are called indentations and the base of the building is called a platform). In this section, I will provide a synopsis of the overall effect of each stage on the design. For a detailed understanding of the rules, please see section 6.5 where the rules are listed in alphabetical order.



6.06 The eight stages of the Durand grammar.

In the first stage of the Durand grammar, the user defines the initial grid size and creates a few axes (figure 6.07). This stage is composed of eight rules where two of the rules are primary rules of macros. There are rules to make, erase, and move axis lines. The user can also place an axis line that bisects the entire grid. There are three restrictions for placing an axis lines. First, the axis lines must always bisect the grid squares, as described by Durand. Second, no axis line can be placed in the edge square of the grid. And third, the axis lines must always extend to the edge of the grid. The rules in this stage are designed to maintain these conditions. Although Durand makes a distinction between primary, secondary and tertiary axis lines, I have made the decision to have only one type of axis line to simplify the grammar.

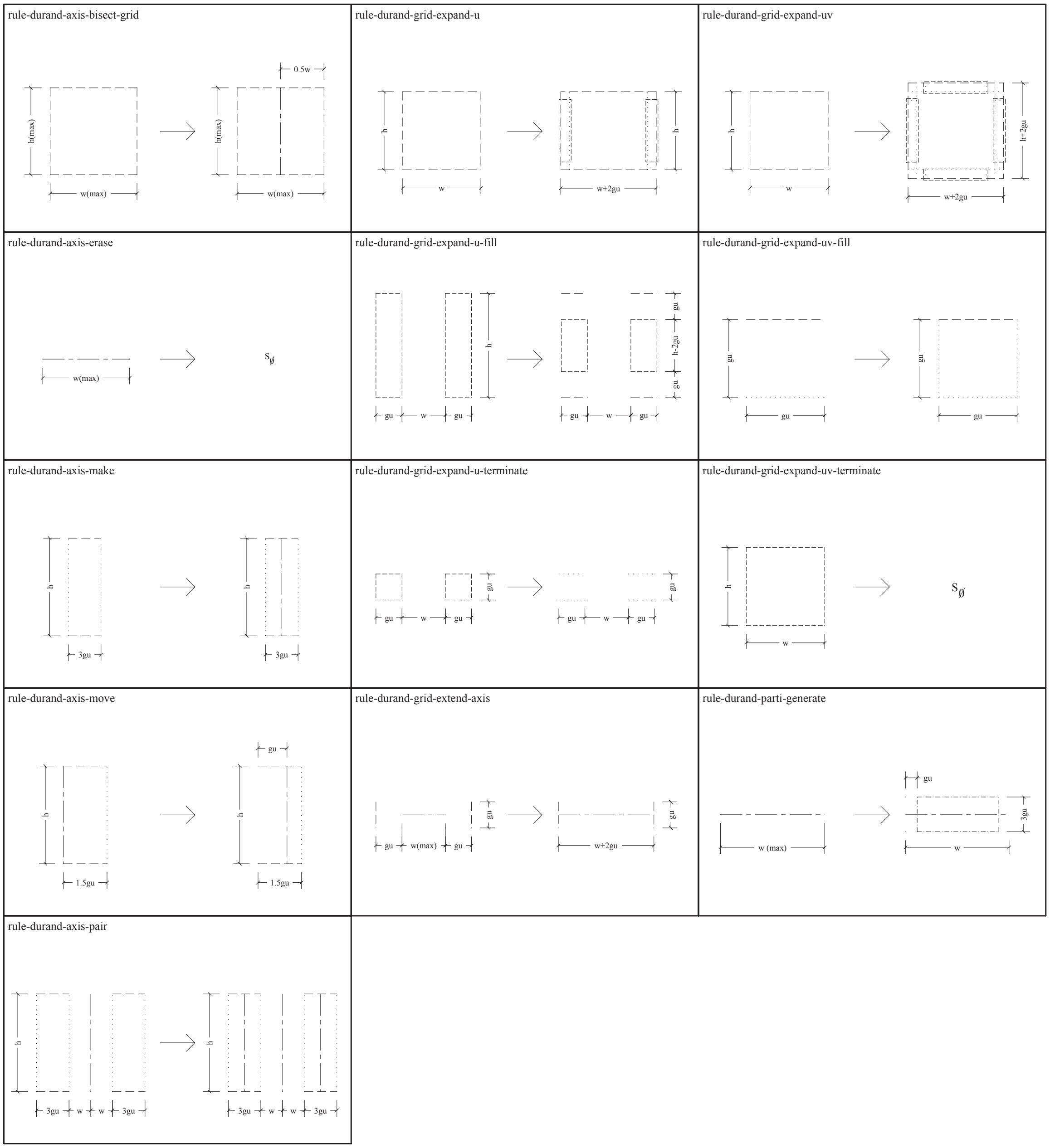
The user can also expand the overall size of the grid horizontally (in the u direction) as well as vertically (in the v direction) using the two macros in this stage. There are two types of macros: procedural and serial. A procedural macro can be divided into three stages. The first stage is to isolate the area that needs to be changed typically using the focus descriptor. The second stage is to perform the necessary changes within the isolated area. The third stage is to un-isolate the area, by removing the focus polygon, and performing any other necessary changes to the drawing to finish the design transformation. A serial macro, on the other hand, is one continuous linear sequence of rules that performs one design transformation.


STAGE (1) Grid and Axis
rule-durand-axis-make
rule-durand-axis-erase
rule-durand-axis-move
rule-durand-axis-pair
rule-durand-axis-bisect-grid
rule-durand-grid-expand-u
rule-durand-grid-expand-uv
rule-durand-parti-generate
rule-durand-grid-expand-u-fill
rule-durand-grid-expand-u-terminate
rule-durand-grid-extend-axis
rule-durand-grid-expand-uv-fill
rule-durand-grid-expand-uv-terminate
rule-durand-grid-extend-axis
STAGE (2) Parti…

6.07 Diagram of the Grid and Axis stage of the Durand grammar. Each stage is represented by a gray box and each white box represents a rule. The arrows show the success rule of the directive descriptor. The failure rules are assumed to return to the current stage unless otherwise noted.

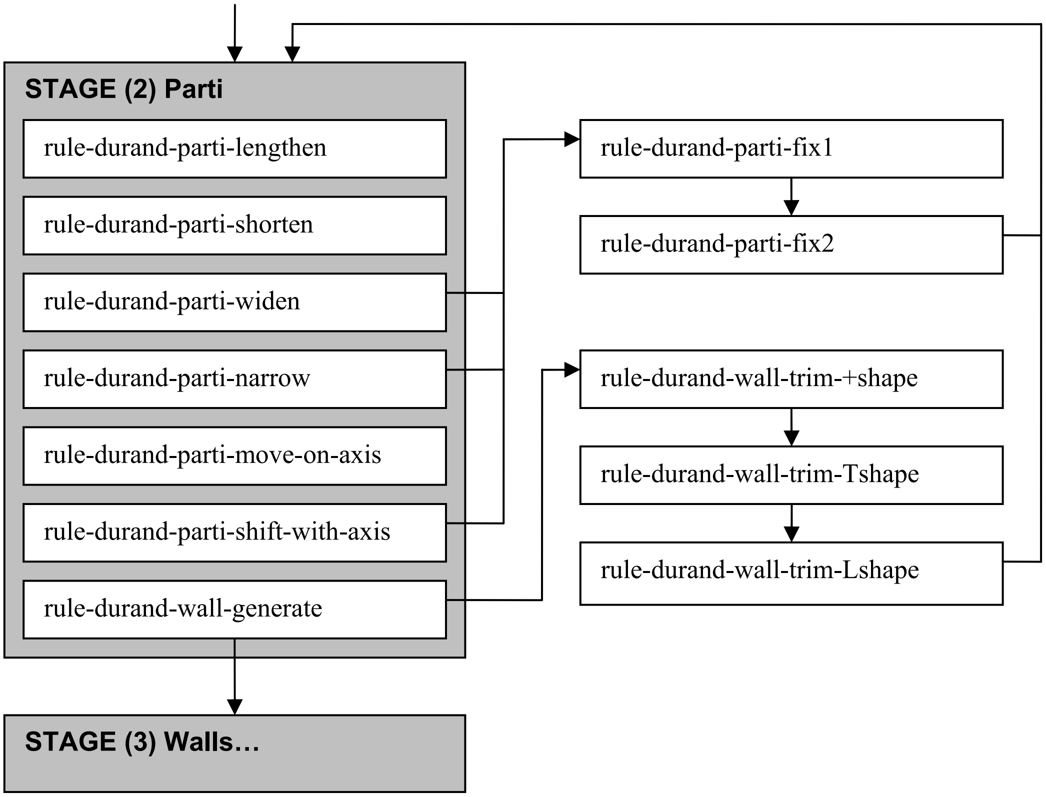
The macro *rule-durand-parti-expand-uv* is a procedural macro and is composed of four rules. The macro begins by expanding the grid by one grid unit in both the u and v direction using *rule-durand-grid-expand-uv*. This rule also places four focus rectangles around the newly created space which isolates the working area. The next step is to fill in all the remaining space with grid lines with *rule-durand-parti-expand-uv-fill*. Once the grid lines are placed, all the focus rectangles are removed using *rule-durand-expand-uv-terminate*. The final step uses *rule-durand-grid-extend-axis* to extend any axial lines that do not touch the edge of the grid.

Rule *rule-durand-parti-generate* creates the parti from the axis and also changes the rule-set definition to go to the next stage. Durand is very free with his use of the parti. The only concrete statement he makes is that the parti should be bisected by an axial line and should be in alignment with the other parti lines that are adjoining. In order to systematize the generation of the parti, I made some assumptions from my analysis of his drawings. The partis are generally long rectangular boxes that run along the length of the axis. The minimum width of the box is typically three grid units. My method to generate the parti is to place a rectangular box that is three units wide and is two grid units short of the maximal axis line it is being placed along. These rectangular parti boxes are applied in parallel to all axis lines in the drawing. An abridged version of the rules in the Grid and Axis stage is shown in figure 6.08.


rule-durand-axis-bisect-grid
h(max)
w(max)
0.5w
rule-durand-grid-expand-u
h
w
w+2gu
rule-durand-grid-expand-uv
h+2gu
rule-durand-axis-erase
w(max)
S∅
rule-durand-grid-expand-u-fill
gu
h-2gu
rule-durand-grid-expand-uv-fill
rule-durand-axis-make
3gu
rule-durand-grid-expand-u-terminate
rule-durand-grid-expand-uv-terminate
rule-durand-axis-move
1.5gu
rule-durand-grid-extend-axis
rule-durand-parti-generate
w (max)
3gu
rule-durand-axis-pair

6.08 The abridged version of the rules used in the Grid and Axes stage of the Durand grammar. The rules modify the size of the grid and place axis lines. For a detailed view of each rule see section 6.5.

The second stage of the grammar allows the user to modify the default parti generated from stage one (figure 6.09). This stage is composed of seven rules where four of the rules are macros. The user can lengthen or shorten a rectangular parti along an axis line, widen or narrow a rectangular parti, shift the parti up or down the axis line, or move the parti and the axis line together perpendicular to the direction of the axis. The placement of the parti lines can be anywhere on the grid except along the edge of the grid.

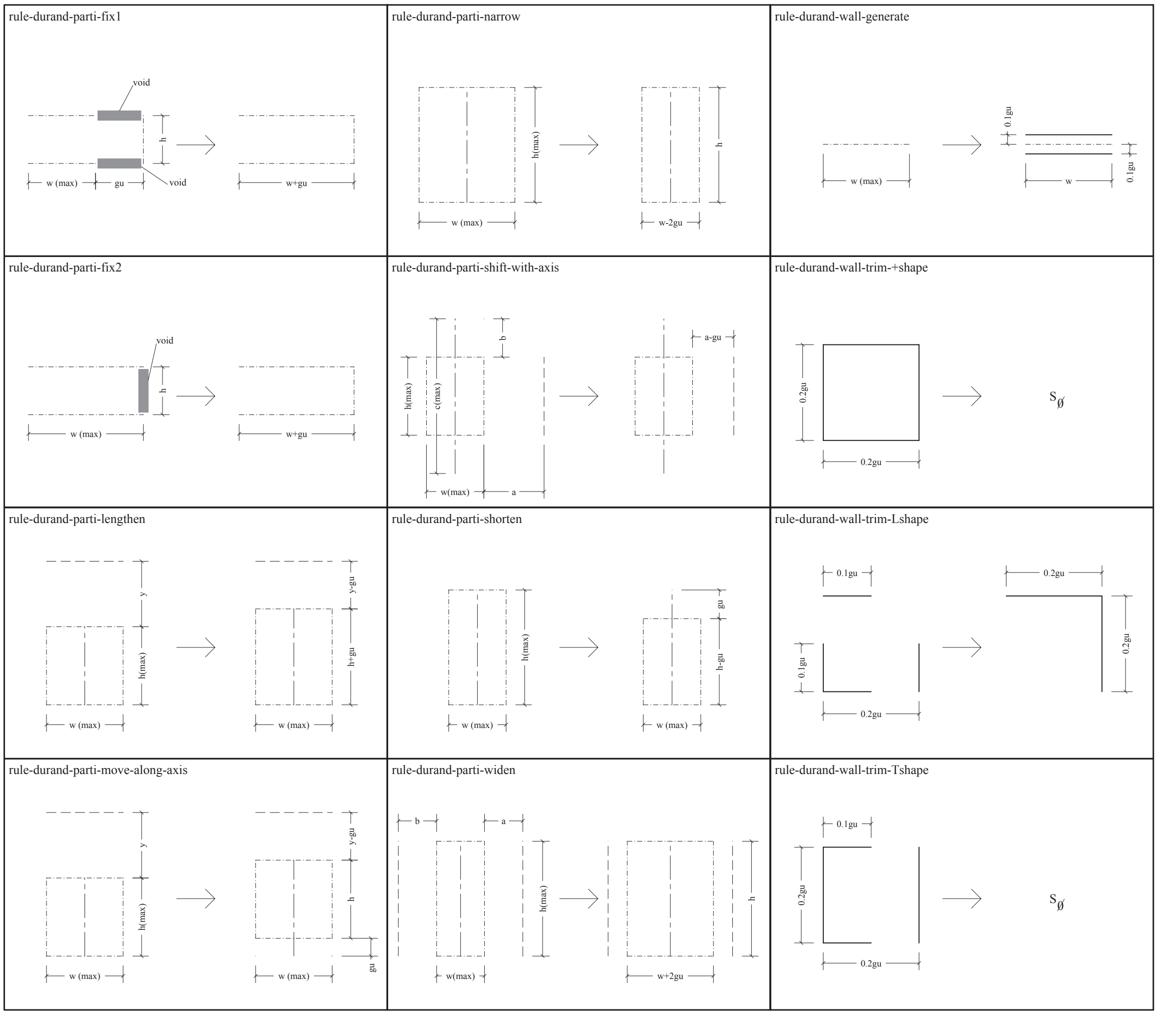

STAGE (2) Parti
rule-durand-parti-lengthen
rule-durand-parti-shorten
rule-durand-parti-widen
rule-durand-parti-narrow
rule-durand-parti-move-on-axis
rule-durand-parti-shift-with-axis
rule-durand-wall-generate
rule-durand-parti-fix1
rule-durand-parti-fix2
rule-durand-wall-trim-+shape
rule-durand-wall-trim-Tshape
rule-durand-wall-trim-Lshape
STAGE (3) Walls…

6.09 The Parti stage of the Durand grammar. The rules in this stage modify the rectangular parti generated from the first stage.

The macros that widen, narrow, and shift the parti with the axis, use the rules *rule-durand-parti-fix1* and *rule-durand-parti-fix2* to fix potential gaps that can occur when manipulating the rectangular parti. These gaps occur when two parti rectangles share a common parti line. When one parti rectangle is moved, a segment of the other parti rectangle is removed. The secondary rules of the macro are designed to replace those missing parti lines.

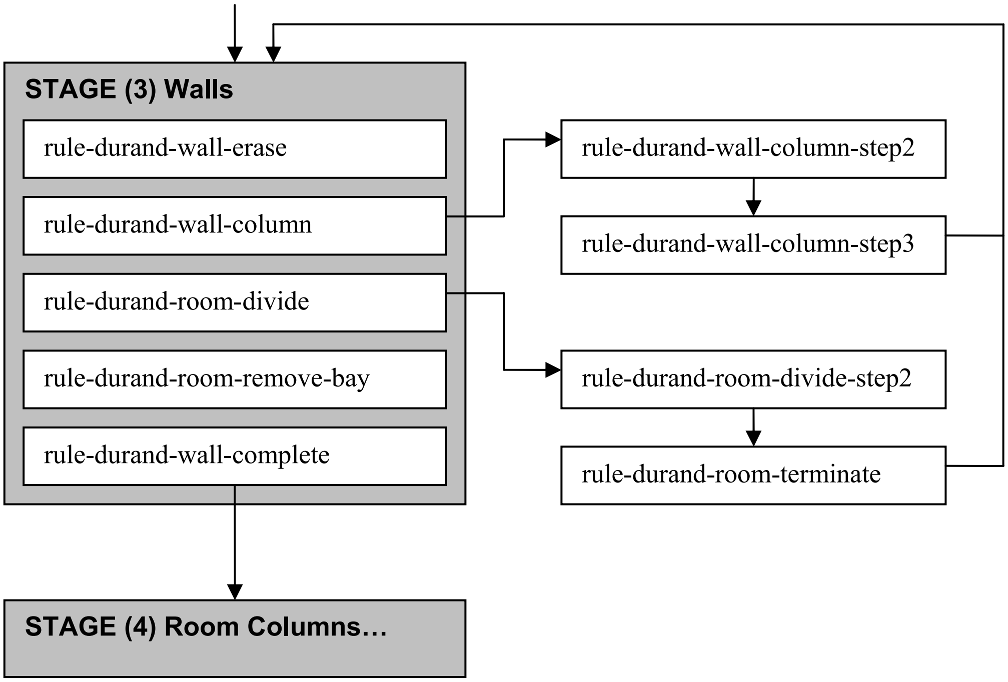
The parti and axis lines act as the underlying control structure for the building. All other architectural elements are placed in relation to the axis and parti lines. Once the user has a desirable parti, the last step in this stage is to create walls based on the parti lines. This is achieved by using the macro *rule-durand-wall-generate* which is a serial macro. Unlike the macro *rule-durand-grid-expand-uv,*which is a procedural macro, a serial macro links a series of rules together in order to execute a design transformation that requires several steps in sequence.

The *rule-durand-wall-generate* macro is identical to the macro in figure 3.50. The macro begins by offsetting, from the parti lines, wall lines that are half the thickness of the wall. This creates a series of parallel wall lines that intersect each other. The next step is to trim off and fillet the intersections between the walls. There are three possible intersection conditions: cross intersections, T-intersections, and L-intersections. The macro checks for each type of intersection in that order. Depending on the parti arrangement, a design may not have all three conditions. If the intersection type does not exist in the drawing, the next intersection type will still be checked because both the success and failure rules are the same for each secondary rule in the macro. An abridged version of the rules in the Parti stage is shown in figure 6.10.



6.10 The abridged version of the rules for the Parti stage of the Durand grammar. The rules in this stage modify the rectangular parti generated from the first stage.

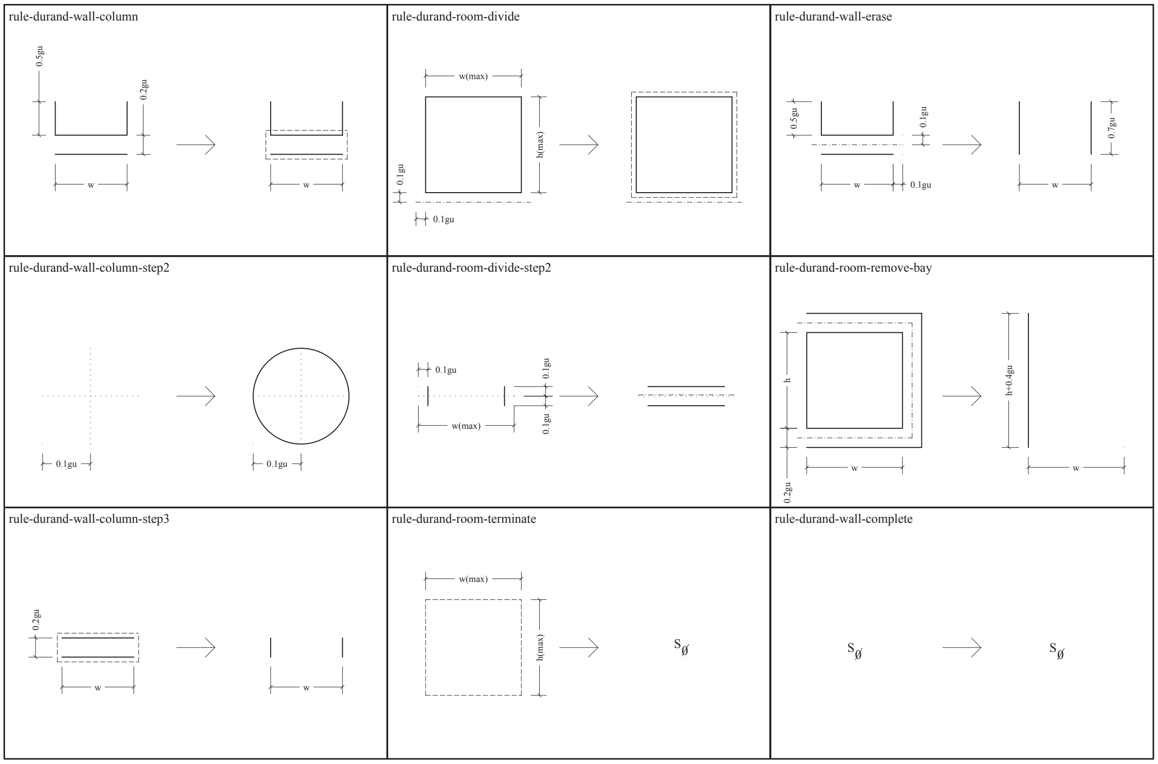
The third stage of the grammar is the Walls stage (figure 6.11). This stage is composed of five rules where two of the rules are macros. The user can erase walls, add walls by dividing a room, turn a wall into a series of columns, or remove a protruding bay. When a wall is erased, the parti lines on axis with the wall are also erased. This keeps a one-to-one correspondence between the underlying structure and the architectural elements of the building. When a wall is converted into a series of columns, the parti line remains because the columns still define an edge albeit a more porous one.


STAGE (3) Walls
rule-durand-wall-erase
rule-durand-wall-column
rule-durand-room-divide
rule-durand-room-remove-bay
rule-durand-wall-complete
rule-durand-wall-column-step2
rule-durand-wall-column-step3
rule-durand-room-divide-step2
rule-durand-room-terminate
STAGE (4) Room Columns…

6.11 The Walls stage of the Durand grammar. The rules in this stage modify the walls generated from the parti stage. The user can add a wall, erase a wall, or convert a wall into a series of columns.

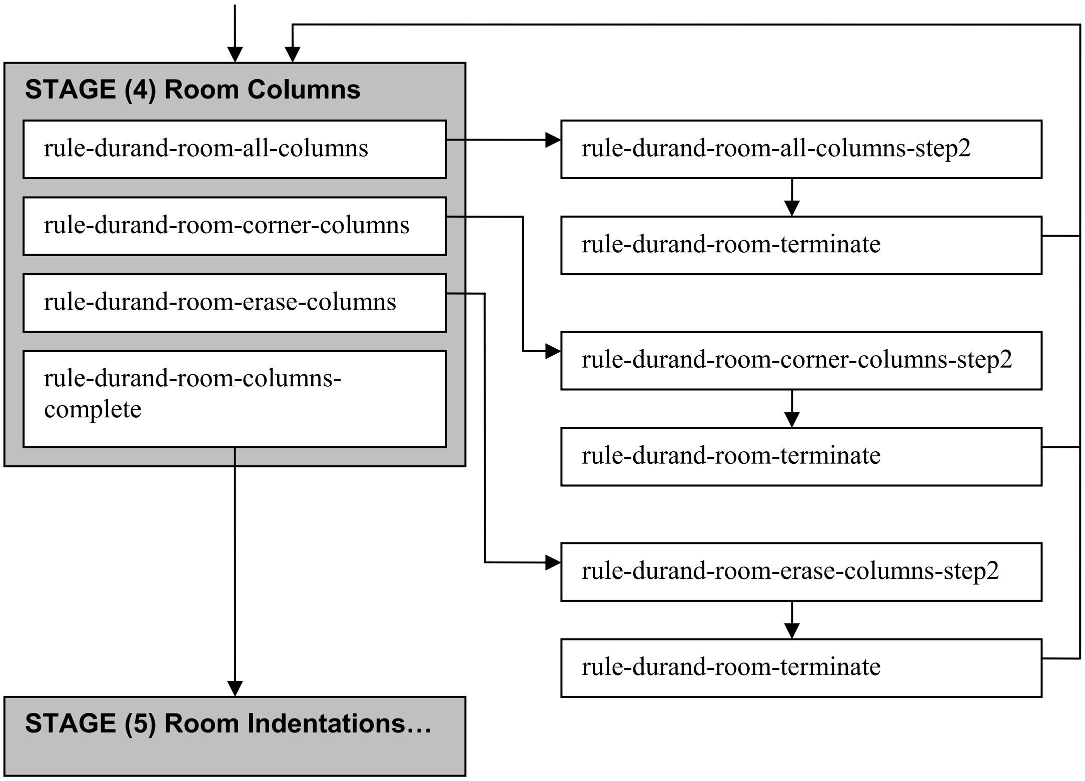
To convert a wall into a series of columns requires a procedural macro. The macro *rule-durand-wall-column* is the same as rules 3.37-3.39 in section 3.5.2. The first step is to isolate the wall that needs to be changed. This is done by placing a focus polygon around the wall. The next step is to place a column at every grid intersection with in the focus area. The last step is to remove the focus polygon and redraw the wall lines so that the old wall is erased and the end intersections are capped.

The macro for dividing a room is also a procedural macro. The rule *rule-durand-room-divide* looks for a four sided room and places a focus rectangle around the room. The next step in the macro is *rule-durand-room-divide-step2* which allows the user to select where inside the room to place a new wall. In addition to placing a new wall, a new parti line is also inserted. The final step is to remove the focus rectangle so that subsequent rules can apply to the entire drawing. The next stage of the grammar provides rules to modify rooms. An abridged version of the rules in the Walls stage is shown in figure 6.12.



6.12 The abridged version of the rules for the Walls stage of the Durand grammar. The rules in this stage modify the walls generated from the parti stage. The user can add a wall, erase a wall, or convert a wall into a series of columns.

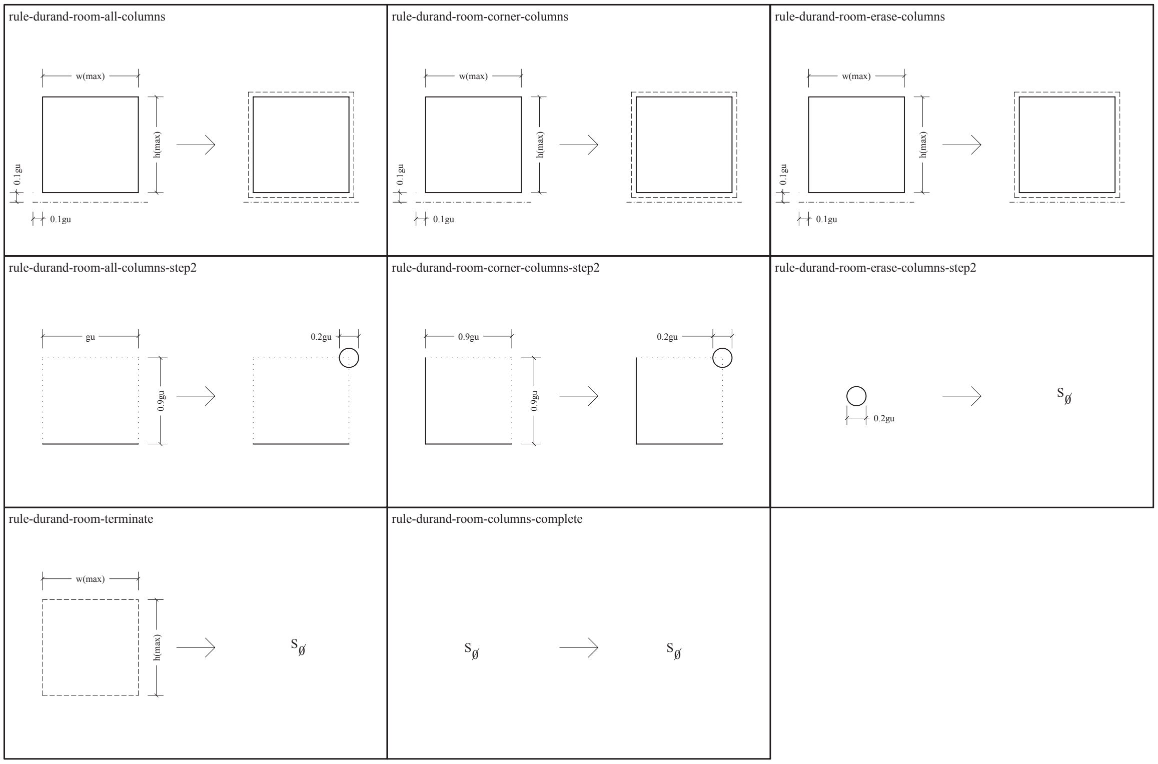
The fourth stage of the grammar is the Room Columns stage (figure 6.13). This stage is composed of five rules where three of the rules are macros. In this stage, the user can add columns all around the edge of a room or only at the corners of the room. If the user makes a mistake there is a rule to erase all the columns in a room.


STAGE (4) Room Columns
rule-durand-room-all-columns
rule-durand-room-corner-columns
rule-durand-room-erase-columns
rule-durand-room-columns-complete
rule-durand-room-all-columns-step2
rule-durand-room-terminate
rule-durand-room-corner-columns-step2
rule-durand-room-terminate
rule-durand-room-erase-columns-step2
rule-durand-room-terminate
STAGE (5) Room Indentations…

6.13 The Room Columns stage of the Durand grammar. In this stage, the user has the option to add columns to selected rooms in the design.

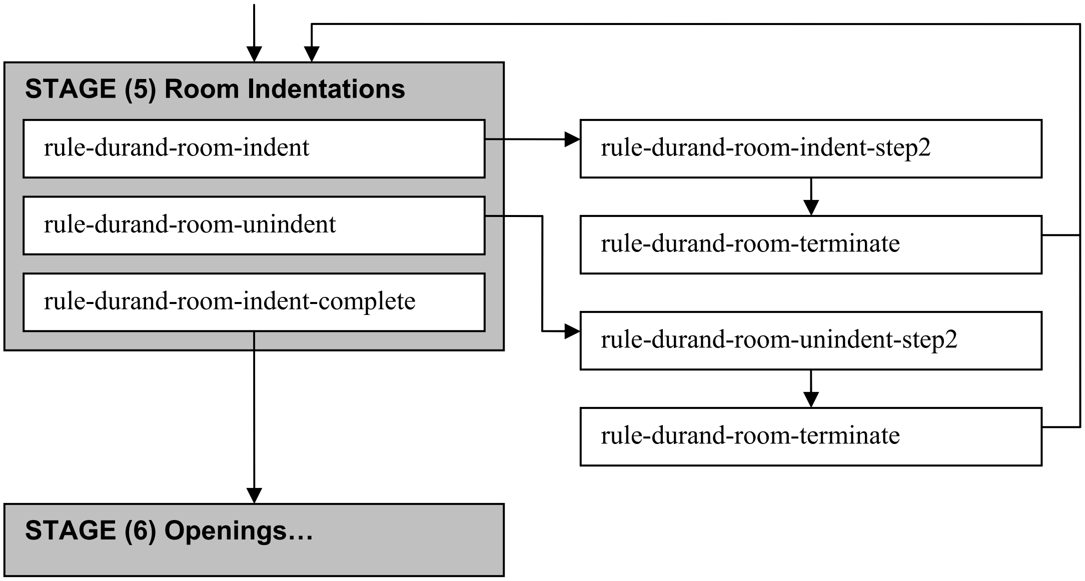
All three macros in this stage are procedural macros. In fact, all three rules use the same left-schema to find a room. And all three macros use the same last rule, *rule-durand-room-terminate*, to remove the focus rectangle. The only difference between these macros is the second rule which performs the actual task of either adding columns all around the room, adding columns at the corners of the room or erasing all the columns in a room. Because of the focus polygon, the actions of the secondary rule are confined to the selected room.

The rules are designed in this manner so that the user can quickly see all the available options in the stage. An alternative method is to combine all three macros into one macro. In this case, the first rule would ask the user to select a room. Then the options for adding and erasing columns would appear in the rule-set. And, as usual, the last step is to remove the focus polygon. This alternative design of the rules is possible because the primary rule of the all three macros is the same. The only difference is the action that needs to be performed inside the room. An abridged version of the rules in the Room Columns stage is shown in figure 6.14.


rule-durand-room-all-columns
w(max)
h(max)
0.1gu
0.1gu
rule-durand-room-corner-columns
w(max)
h(max)
0.1gu
0.1gu
rule-durand-room-erase-columns
w(max)
h(max)
0.1gu
0.1gu
rule-durand-room-all-columns-step2
gu
0.9gu
0.2gu
rule-durand-room-corner-columns-step2
0.9gu
0.9gu
0.2gu
rule-durand-room-erase-columns-step2
0.2gu
S_Ø
rule-durand-room-terminate
w(max)
h(max)
S_Ø
rule-durand-room-columns-complete
S_Ø
S_Ø

6.14 The abridged version of the rules for the Room Columns stage of the Durand grammar. In this stage, the user has the option to add columns to selected rooms in the design.

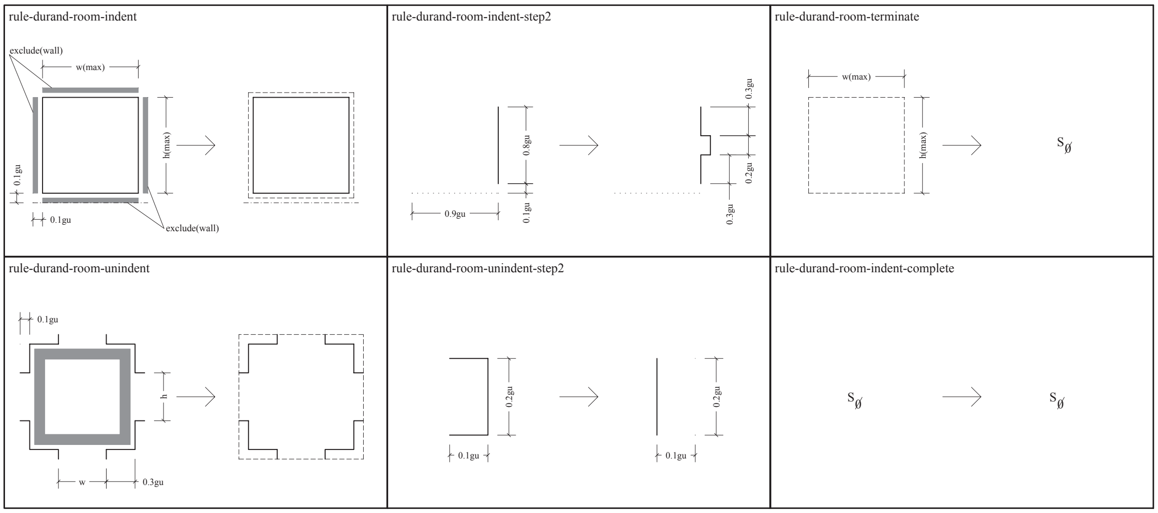
The fifth stage of the grammar is the Room Indentations stage (figure 6.15). This stage is composed of three rules where two of the rules are procedural macros. In this stage, the user can place niches along the inside wall of a room or remove the niches. Making an indentation in the wall is the equivalent to making a niche.


STAGE (5) Room Indentations
rule-durand-room-indent
rule-durand-room-unindent
rule-durand-room-indent-complete
rule-durand-room-indent-step2
rule-durand-room-terminate
rule-durand-room-unindent-step2
rule-durand-room-terminate
STAGE (6) Openings…

6.15 The Room Indentations stage of the Durand grammar. In this stage, the user can add or remove indentations from a room.

The left-hand schema of rule *rule-durand-room-indent* looks for a room where there are no wall lines close to the room. This is done by using the exclude zone to define a zone around the room where there are no wall labeled lines. The reason for the use of the exclude zone is to prevent the placement of two indented rooms next to each other. Because the indentations are half the thickness of the wall, placing niches on both sides of the wall will create a hole going through the wall. From my analysis of Durand's drawing, this condition is not part of his vocabulary.

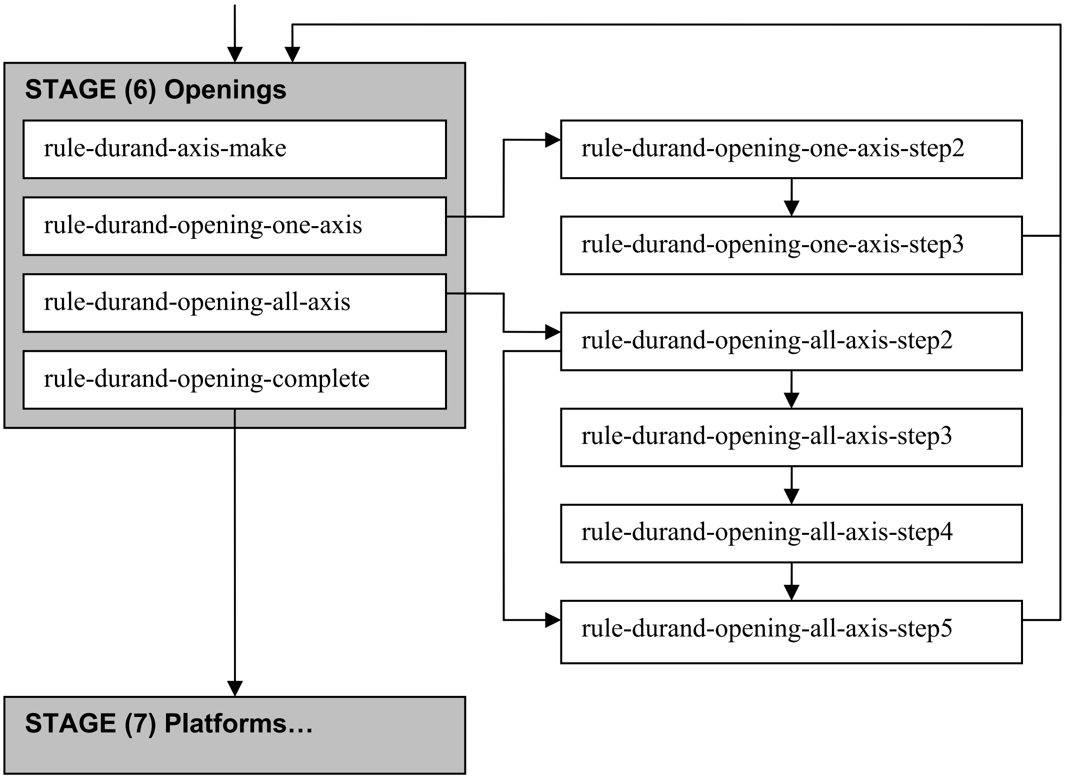
This stage of the grammar also includes a procedural macro to fill in the niches to "unindent" a wall. The left-hand schema of the macro *rule-durand-room-unindent* looks for four indented corners of a room and places a focus rectangle around the four corners. A void zone is used to prevent the rule from picking two collinear rooms. The next step is to erase in all the niches and replace them with a straight wall line. The final step is to remove the focus rectangle. An abridged version of the rules in the Room Indentation stage is shown in figure 6.16.



6.16 The abridged version of the rules for the Room Indentations stage of the Durand grammar. In this stage, the user can add or remove indentations from a room.

The sixth stage of the grammar is the Openings stage (figure 6.17). This stage has four rules where two of the rules are macros. According to Durand, all doorways or windows must be made on walls that intersect the axis line. The two procedural macros in this stage adhere to those restrictions. *Rule-durand-opening-one-axis*, makes one opening at the point where a wall intersects an axis. *Rule-durand-opening-all-axis*, makes an opening on all the walls that intersect a selected axis line. *Rule-durand-axis-make* is included in this stage to make openings in areas where there are no axis lines.

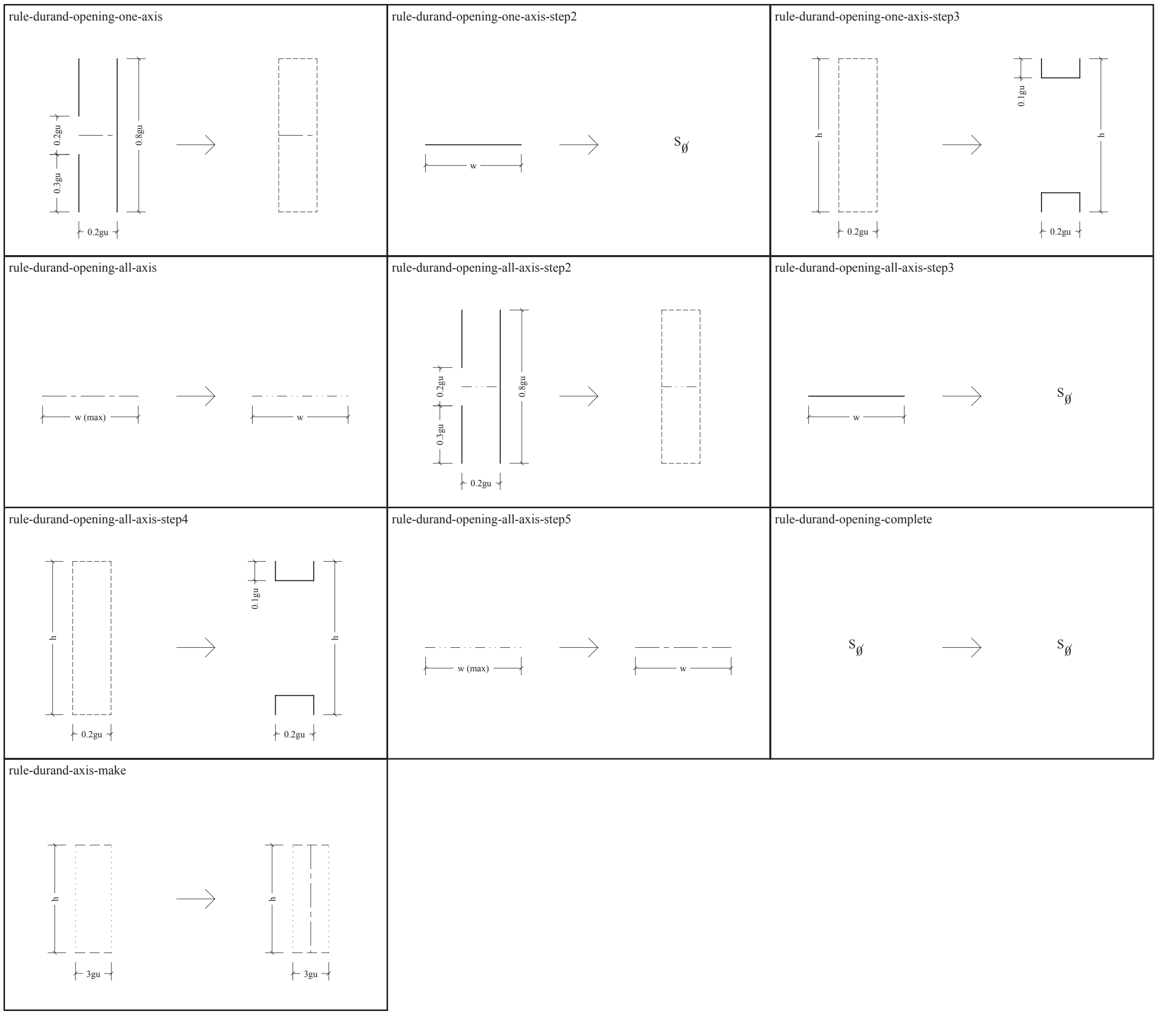
Because this stage comes after the Room Indentations stage, the rules for making an opening have to account for not only solid walls but also walls with niches. I could have included one rule for each type of wall. One rule to make an opening for solid walls and one rule to make an opening in niched walls. I have instead decided to combine the two conditions to simplify the choices for the user.


STAGE (6) Openings
rule-durand-axis-make
rule-durand-opening-one-axis
rule-durand-opening-all-axis
rule-durand-opening-complete
rule-durand-opening-one-axis-step2
rule-durand-opening-one-axis-step3
rule-durand-opening-all-axis-step2
rule-durand-opening-all-axis-step3
rule-durand-opening-all-axis-step4
rule-durand-opening-all-axis-step5
STAGE (7) Platforms…

6.17 The Openings stage of the Durand grammar. In this stage, the user can create openings on any wall that is intersected by an axis line.

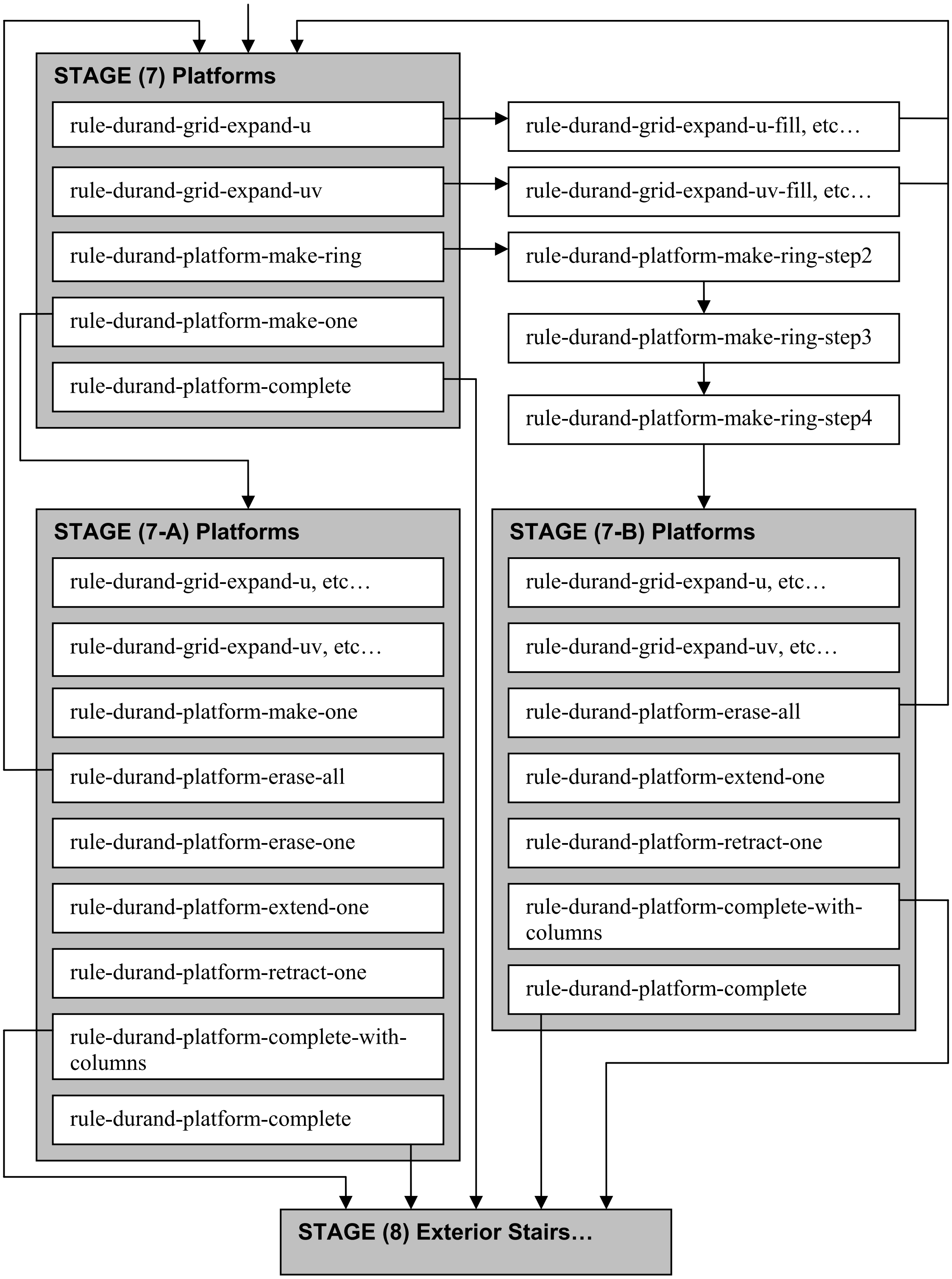
To create an opening, the macro *rule-durand-opening-one-axis*, first looks for a wall segment that intersects an axis. The wall segment contains lines that are common to both solid and niched walls. Once that is found, the wall lines are erased and a focus rectangle is placed around the area where the opening should go. The next rule erases all wall labeled lines in the area. This is to ensure that the entire niche is erased. The final step is to remove the focus rectangle and cap off the ends of the wall to create the opening.

The rule, *rule-durand-opening-all-axis*, which creates an opening on all the walls that intersect an axis line, uses a similar sequence of rules. The first step in the macro is for the user to select an axis line and transform it into a temporary labeled line. The subsequent rules are the same as the macro *rule-durand-opening-one-axis*, except the axial labeled line has been replace with the temporary labeled line. By applying the secondary rules in parallel, all the walls that intersect the temporary line will have an opening. The last step of the macro is to change the temporary line back to an axis line. An abridged version of the rules in the Openings stage is shown in figure 6.18.


rule-durand-opening-one-axis
0.2gu
0.3gu
0.8gu
0.2gu
rule-durand-opening-one-axis-step2
w
$S_\emptyset$
rule-durand-opening-one-axis-step3
h
0.2gu
0.1gu
h
0.2gu
rule-durand-opening-all-axis
w (max)
w
rule-durand-opening-all-axis-step2
0.2gu
0.3gu
0.8gu
0.2gu
rule-durand-opening-all-axis-step3
w
$S_\emptyset$
rule-durand-opening-all-axis-step4
h
0.2gu
0.1gu
h
0.2gu
rule-durand-opening-all-axis-step5
w (max)
w
rule-durand-opening-complete
$S_\emptyset$
$S_\emptyset$
rule-durand-axis-make
h
3gu
h
3gu

6.18 The abridged version of the rules for the Openings stage of the Durand grammar. In this stage, the user can create openings on any wall that is intersected by an axis line.

The seventh stage of the grammar is the Platforms stage (figure 6.19). This stage is composed of two sub-stages which contain different sets of rules depending on the type of platform created. The objective of this stage is to create a platform or base for the building. There are 20 different rules used in this stage where only three of the rules are macros. The initial stage has five rules. The user can create one platform against the building or make a ringed platform that surrounds the entire building. This stage also includes grid expansion rules if the grid is not large enough to accommodate the platforms.


STAGE (7) Platforms
rule-durand-grid-expand-u
rule-durand-grid-expand-uv
rule-durand-platform-make-ring
rule-durand-platform-make-one
rule-durand-platform-complete
rule-durand-grid-expand-u-fill, etc…
rule-durand-grid-expand-uv-fill, etc…
rule-durand-platform-make-ring-step2
rule-durand-platform-make-ring-step3
rule-durand-platform-make-ring-step4
STAGE (7-A) Platforms
rule-durand-grid-expand-u, etc…
rule-durand-grid-expand-uv, etc…
rule-durand-platform-make-one
rule-durand-platform-erase-all
rule-durand-platform-erase-one
rule-durand-platform-extend-one
rule-durand-platform-retract-one
rule-durand-platform-complete-with-columns
rule-durand-platform-complete
STAGE (7-B) Platforms
rule-durand-grid-expand-u, etc…
rule-durand-grid-expand-uv, etc…
rule-durand-platform-erase-all
rule-durand-platform-extend-one
rule-durand-platform-retract-one
rule-durand-platform-complete-with-columns
rule-durand-platform-complete
STAGE (8) Exterior Stairs…

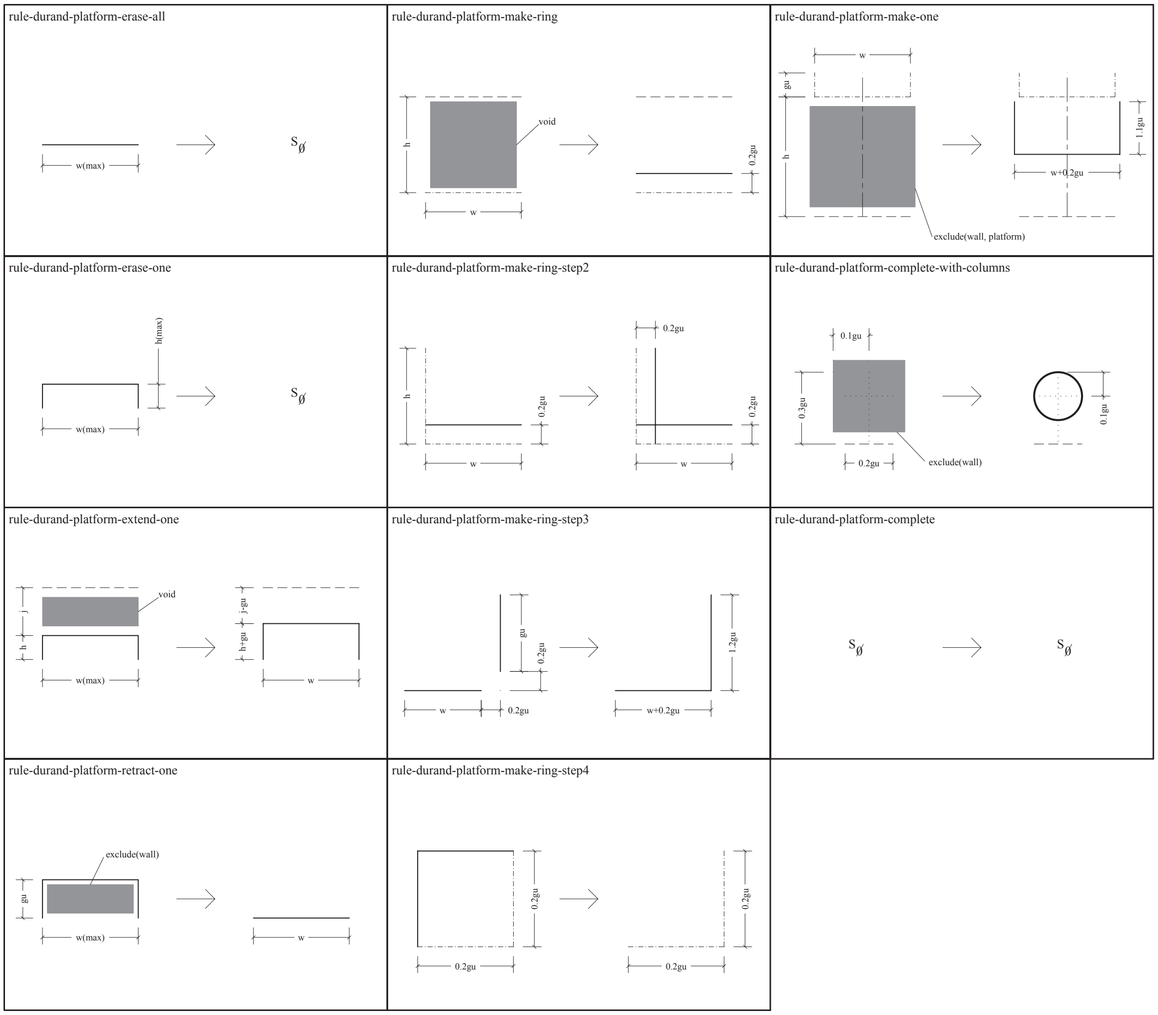
6.19 The Platform stage of the Durand grammar. There are two sub-stages depending on if the user created a ringed platform (7-B) or a single platform (7-A).

If the user decides to make one platform using *rule-durand-platform-make-one*, one platform is created and the rule-set is changed to provide the user with options to modify the newly created platform (stage 7-A). These options include erasing, extending, retracting or making another platform. Should the user decide that he wants to make a ringed platform instead, he must erase all of the platforms with rule, *rule-durand-platform-erase-all*, which will also change the rule-set back to the original definition (stage 7).

The serial macro, *rule-durand-platform-make-ring*, creates a platform around the entire building in five steps. Since the building can be any rectilinear shape, the macro has to be able to accommodate the various forms of the building outline. The rules in the macro use the parti lines to identify the edge of the building. The placement of the new platform lines is at a fixed distance proportional to the grid size.

The first step of the macro is to place a platform line against those sections of the building that are protruding out. The second step is to place a platform line along the sides of the protruding sections from the first step. The third step is to fill in any U-shaped conditions along the platform. The fourth step fillets the concave corners while the fifth step trims the convex corners of the platform. The last step also changes the rule-set to be stage 7-B.

The rule-set in stage 7-B is the same as stage 7-A except two rules are not included in the rule-set. The two missing rules are *rule-durand-platform-make-one* and *rule-durand-platform-erase-one*. Because the user has created a ringed platform around the entire building, it does not make sense to allow the user to erase a portion of the platform or to make another platform. If the user wants to create multiple single platforms, he must erase the ringed platform and then make single platforms. Once the user creates a desirable platform arrangement, the user can go to the next stage without modifications or go to the next stage with added columns at the edge of the platform. An abridged version of the rules in the Platform stage is shown in figure 6.20.

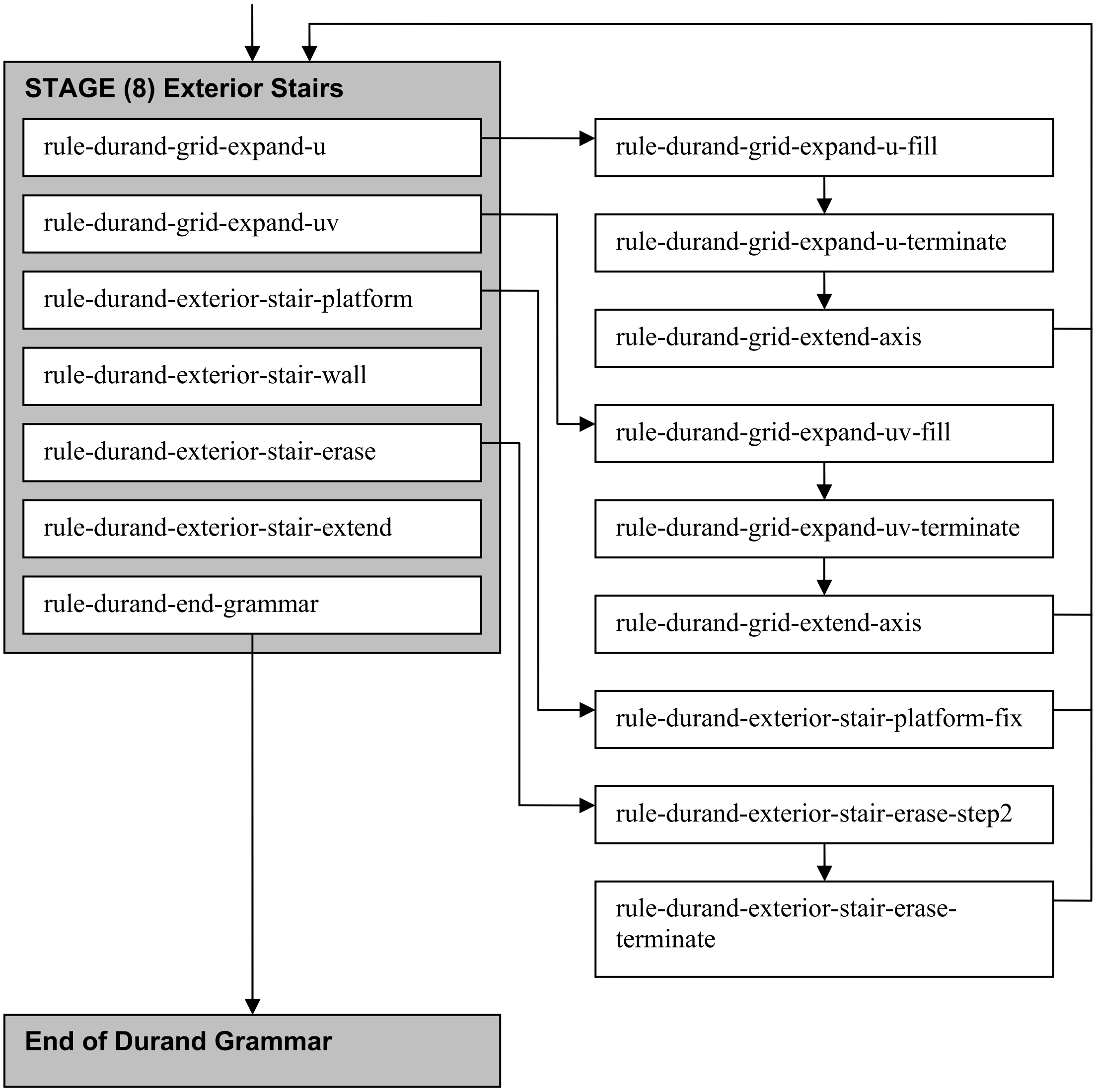

rule-durand-platform-erase-all
w(max)
S∅
rule-durand-platform-make-ring
void
h
w
0.2gu
rule-durand-platform-make-one
w
gu
h
exclude(wall, platform)
1.1gu
w+0.2gu
rule-durand-platform-erase-one
h(max)
w(max)
S∅
rule-durand-platform-make-ring-step2
h
w
0.2gu
0.2gu
w
0.2gu
rule-durand-platform-complete-with-columns
0.1gu
0.3gu
0.2gu
exclude(wall)
0.1gu
rule-durand-platform-extend-one
void
j
h
w(max)
j-gu
h+gu
w
rule-durand-platform-make-ring-step3
gu
0.2gu
w
0.2gu
1.2gu
w+0.2gu
rule-durand-platform-complete
S∅
S∅
rule-durand-platform-retract-one
exclude(wall)
gu
w(max)
w
rule-durand-platform-make-ring-step4
0.2gu
0.2gu
0.2gu
0.2gu

6.20 The abridged version of the rules for the Platform stage of the Durand grammar. There are two sub-stages depending on if the user created a ringed platform (7-B) or a single platform (7-A).

The last stage of the grammar builds the exterior stairs of the building (figure 6.21). This stage has seven rules where four of the rules are macros. The options in this stage allow the user to place an exterior stair against a wall or a platform. There are also options to erase a staircase or extend the length of the staircase. Since the extension of the staircase requires a larger grid, rules to expand the grid are also included in this stage.

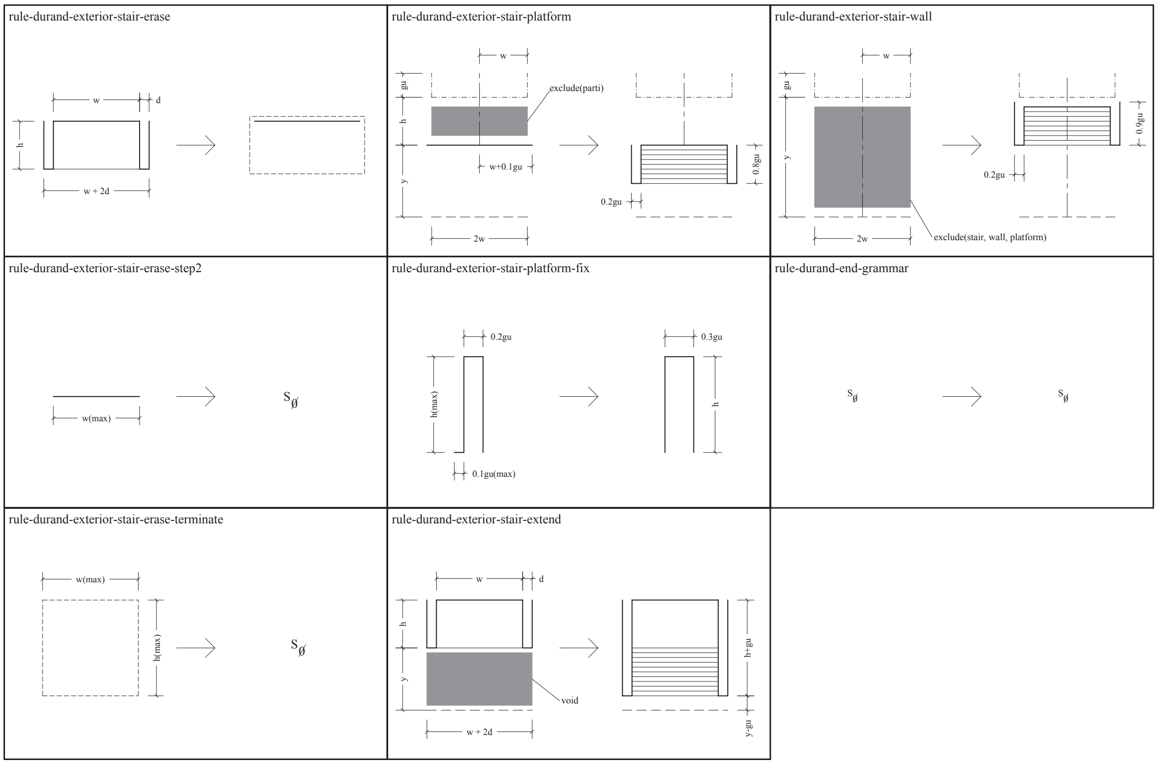
To erase a staircase requires a rule that can find a variety of staircase sizes since the rules in this stage allow the user to extend the length of the stairs. Instead of enumerating rules for all possible staircase sizes, a procedural macro is defined to erase the staircases. The first step of the macro *rule-durand-exterior-stair-erase*, places a focus rectangle around the staircase. This isolates the working area. The next step is to erase all the stair and platform labeled lines in that area. The last step is to remove the focus rectangle so that subsequent rule applications apply to the entire drawing.

When placing a stair against a platform using *rule-durand-exterior-stair-platform*, sometimes a slight niche is created in the side of the platform. I found this unattractive and added a secondary rule that adjusts the size of the platform so that the platform is always flush with the rest of the building.


STAGE (8) Exterior Stairs
rule-durand-grid-expand-u
rule-durand-grid-expand-uv
rule-durand-exterior-stair-platform
rule-durand-exterior-stair-wall
rule-durand-exterior-stair-erase
rule-durand-exterior-stair-extend
rule-durand-end-grammar
rule-durand-grid-expand-u-fill
rule-durand-grid-expand-u-terminate
rule-durand-grid-extend-axis
rule-durand-grid-expand-uv-fill
rule-durand-grid-expand-uv-terminate
rule-durand-grid-extend-axis
rule-durand-exterior-stair-platform-fix
rule-durand-exterior-stair-erase-step2
rule-durand-exterior-stair-erase-terminate
End of Durand Grammar

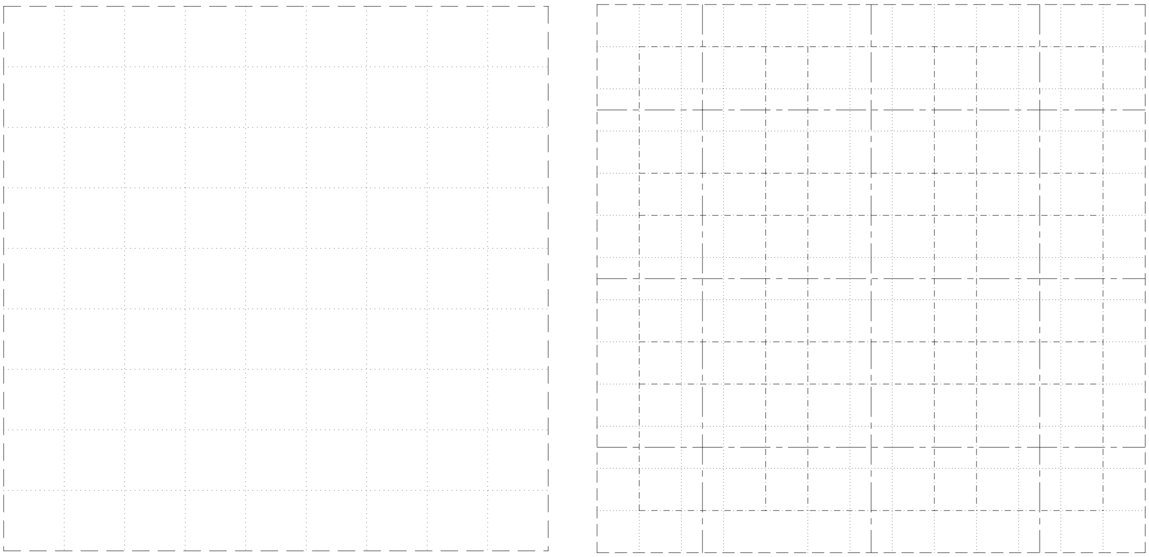
6.21 The Exterior Stairs stage of the Durand grammar. This stage is the last stage of the grammar which allows the user to add exterior staircases to the building. An exterior staircase can be abutted against the wall or against a platform.

The Durand grammar terminates when *rule-durand-end-grammar* is applied which removes all rules from the rule-set. To summarize, the Durand grammar is composed of eight stages. At each stage of the grammar, the user is allowed to modify only certain aspects of the design. The grammar uses the rule-set descriptor to guide the user through the design process. The first stage is to create the grid and axis. From the grid and axis, a default parti is created. In the second stage, the user can modify the parti to suit his needs. Once the parti is fixed, the walls are generated. In the third stage, the user can alter the arrangement of walls. Once the walls are fixed, the user can place columns inside the rooms of the building. After placing columns, the user can now place indentations in the walls. The next step is to make the openings in the walls. Once that is set, the user can create a platform around the building. The last step is to place exterior stairs. An abridged version of the rules in the Exterior Stair stage is shown in figure 6.22.


rule-durand-exterior-stair-erase
w
d
h
w + 2d
rule-durand-exterior-stair-platform
w
gu
h
exclude(parti)
w+0.1gu
y
2w
0.8gu
0.2gu
rule-durand-exterior-stair-wall
w
gu
y
2w
exclude(stair, wall, platform)
0.9gu
0.2gu
rule-durand-exterior-stair-erase-step2
w(max)
$S_\emptyset$
rule-durand-exterior-stair-platform-fix
0.2gu
h(max)
0.1gu(max)
0.3gu
h
rule-durand-end-grammar
$S_\emptyset$
$S_\emptyset$
rule-durand-exterior-stair-erase-terminate
w(max)
h(max)
$S_\emptyset$
rule-durand-exterior-stair-extend
w
d
h
y
void
w + 2d
h+gu
y-gu

6.22 The abridged version of the rules for the Exterior Stairs stage of the Durand grammar. This stage is the last stage of the grammar which allows the user to add exterior staircases to the building. An exterior staircase can be abutted against the wall or against a platform.

Figures 6.23-6.28 summarizes the eight stages of the Durand grammar. A sample derivation is shown at the top with the matching rule-set at the bottom. The derivation is a sample design in Durand's book shown on the right in figure 6.29.

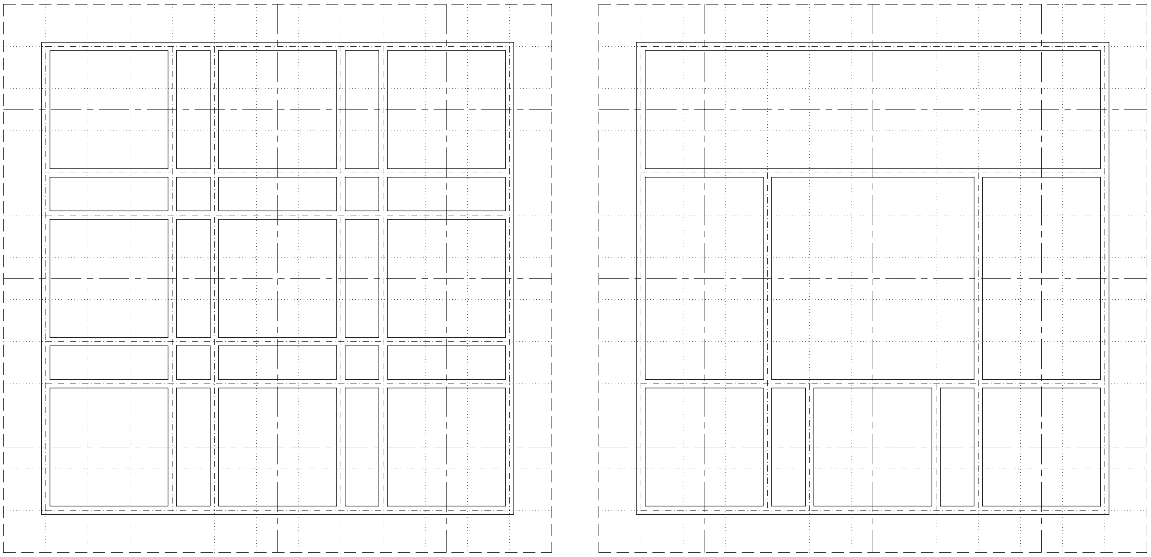


STAGE (0) Initial Shape

STAGE (1) Grid and Axis

rule-durand-axis-make
rule-durand-axis-erase
rule-durand-axis-move
rule-durand-axis-pair
rule-durand-axis-bisect-grid
rule-durand-grid-expand-u
  rule-durand-grid-expand-u-fill
  rule-durand-grid-expand-u-terminate
  rule-durand-grid-extend-axis
rule-durand-grid-expand-uv
  rule-durand-grid-expand-uv-fill
  rule-durand-grid-expand-uv-terminate
  rule-durand-grid-extend-axis
rule-durand-parti-generate

6.23 Diagram summarizing the initial shape and the first stage (Grid and Axis) of the Durand grammar. In the first stage of the grammar the user makes an arrangement of axis lines which is the basis for the parti. The top image shows the final result of the stage mentioned in the box directly below the image. Each box shows all the rules used in the stage including secondary rules of a macro. The black colored texts are rules in the rule-set. The gray colored texts are the secondary rules of a macro and are typically excluded from the rule-set.



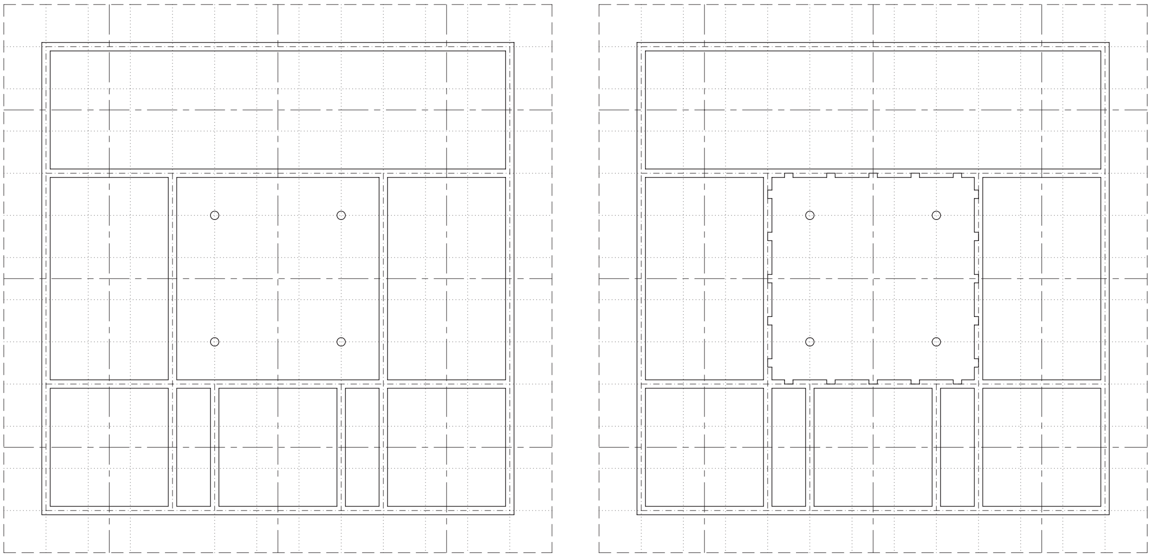
STAGE (2) Parti

rule-durand-parti-lengthen
rule-durand-parti-shorten
rule-durand-parti-widen
rule-durand-parti-narrow
rule-durand-parti-move-on-axis
rule-durand-parti-shift-with-axis
rule-durand-wall-generate
    rule-durand-wall-trim-+shape
    rule-durand-wall-trim-Tshape
    rule-durand-wall-trim-Lshape

STAGE (3) Walls

rule-durand-wall-erase
rule-durand-wall-column
    rule-durand-wall-column-step2
    rule-durand-wall-column-step3
rule-durand-room-divide
    rule-durand-room-divide-step2
    rule-durand-room-terminate
rule-durand-room-remove-bay
rule-durand-wall-complete

6.24 Diagram summarizing the second (Parti) and third (Walls) stage of the Durand grammar. In the Parti stage, the user can adjust the design of the parti lines. The final step, in the Parti stage, is to generate the walls based on the parti lines. In the Walls stage, the user can modify the arrangements of the walls by erasing walls, adding walls, or converting a wall into a series of columns.



STAGE (4) Room Columns

rule-durand-room-all-columns
- rule-durand-room-all-columns-step2
- rule-durand-room-terminate

rule-durand-room-corner-columns
- rule-durand-room-corner-columns-step2
- rule-durand-room-terminate

rule-durand-room-erase-columns
- rule-durand-room-erase-columns-step2
- rule-durand-room-terminate

rule-durand-room-complete

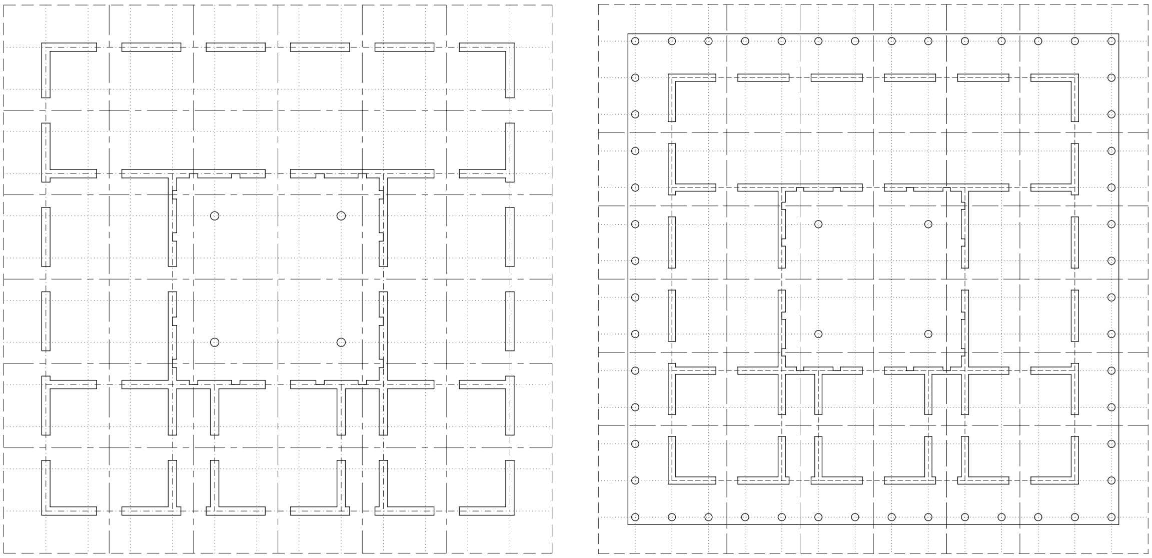
STAGE (5) Room Indentations

rule-durand-room-indent
- rule-durand-room-indent-step2
- rule-durand-room-terminate

rule-durand-room-unindent
- rule-durand-room-unindent-step2
- rule-durand-room-terminate

rule-durand-room-indent-complete

6.25 Diagram summarizing the fourth (Room Columns) and fifth (Room Indentations) stages of the Durand grammar. In the Room Columns stage, the user can add or remove columns in a room. In the Room Indentations stage, the user can add or remove indentations to a room.



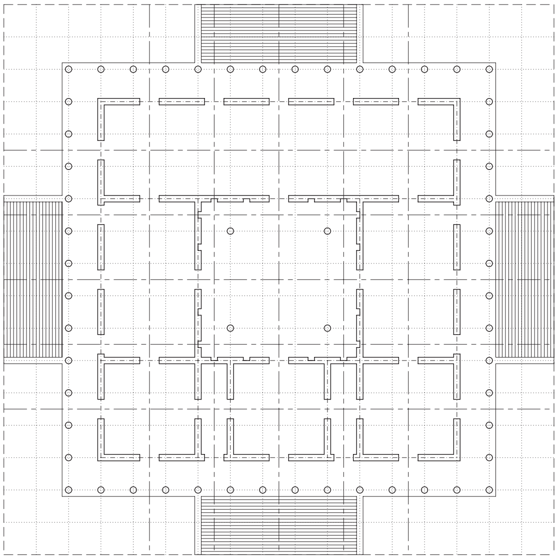
STAGE (6) Openings

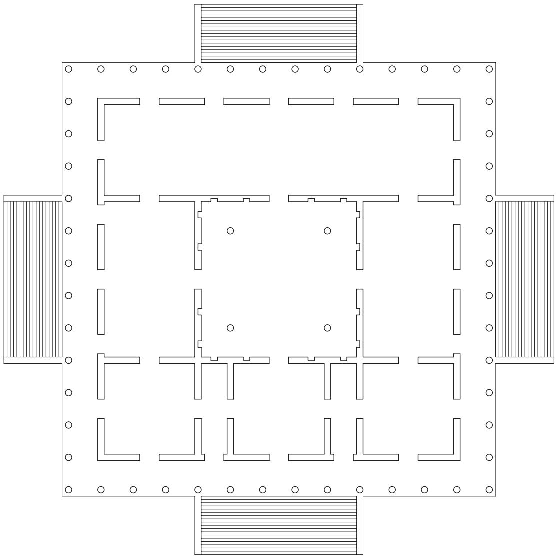
rule-durand-axis-make
rule-durand-opening-one-axis
    rule-durand-opening-one-axis-step2
    rule-durand-opening-one-axis-step3
rule-durand-opening-all-axis
    rule-durand-opening-all-axis-step2
    rule-durand-opening-all-axis-step3
    rule-durand-opening-all-axis-step4
    rule-durand-opening-all-axis-step5
rule-durand-opening-complete

STAGE (7) Platforms

rule-durand-grid-expand-u
    rule-durand-grid-expand-u-fill
    rule-durand-grid-expand-u-terminate
    rule-durand-grid-extend-axis
rule-durand-grid-expand-uv
    rule-durand-grid-expand-uv-fill
    rule-durand-grid-expand-uv-terminate
    rule-durand-grid-extend-axis
rule-durand-platform-make-ring
    rule-durand-platform-make-ring-step2
    rule-durand-platform-make-ring-step3
    rule-durand-platform-make-ring-step4
rule-durand-platform-make-one
rule-durand-platform-erase-all
rule-durand-platform-erase-one
rule-durand-platform-extend-one
rule-durand-platform-retract-one
rule-durand-platform-complete-with-columns
rule-durand-platform-complete

6.26 Diagram summarizing the sixth (Openings) and seventh (Platforms) stages of the Durand grammar. In the Openings stage, the user can make openings along any axis line that intersects a wall. The Platforms stage creates a platform around the outside edge of the building.





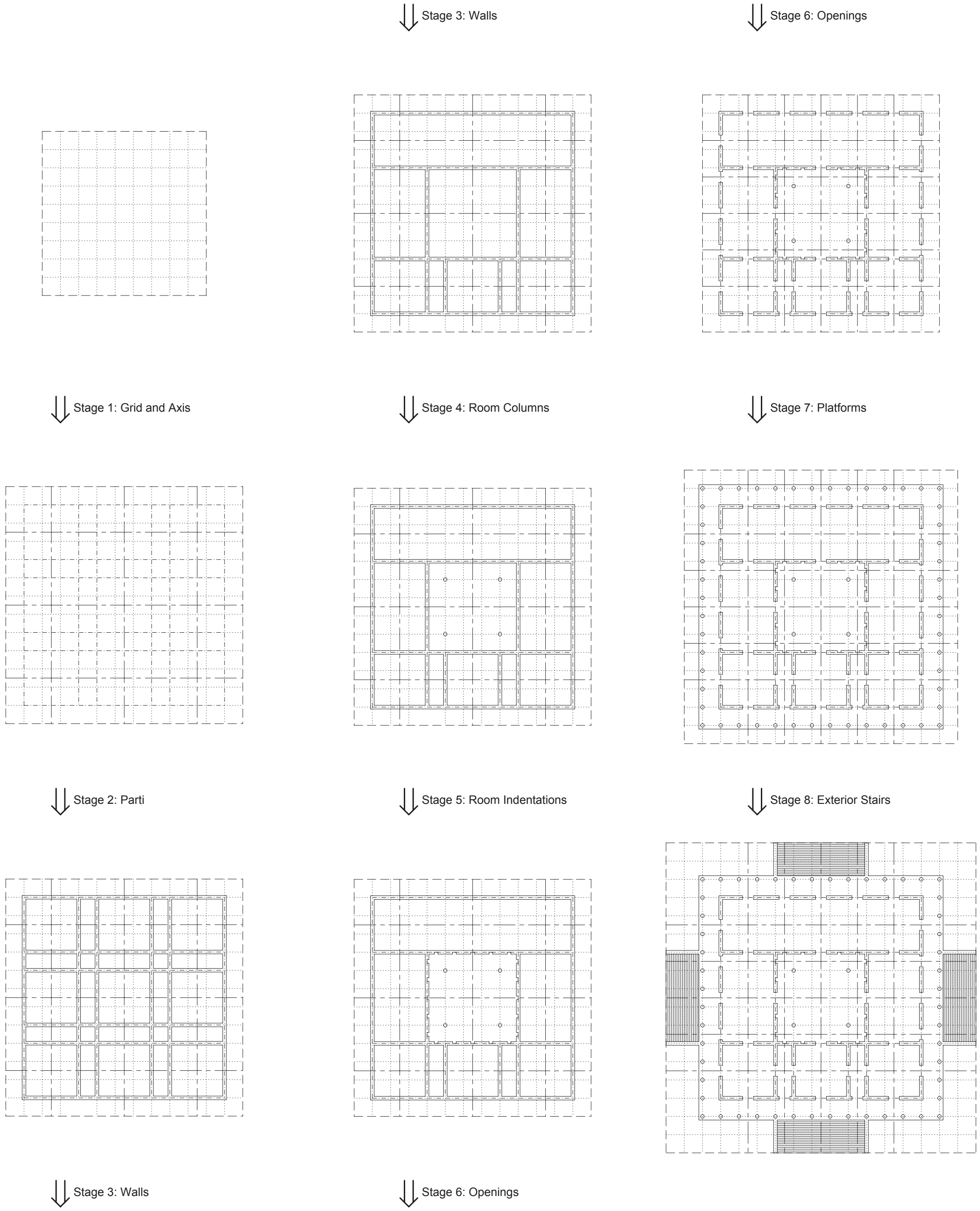
STAGE (8) Exterior Stairs

rule-durand-grid-expand-u
  rule-durand-grid-expand-u-fill
  rule-durand-grid-expand-u-terminate
  rule-durand-grid-extend-axis
rule-durand-grid-expand-uv
  rule-durand-grid-expand-uv-fill
  rule-durand-grid-expand-uv-terminate
  rule-durand-grid-extend-axis
rule-durand-exterior-stair-platform
  rule-durand-exterior-stair-platform-fix
rule-durand-exterior-stair-wall
rule-durand-exterior-stair-erase
  rule-durand-exterior-stair-erase-step2
  rule-durand-exterior-stair-erase-terminate
rule-durand-exterior-stair-extend
rule-durand-end-grammar

Final Design

Showing only the labeled shapes wall, platform, and stair.

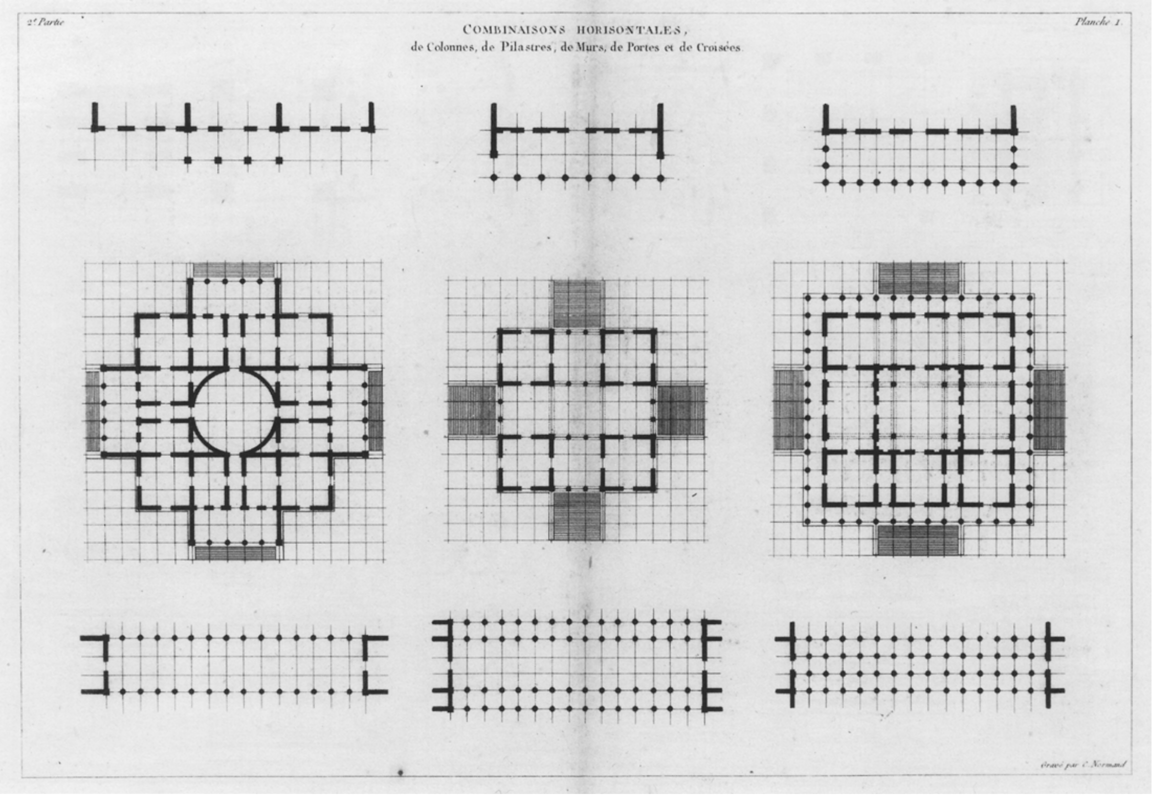
6.27 Diagram summarizing the eighth and last stage (Exterior Stairs) of the Durand grammar. Here the user can place exterior stairs around the exterior edge of the building. The final design shows the floor plan without the grid lines, parti lines, or axis lines.



6.28 Derivation summarizing the eight stages of the Durand grammar.

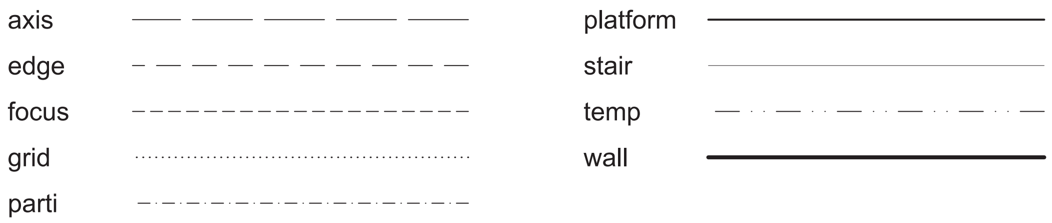
6.3 Derivation of the Durand grammar

This section will show a sample derivation of the Durand grammar. The example is based on one of Durand's drawings and will demonstrate how the meta-language descriptors can be used in a shape grammar to produce some of Durand's architectural floor plans. The derivation is based on one of the floor plans in *Précis of the Lectures on Architecture* (figure 6.29 center). The floor plan has four corner rooms with columns along the central corridors. The building is a symmetrical about the horizontal and vertical axis and sits on top of a platform which can be reached by four staircases placed on axis.


2e Partie
COMBINAISONS HORISONTALES,
de Colonnes, de Pilastres, de Murs, de Portes et de Croisées
Planche 1.
Gravé par C. Normand

6.29 Plate 1 in Part II of *Précis of the Lectures on Architecture*. The drawing shows three sample floor plans. The derivation in this section will make the center floor plan.

The Durand grammar uses nine labels: axis, edge, focus, grid, parti, platform, stair, temp, and wall. The different labeled lines are distinguished by a variety of linetypes and lineweights (figure 6.30). The axis label is used for the axis lines. The edge label is used for lines that define the edge of the grid. The grid label is used for the interior lines of the grid. The focus label is used for the focus descriptor. The parti, platform, stair, and wall labeled lines are used to represent their respective architectural elements. The temp label is used for a temporary line.



6.30 The linetype chart for each of the different labeled lines in the Durand grammar.

All the steps for the derivation are shown in figure 6.31. Each frame is composed of two parts: the drawing and its corresponding rule-set parallel description. Between each frame, next to the arrow, is the rule selected to go to the next frame. There are a total of 42 steps in this derivation and the grammar begins with the initial shape of a 9x9 grid in frame 1.

Frame 1: A 9x9 grid is the initial shape for the Durand grammar. On the right is the initial rule-set which contains eight rules. The variable "gu", which determines the distance of one grid unit in the grid, is also set at this time.

Frame 2: In this frame, the user creates a vertical axis in the center of the grid using *rule-durand-axis-bisect-grid.* This rule finds the edge of the grid and places an axis line down the middle. Since the number of grids will always be odd, the rule will always place an axis line that bisects the center grids.

Frame 3: The macro *rule-durand-grid-expand-uv* is selected to expand the grid in both the u and v direction. A macro is used because the size of the grid can vary. The first step in the macro expands the grid by one grid unit in all four directions and places four focus rectangles around the new expanded area. This isolates the area where changes are needed. The next rule to apply is restricted by the success rule. Note: the scale of the drawing has been shrunk so the entire drawing will fit inside the frame.

Frame 4: *Rule-durand-grid-expand-uv-fill* fills the areas inside the four focus rectangles with grid lines to complete the grid.

Frame 5: *Rule-durand-grid-expand-uv-terminate* removes the four focus rectangles so that the next rule can apply to the entire drawing.

Frame 6: The final step is to make sure that all the axis lines extend to the edge of the grid. *Rule-durand-grid-extend-axis* searches the drawing for maximal axis lines and extends them by one grid unit at both endpoints. After this step, all the rules in the stage become available again because this is the last rule in the macro.

Frame 7: A horizontal axis line is created using *rule-durand-axis-make*.

Frame 8: A pair of vertical axis lines is created around the center vertical axis using *rule-durand-axis-make-pair*.

Frame 9: A pair of horizontal axis lines is created around the center horizontal axis using *rule-durand-axis-make-pair*.

Frame 10: *Rule-durand-parti-generate* creates a default parti by placing, in parallel, parti labeled rectangles around all axis lines. The rectangles are three units wide and its length is two units short of the maximal length of the axis line. In this case, the default parti creates a 3x3 grid. The rule also changes the rule-set so that rules in the next stage of the grammar, which is Parti, are available for use.

Frame 11: Since the parti arrangement is what we want (figure 6.18 center), we go straight to the next stage of the grammar which is to generate the walls. This process is done using a four step serial macro. The primary rule is *rule-durand-wall-generate* which create a pair of wall labeled lines using the parti line as the centerline. The distance between the parti line and the newly created wall line is half the thickness of the walls. Although this rule does change the rule-set, the next step taken is dictated by the success rule of the directive descriptor.

Frame 12: Once the walls are created, the next step is to trim off and fillet the intersections. There are three types of intersections to modify: cross intersections, T-intersections, and L-intersections. In this step, the cross intersections are trimmed off using *rule-durand-wall-trim-+shape*. There are four cross intersections in the drawing.

Frame 13: After trimming off the cross intersections, the next type of intersection to trim are T-intersections using *rule-durand-wall-trim-Lshape* as dictated by the success rule of the directive descriptor. There are eight T-intersections in the drawing.

Frame 14: The last intersection condition to fix is the L-intersection. Here we trim and fillet the wall lines to finish off the corners of the walls using *rule-durand-wall-trim-Lshape*. There are four L-intersections in the drawing. Since this rule is the last rule of the macro, all the rules in this stage become available to the user after the rule is applied. The final result is a building floor plan with nine rooms.

Frame 15: To get the same design as the center floor plan in figure 6.18, two of the walls in the center of the drawing need to be removed. This can be achieved using *rule-durand-wall-erase*. Notice that the parti lines are removed as well as the wall lines.

Frame 16: Another wall is erased using the rule *rule-durand-wall-erase*.

Frame 17: The next step is to replace some of the walls with a series of columns. The procedural macro used to perform this design transformation has three rules. The primary rule of the macro, *rule-durand-wall-column*, isolates the working area using a focus rectangle. In this case, the working area is the bottom center wall.

Frame 18: The next step in the macro is to place the columns at the grid intersections in the focus rectangle. This is done in parallel to place a column on all grid intersections in the working area. *Rule-durand-wall-columns-step2* places two new columns inside the focus rectangle.

Frame 19: The last step in the macro is to erase the focus lines and walls lines and cap off the ends of the wall using *rule-durand-wall-columns-step3.* Since this rule is the last rule of the macro, all the rules in the rule-set of this stage are again made available to the user.

Frame 20: *Rule-durand-wall-columns* is applied five times to five different walls to have the same design as the center floor plan shown in figure 6.18.

Frame 21: There are no more changes that need to be made to the walls so *rule-durand-wall-complete* is used to go to the next stage which is Room Columns.

Frame 22: The design we want to make does not have any columns inside the rooms. *Rule-durand-room-columns-complete* is applied to go to the next stage which is Room Indentations.

Frame 23: The design we want to make does not have any niches or indentations in the walls. *Rule-durand-room-indentations-complete* is applied to go to the next stage which is Openings.

Frame 24: According to the design, each room should have four openings. These openings should be placed only where an axis line intersects a wall. The macro *rule-durand-opening-all-axis* creates an opening on all walls that intersects a chosen axis line. The reason for using a macro is because there are two different types of walls: regular and indented walls. Instead of using two rules, one for each type of wall, the grammar has one macro that can make openings in either a regular or indented wall. This macro is composed of five rules. The primary rule of the macro converts a selected axis line into a temp labeled line. In this case, the user picks the right vertical axis line.

Frame 25: The second step in the macro is to find all the wall sections that intersect the temp labeled line and place a focus rectangle around those areas. In this case, *rule-durand-opening-all-axis-step2* places four focus rectangles at the four walls that intersect the temp line.

Frame 26: The third step is to erase all the wall lines within the focus rectangles using *rule-durand-opening-all-axis-step3*. The result is an empty area in each of the focus rectangles where a new opening can be placed.

Frame 27: The fourth step is to erase all the focus rectangles and cap off all the ends of the wall labeled lines to make the opening using *rule-durand-opening-all-axis-step4*.

Frame 28: The last step is to change the temp labeled line back to an axis labeled line using *rule-durand-opening-all-axis-step5*. Since this is the last rule in the macro, the other rules in the rule-set become available again.

Frame 29: The macro *rule-durand-opening-all-axis* is applied three times to the other axis lines.

Frame 30: The openings now match the design (figure 6.18 center) and *rule-durand-opening-complete* is applied to go to the next stage which is Platforms.

Frame 31: The next stage is to create a platform that surrounds the entire building. Because the building outline can be any rectilinear shape, using a single rule to create the platform is not possible. Instead, the serial macro *rule-durand-platform-make-ring* is used to account for the varying shapes and sizes of the building. The macro is composed of four rules and creates the platform based on the possible conditions of a rectilinear building parti. These conditions include a U-shape, an L-shaped corner, and a Z-shape. The primary rule of the macro places a platform line at a fixed distance from an inside U-shaped parti, an outside U-shaped parti, and a Z-shape. An inside U-shape is defined as a U-shape where the area within the U is inside the building outline. An outside U-shape is defined as a U-shape where the area within the U is outside of the buildings outline. In this drawing there are four inside U-shape cases, one for each side of the building.

Frame 32: The second step is to place a platform line where an inside or outside L-shaped corner condition exists. In this drawing there are no such conditions so the failure rule is applied which is *rule-durand-platform-make-ring-step3*.

Frame 33: The third step is to fillet the inside L-shaped corners of the platform. An inside L-shaped corner is one where the area within the corner is inside the building. There are four such conditions in this drawing; one for each corner of the building. The next rule to apply is *rule-durand-platform-make-ring-step4*.

Frame 34: The fourth and last step is to trim off any excess platform lines at the outside L-shaped corners. An outside L-shaped corner is one where the area within the corner is outside of the building. In this drawing, there are no such conditions. Since this rule does not have any directive options, all the rules in the rule-set are again available to the user.

Frame 35: The design of the platform does not require any modifications to match the center drawing in figure 6.18 so *rule-durand-platform-complete* is applied to go to the next stage which is Exterior Stairs.

Frame 36: Because the desired staircase size is three grid units in length, the first step is to expand the grid by two grid units using *rule-durand-grid-expand-uv*. Note: the scale of the drawing has been shrunk to fit inside the frame.

Frame 37: In this frame, a staircase is added next to the platform along the bottom center of the drawing using *rule-durand-exterior-stair-platform*. The next rule to apply, *rule-durand-exterior-stair-platform-fix*, is dictated by the success rule.

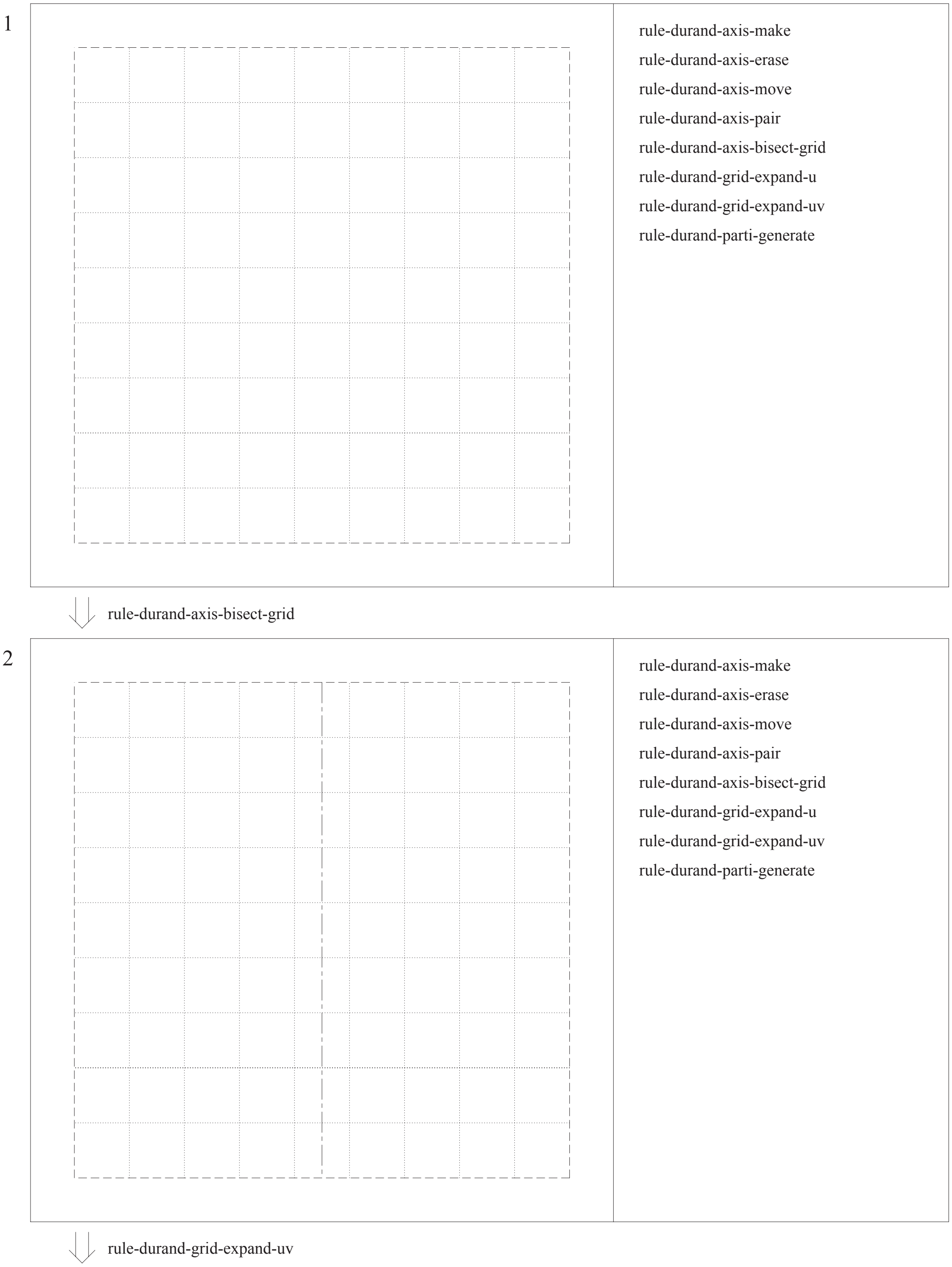
Frame 38: Sometimes a newly made staircase creates a minor notch along the edge of the platform. *Rule-durand-exterior-stair-platform-fix* searches for any possible situations and erases the notch. In this drawing, there are no such conditions.

Frame 39: *Rule-durand-exterior-stair-platform* is applied three times to create the other required staircases.

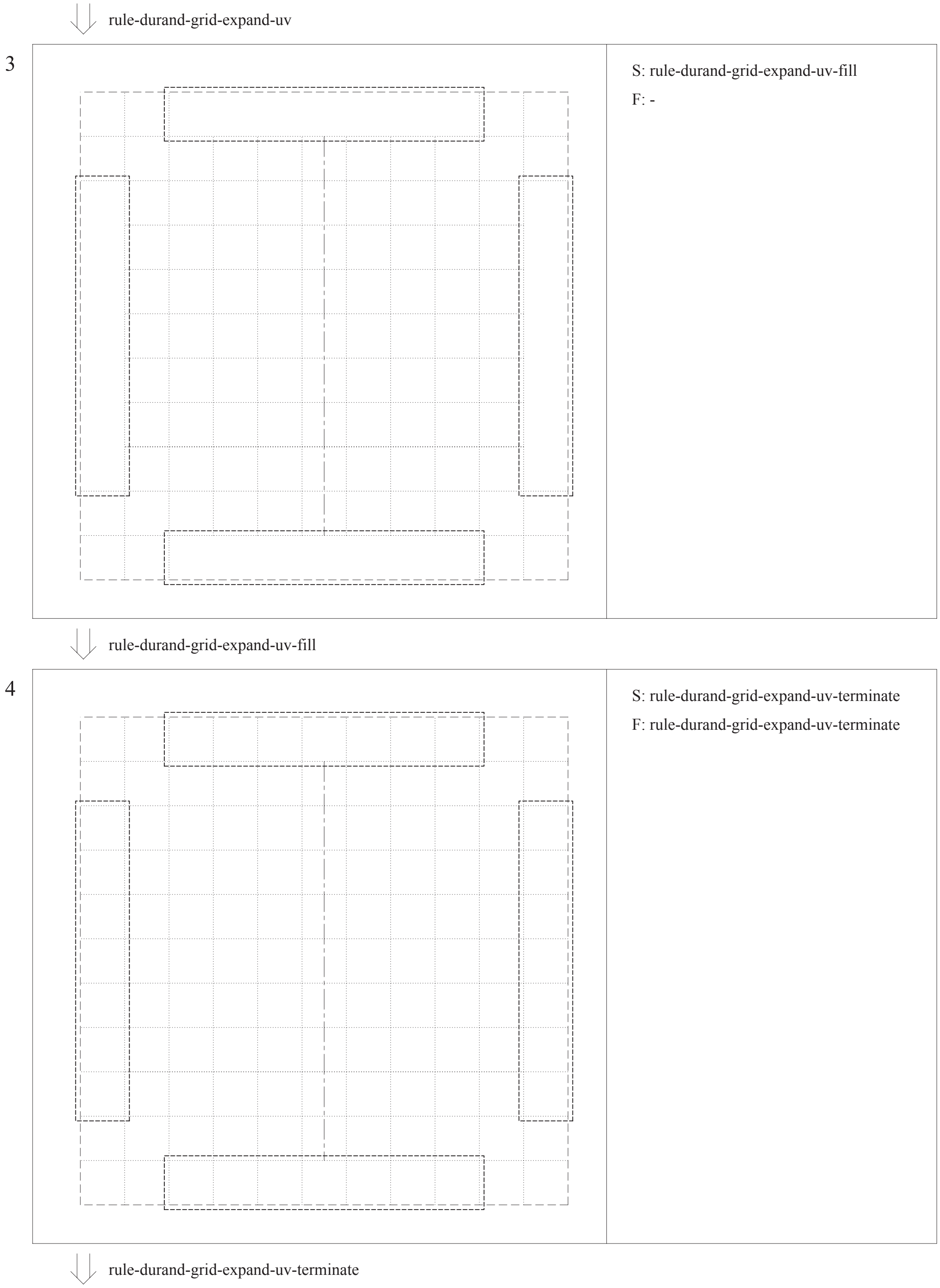
Frame 40: The bottom staircase is extended by using *rule-durand-exterior-stair-extend*.

Frame 41: The stairs are all extended to be three grid units in length using *rule-durand-exterior-stair-extend*.

Frame 42: The design is now complete and *rule-durand-end-grammar* is applied to end the grammar by removing all the rules in the rule-set.


1
rule-durand-axis-make
rule-durand-axis-erase
rule-durand-axis-move
rule-durand-axis-pair
rule-durand-axis-bisect-grid
rule-durand-grid-expand-u
rule-durand-grid-expand-uv
rule-durand-parti-generate
rule-durand-axis-bisect-grid
2
rule-durand-axis-make
rule-durand-axis-erase
rule-durand-axis-move
rule-durand-axis-pair
rule-durand-axis-bisect-grid
rule-durand-grid-expand-u
rule-durand-grid-expand-uv
rule-durand-parti-generate
rule-durand-grid-expand-uv

6.31 Derivation of Durand grammar.



rule-durand-grid-expand-uv

3

S: rule-durand-grid-expand-uv-fill

F: -

rule-durand-grid-expand-uv-fill

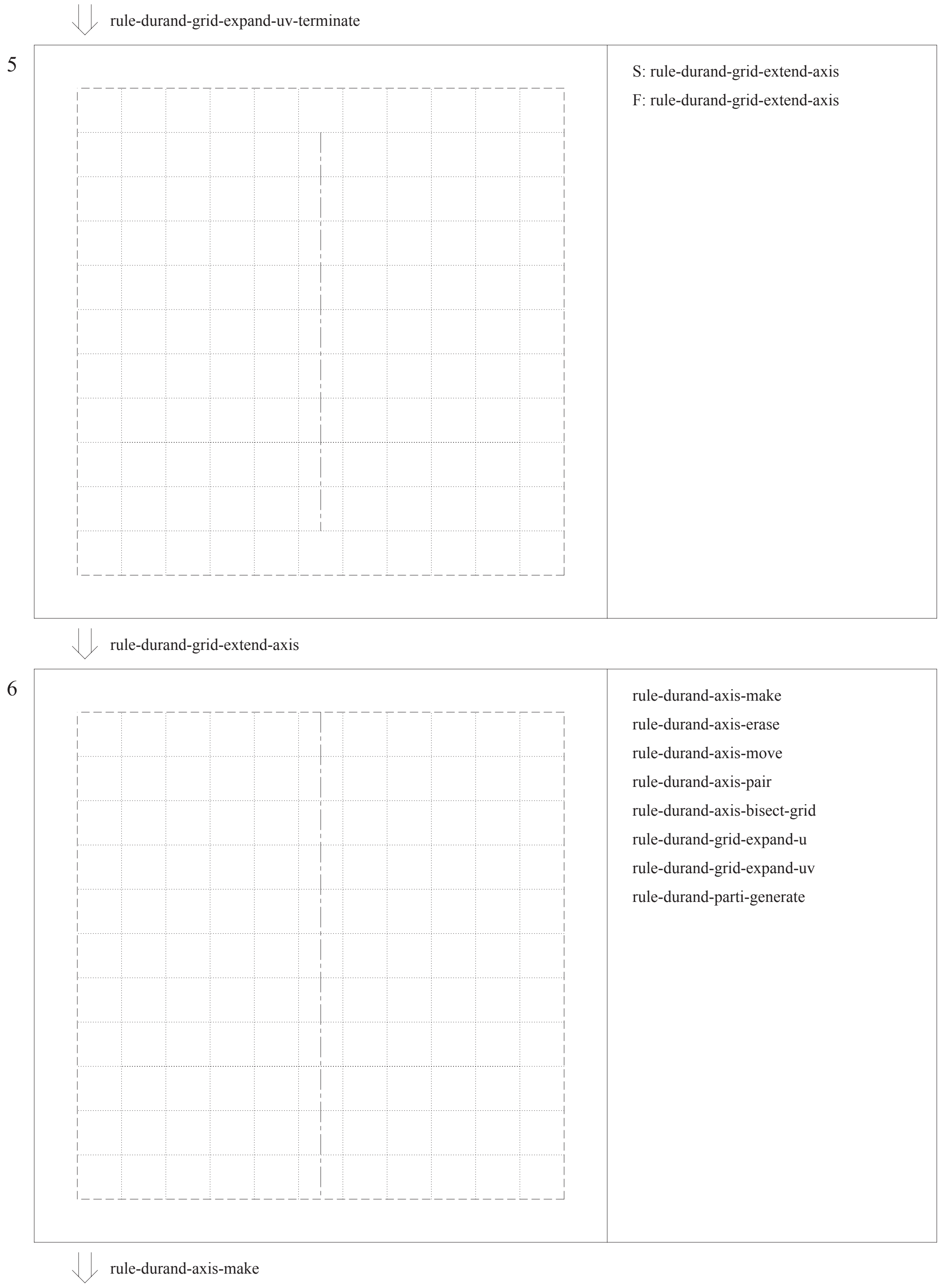
4

S: rule-durand-grid-expand-uv-terminate

F: rule-durand-grid-expand-uv-terminate

rule-durand-grid-expand-uv-terminate

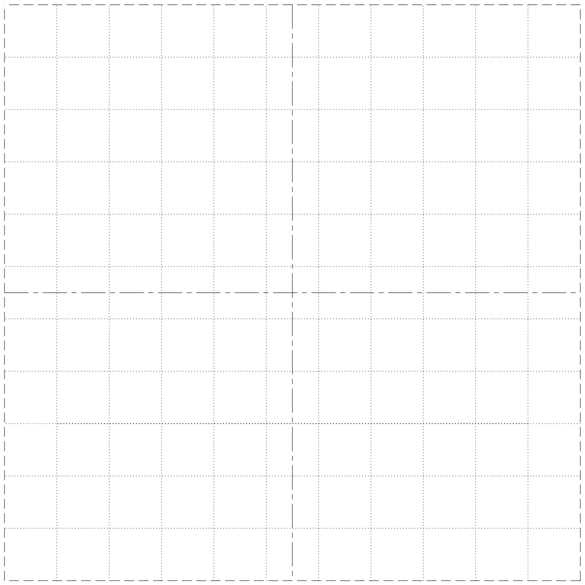
6.31 Derivation of Durand grammar continued.


rule-durand-grid-expand-uv-terminate
5
S: rule-durand-grid-extend-axis
F: rule-durand-grid-extend-axis
rule-durand-grid-extend-axis
6
rule-durand-axis-make
rule-durand-axis-erase
rule-durand-axis-move
rule-durand-axis-pair
rule-durand-axis-bisect-grid
rule-durand-grid-expand-u
rule-durand-grid-expand-uv
rule-durand-parti-generate
rule-durand-axis-make

6.31 Derivation of Durand grammar continued.

⇓ rule-durand-axis-make

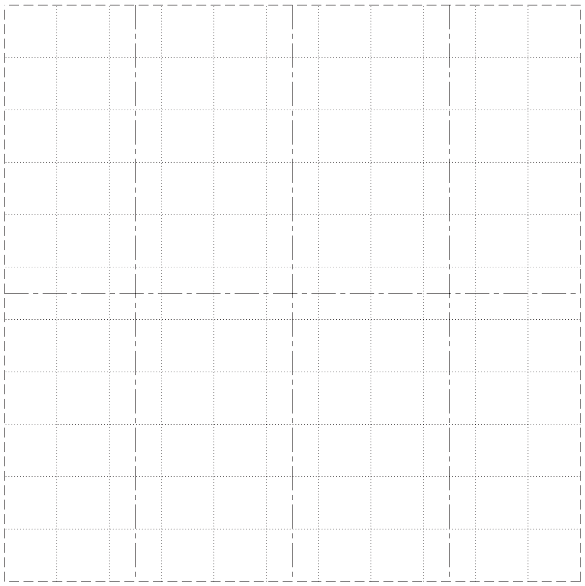
7



rule-durand-axis-make
rule-durand-axis-erase
rule-durand-axis-move
rule-durand-axis-pair
rule-durand-axis-bisect-grid
rule-durand-grid-expand-u
rule-durand-grid-expand-uv
rule-durand-parti-generate

⇓ rule-durand-axis-pair

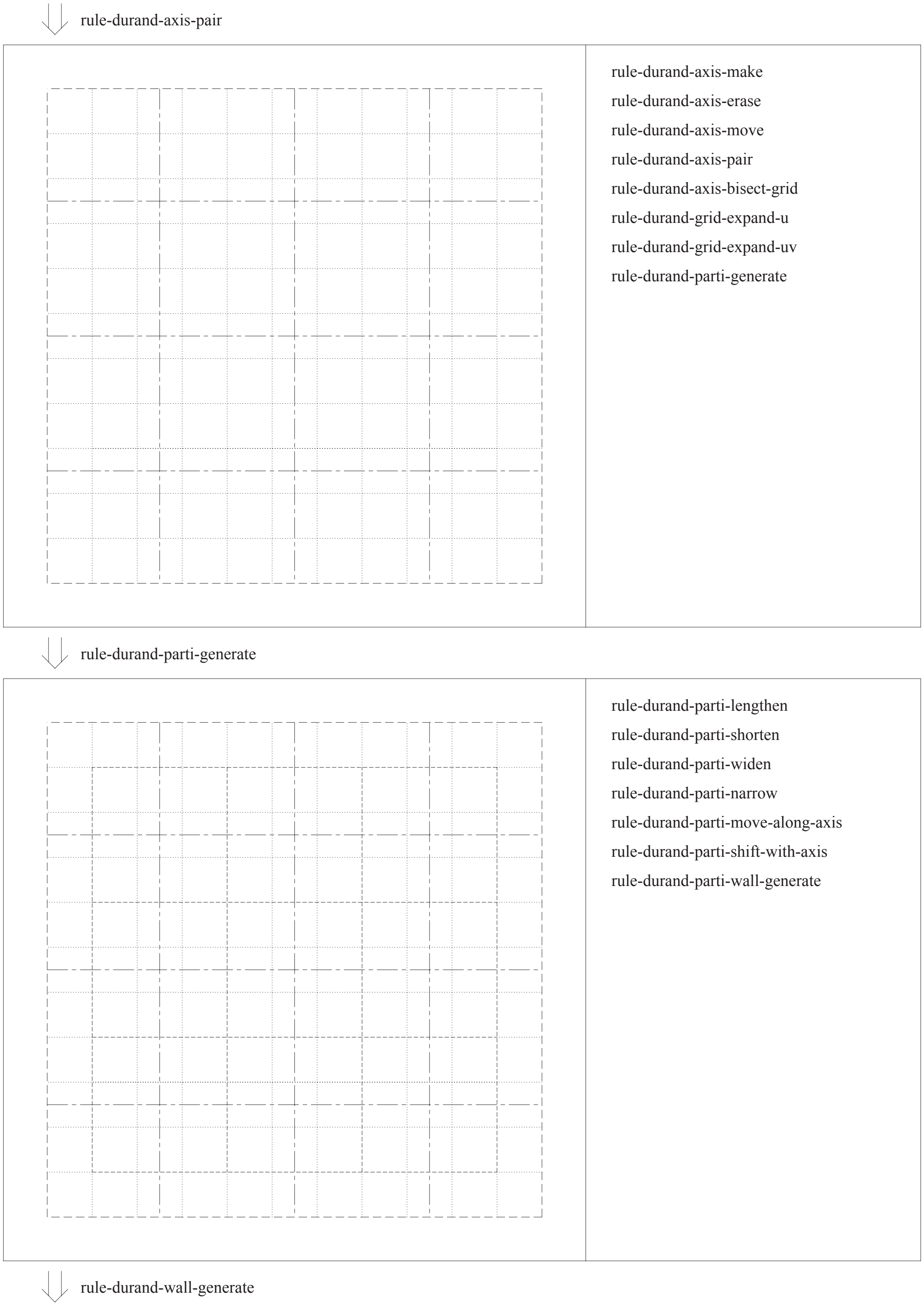
8



rule-durand-axis-make
rule-durand-axis-erase
rule-durand-axis-move
rule-durand-axis-pair
rule-durand-axis-bisect-grid
rule-durand-grid-expand-u
rule-durand-grid-expand-uv
rule-durand-parti-generate

⇓ rule-durand-axis-pair

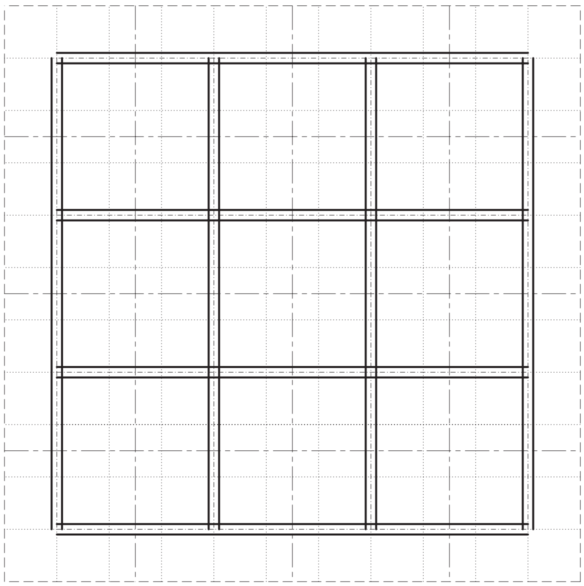
6.31 Derivation of Durand grammar continued.


rule-durand-axis-pair
9
rule-durand-axis-make
rule-durand-axis-erase
rule-durand-axis-move
rule-durand-axis-pair
rule-durand-axis-bisect-grid
rule-durand-grid-expand-u
rule-durand-grid-expand-uv
rule-durand-parti-generate
rule-durand-parti-generate
10
rule-durand-parti-lengthen
rule-durand-parti-shorten
rule-durand-parti-widen
rule-durand-parti-narrow
rule-durand-parti-move-along-axis
rule-durand-parti-shift-with-axis
rule-durand-parti-wall-generate
rule-durand-wall-generate

6.31 Derivation of Durand grammar continued.

rule-durand-wall-generate

11

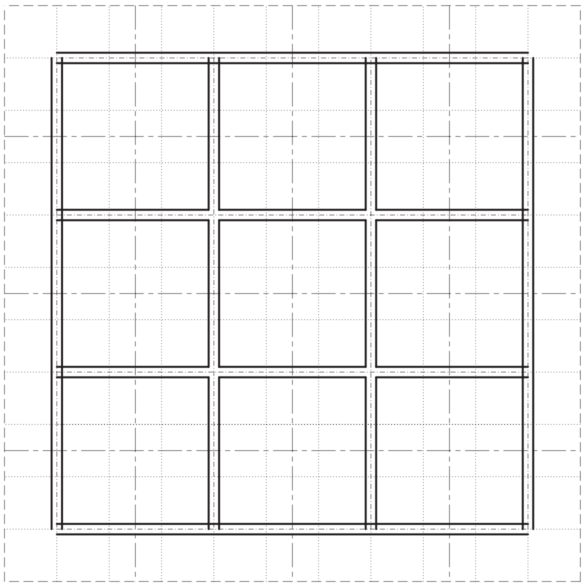


S: rule-durand-wall-trim-+shape

F: -

rule-durand-wall-trim-+shape

12



S: rule-durand-wall-trim-Tshape

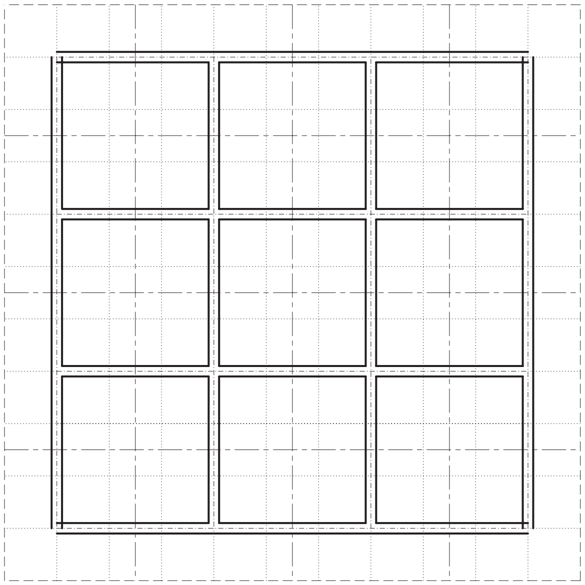
F: rule-durand-wall-trim-Tshape

rule-durand-wall-trim-Tshape

6.31 Derivation of Durand grammar continued.

rule-durand-wall-trim-Tshape

13

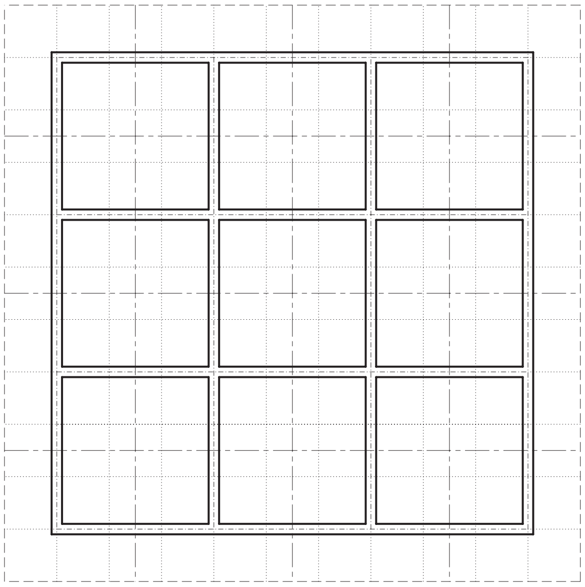


S: rule-durand-wall-trim-Lshape

F: rule-durand-wall-trim-Lshape

rule-durand-wall-trim-Lshape

14



rule-durand-wall-erase

rule-durand-wall-column

rule-durand-room-divide

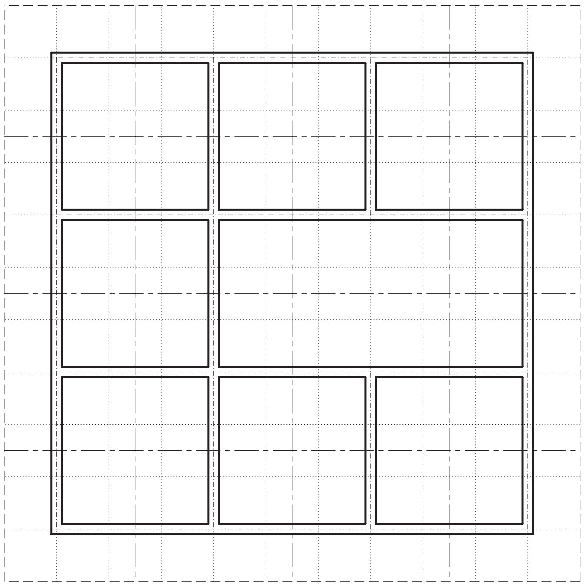
rule-durand-wall-complete

rule-durand-wall-erase

6.31 Derivation of Durand grammar continued.

rule-durand-wall-erase

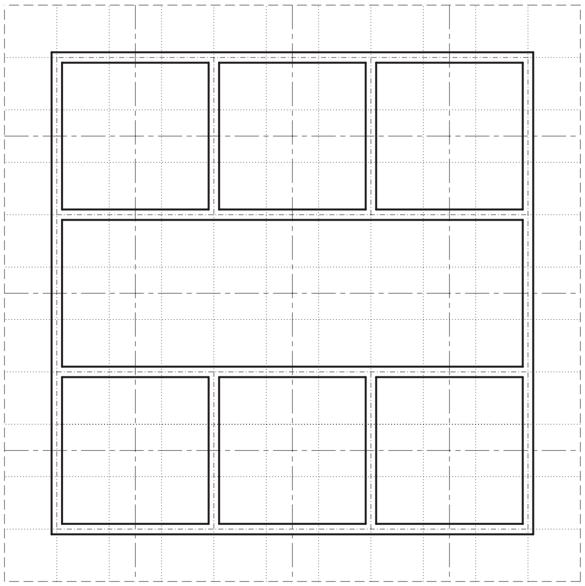
15



rule-durand-wall-erase
rule-durand-wall-column
rule-durand-room-divide
rule-durand-wall-complete

rule-durand-wall-erase

16

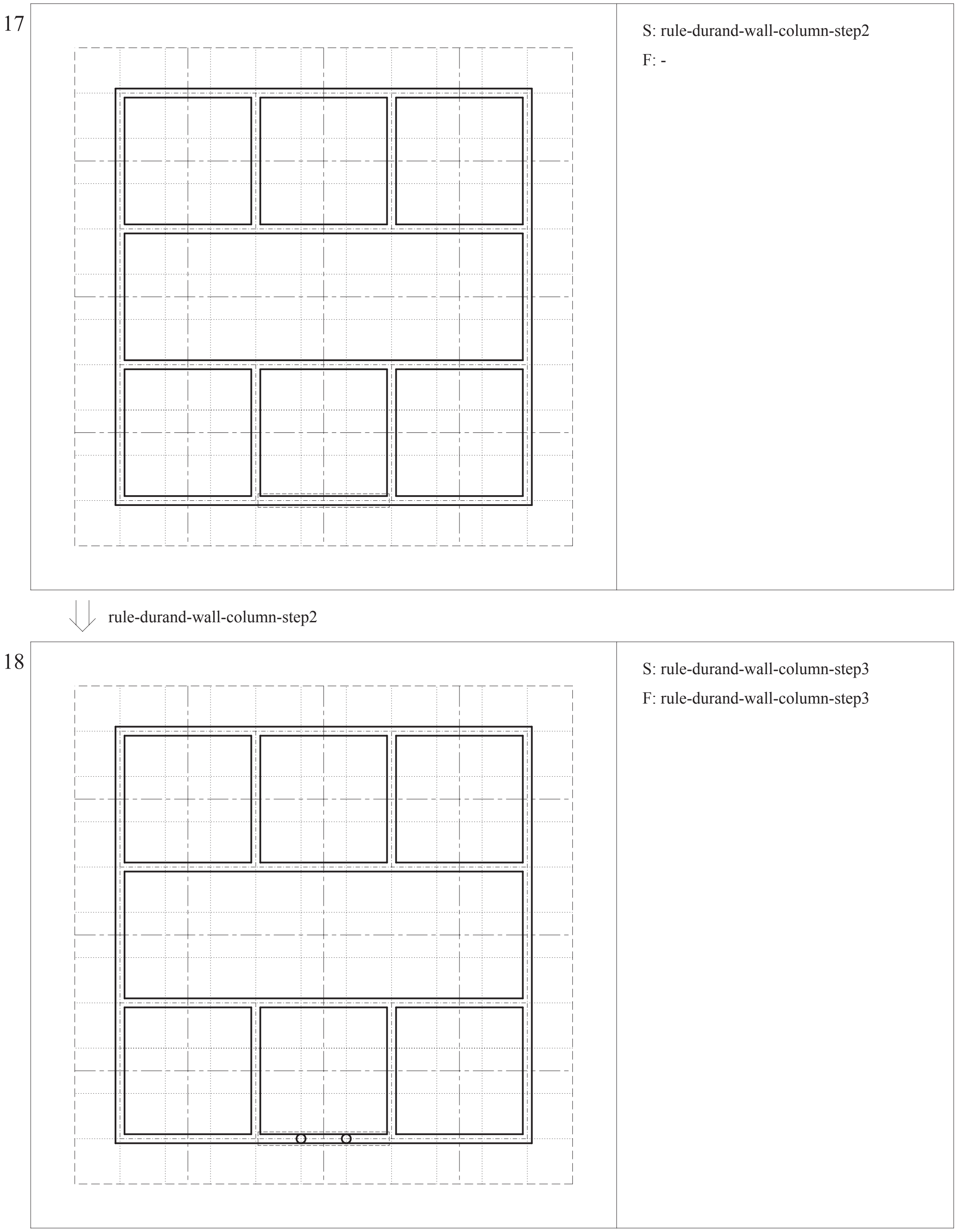


rule-durand-wall-erase
rule-durand-wall-column
rule-durand-room-divide
rule-durand-wall-complete

rule-durand-wall-column

6.31 Derivation of Durand grammar continued.

rule-durand-wall-column



17

S: rule-durand-wall-column-step2

F: -

rule-durand-wall-column-step2

18

S: rule-durand-wall-column-step3

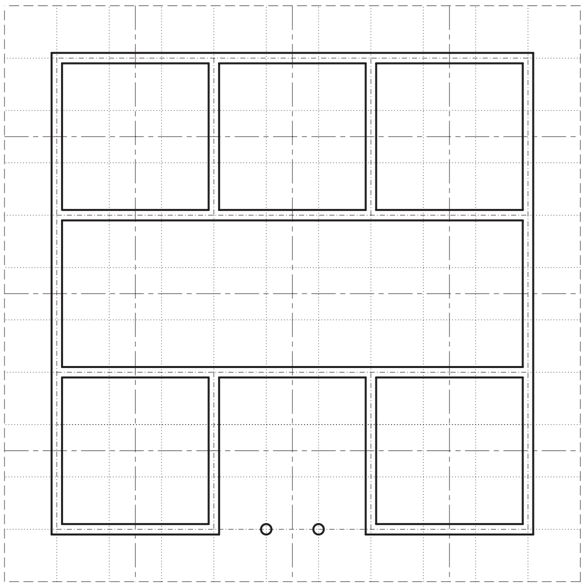
F: rule-durand-wall-column-step3

rule-durand-wall-column-step3

6.31 Derivation of Durand grammar continued.

↓ rule-durand-wall-column-step3

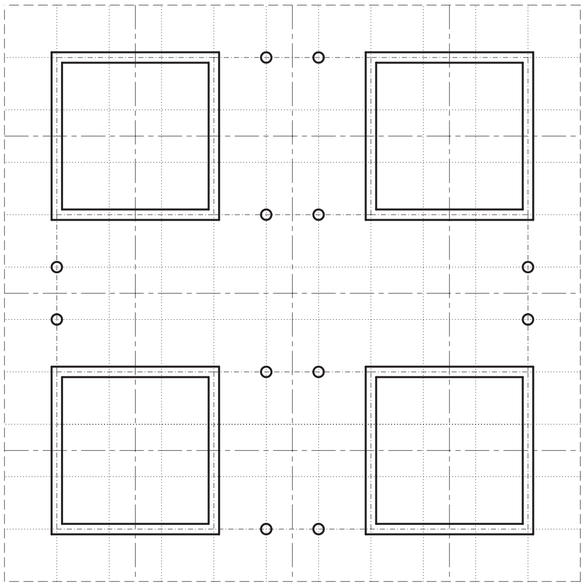
19



rule-durand-wall-erase
rule-durand-wall-column
rule-durand-room-divide
rule-durand-wall-complete

↓ 5x [rule-durand-wall-column, rule-durand-wall-column-step2, rule-durand-wall-column-step3]

20



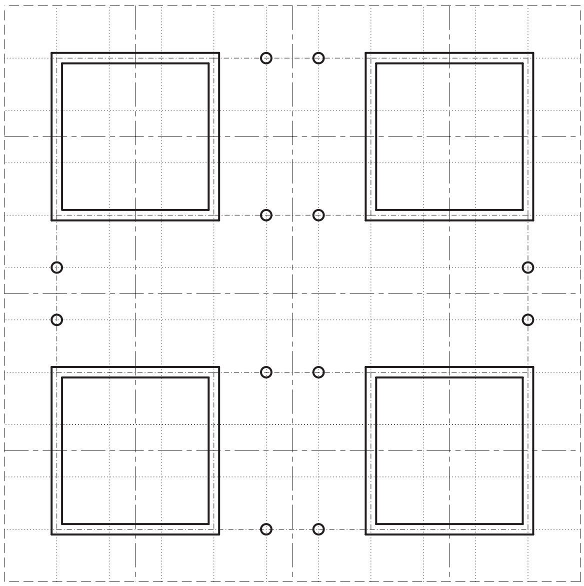
rule-durand-wall-erase
rule-durand-wall-column
rule-durand-room-divide
rule-durand-wall-complete

↓ rule-durand-wall-complete

6.31 Derivation of Durand grammar continued.

rule-durand-wall-complete

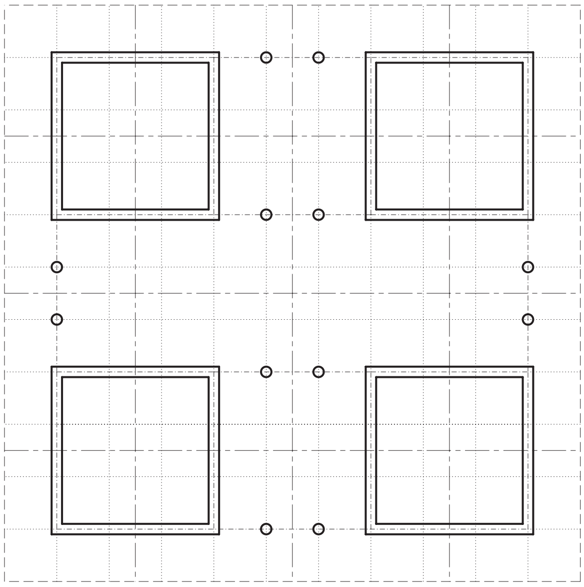
21



rule-durand-room-all-columns
rule-durand-room-corner-columns
rule-durand-room-erase-columns
rule-durand-room-complete

rule-durand-room-complete

22



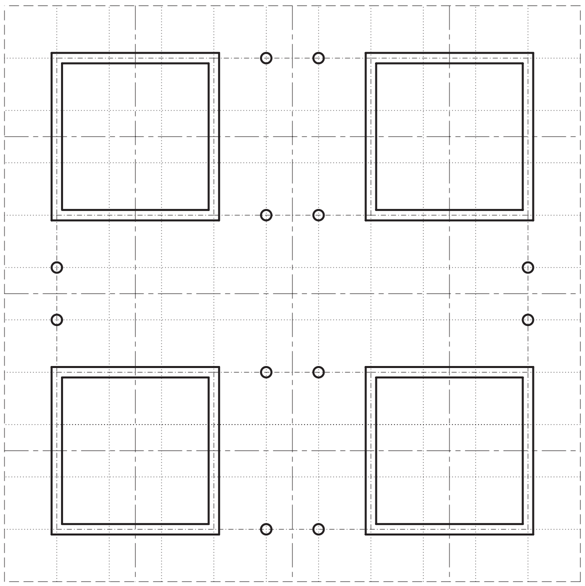
rule-durand-room-indent
rule-durand-room-unindent
rule-durand-room-indent-complete

rule-durand-room-indent-complete

6.31 Derivation of Durand grammar continued.

rule-durand-room-indent-complete

23



rule-durand-axis-make

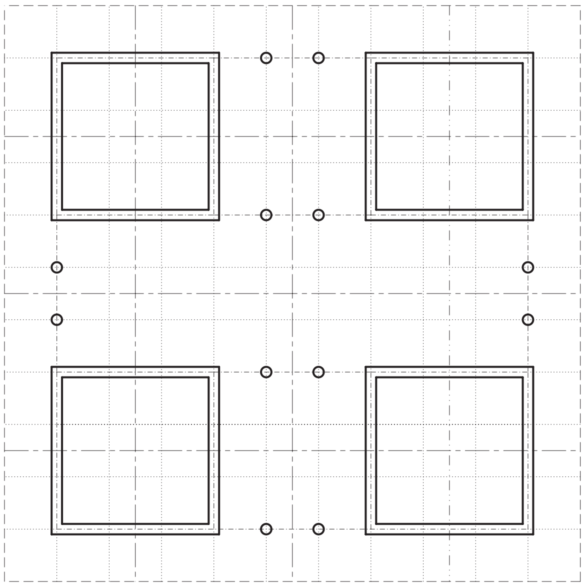
rule-durand-opening-one-axis

rule-durand-opening-all-axis

rule-durand-opening-complete

rule-durand-opening-all-axis

24



S: rule-durand-opening-all-axis-step2

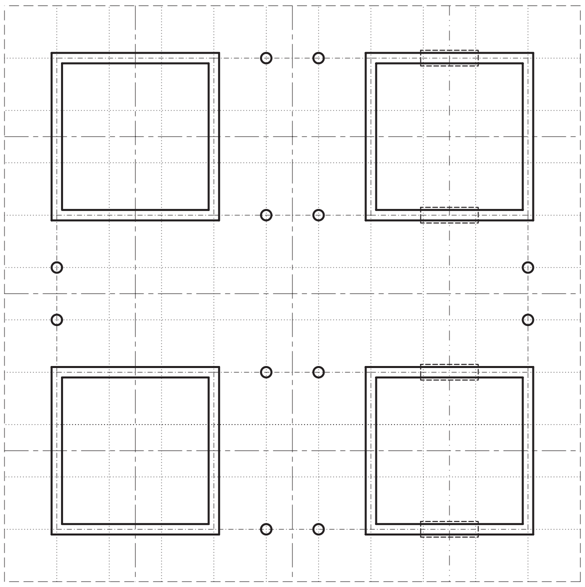
F: -

rule-durand-opening-all-axis-step2

6.31 Derivation of Durand grammar continued.

↓ rule-durand-opening-all-axis-step2

25

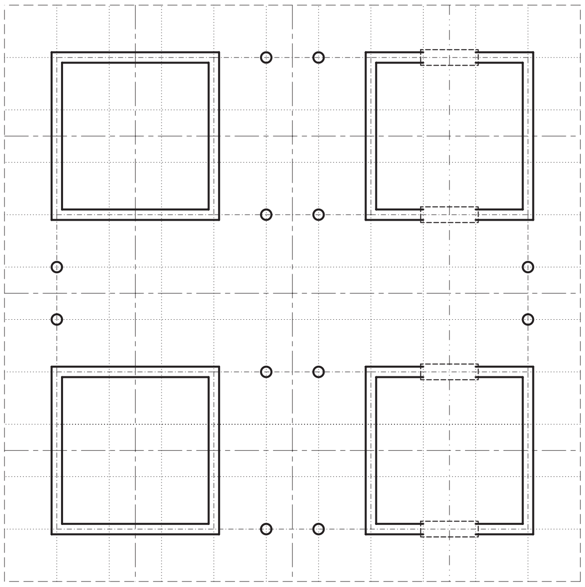


S: rule-durand-opening-all-axis-step3

F: rule-durand-opening-all-axis-step3

↓ rule-durand-opening-all-axis-step3

26



S: rule-durand-opening-all-axis-step4

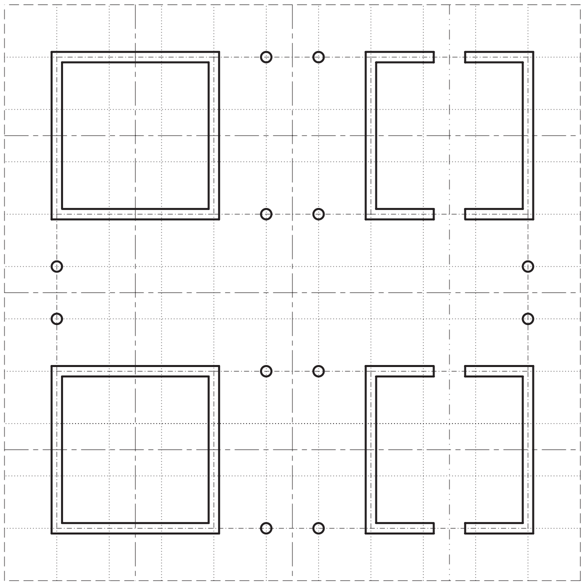
F: rule-durand-opening-all-axis-step4

↓ rule-durand-opening-all-axis-step4

6.31 Derivation of Durand grammar continued.

rule-durand-opening-all-axis-4

27

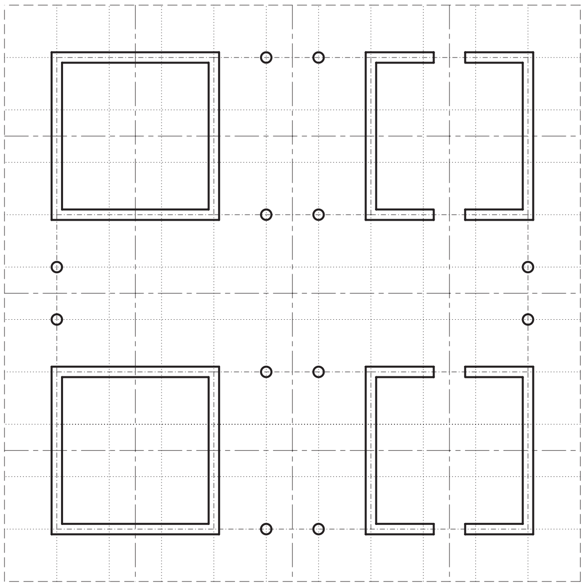


S: rule-durand-opening-all-axis-step5

F: rule-durand-opening-all-axis-step5

rule-durand-opening-all-axis-step5

28



rule-durand-axis-make

rule-durand-opening-one-axis

rule-durand-opening-all-axis

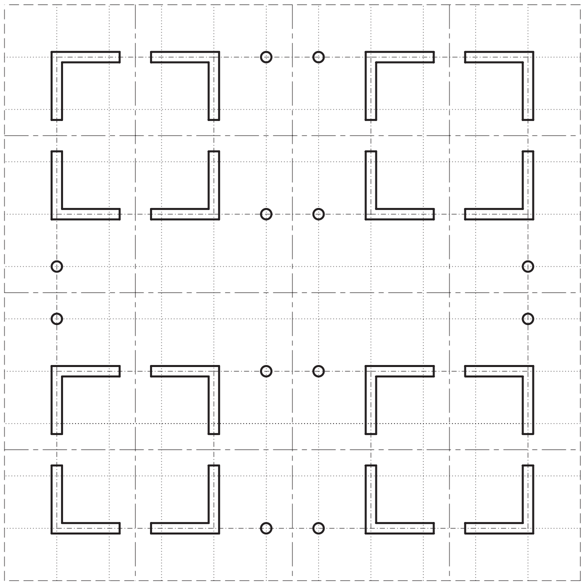
rule-durand-opening-complete

3x [rule-durand-opening-all-axis, -axis-step2, -axis-step3, -axis-step4, -axis-step5]

6.31 Derivation of Durand grammar continued.

↓ 3x [rule-durand-opening-all-axis, -axis-step2, -axis-step3, -axis-step4, -axis-step5]

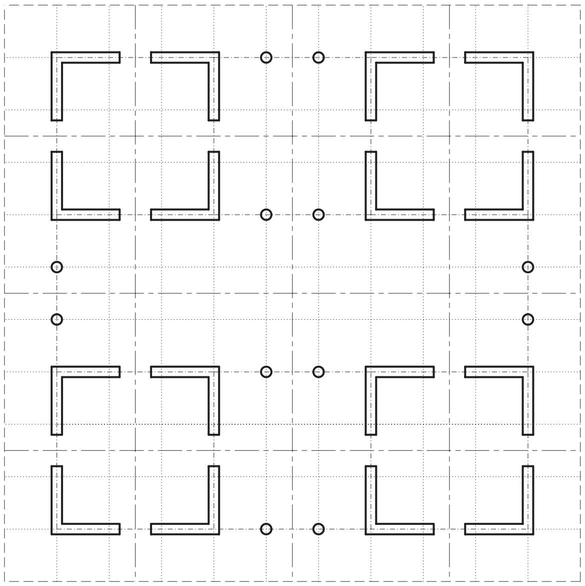
29



rule-durand-axis-make
rule-durand-opening-one-axis
rule-durand-opening-all-axis
rule-durand-opening-complete

↓ rule-durand-opening-complete

30



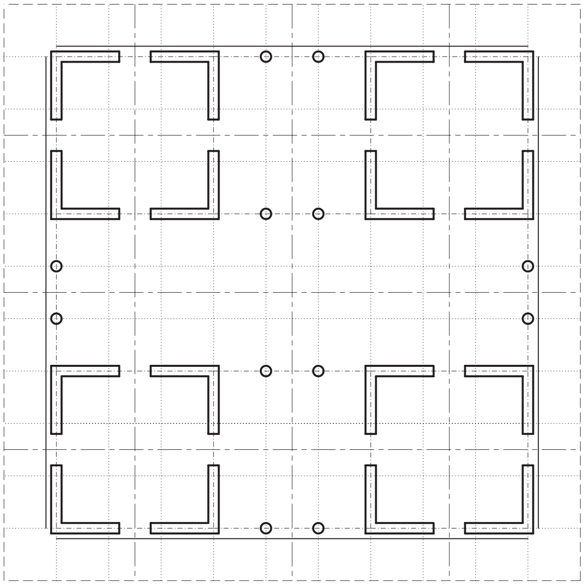
rule-durand-grid-expand-u
rule-durand-grid-expand-uv
rule-durand-platform-make-one
rule-durand-platform-make-ring
rule-durand-platform-complete

↓ rule-durand-platform-make-ring

6.31 Derivation of Durand grammar continued.

rule-durand-platform-make-ring

31

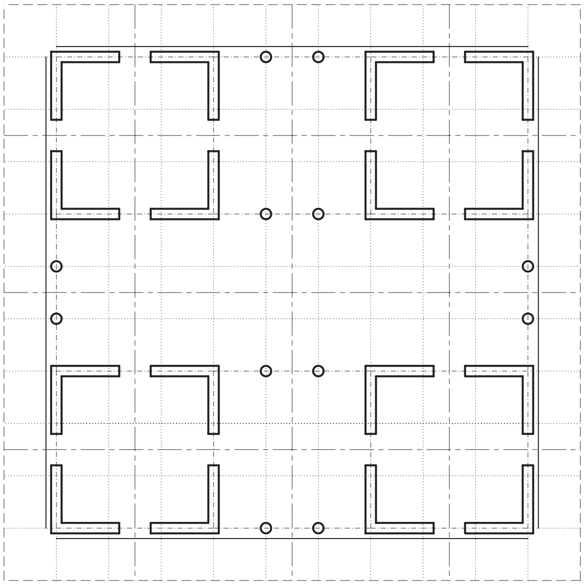


S: rule-durand-platform-make-ring-step2

F: -

rule-durand-platform-make-ring-step2

32

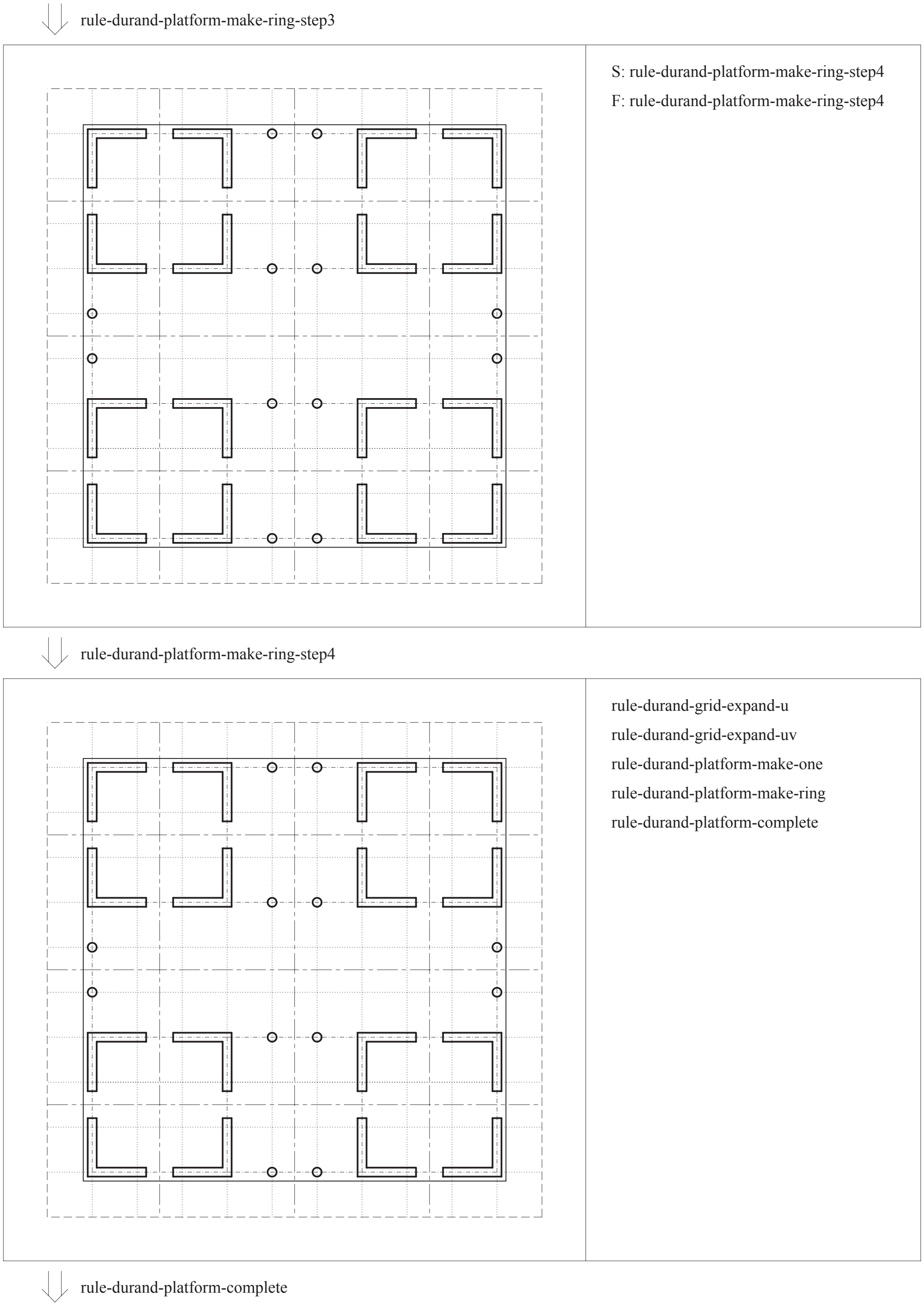


S: rule-durand-platform-make-ring-step3

F: rule-durand-platform-make-ring-step3

rule-durand-platform-make-ring-step3

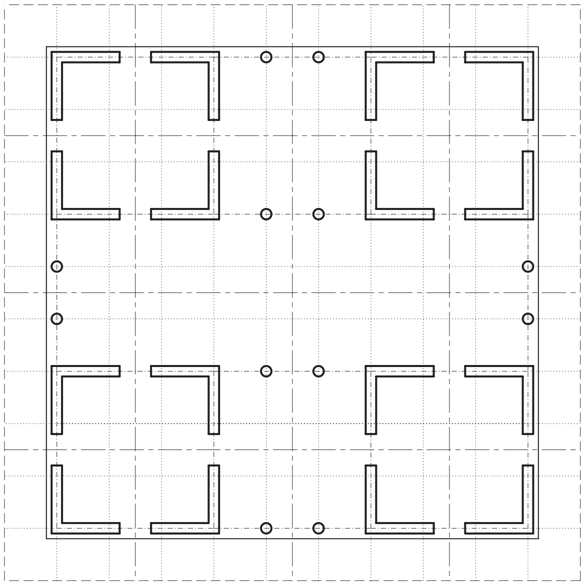
6.31 Derivation of Durand grammar continued.


rule-durand-platform-make-ring-step3
33
S: rule-durand-platform-make-ring-step4
F: rule-durand-platform-make-ring-step4
rule-durand-platform-make-ring-step4
34
rule-durand-grid-expand-u
rule-durand-grid-expand-uv
rule-durand-platform-make-one
rule-durand-platform-make-ring
rule-durand-platform-complete
rule-durand-platform-complete

6.31 Derivation of Durand grammar continued.

⇓ rule-durand-platform-complete

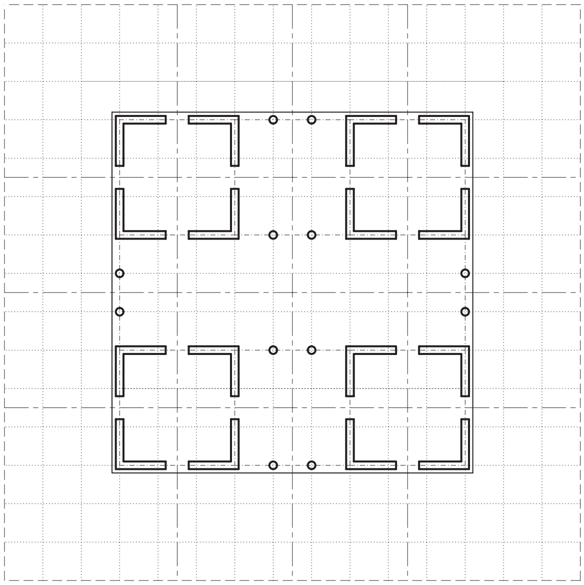
35



rule-durand-grid-expand-u
rule-durand-grid-expand-uv
rule-durand-exterior-stair-platform
rule-durand-exterior-stair-wall
rule-durand-exterior-stair-extend
rule-durand-end-grammar

⇓ 2x [rule-durand-grid-expand-uv, -expand-uv-fill, -expand-uv-terminate, -extend-axis]

36



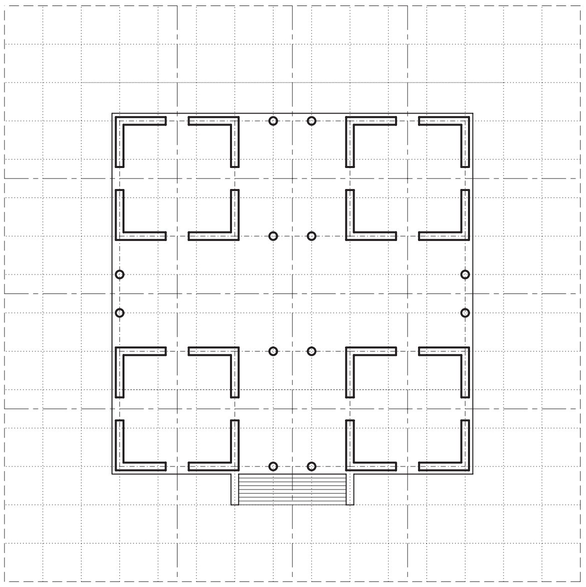
rule-durand-grid-expand-u
rule-durand-grid-expand-uv
rule-durand-exterior-stair-platform
rule-durand-exterior-stair-wall
rule-durand-exterior-stair-extend
rule-durand-end-grammar

⇓ rule-durand-exterior-stair-platform

6.31 Derivation of Durand grammar continued.

rule-durand-exterior-stair-platform

37

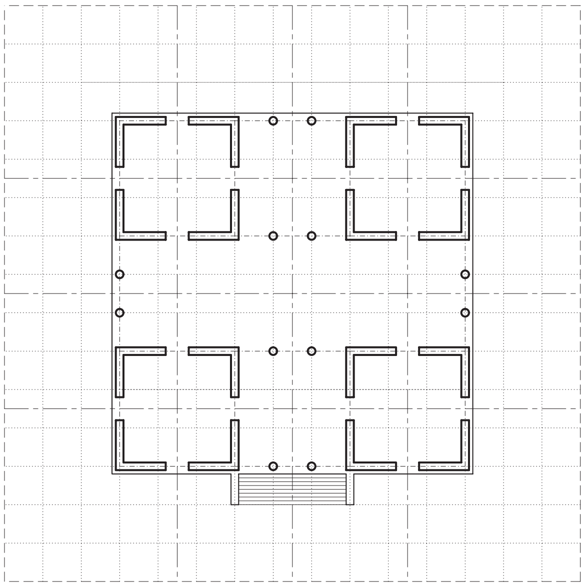


S: rule-durand-exterior-stair-platform-fix

F: -

rule-durand-exterior-stair-platform-fix

38



rule-durand-grid-expand-u

rule-durand-grid-expand-uv

rule-durand-exterior-stair-platform

rule-durand-exterior-stair-wall

rule-durand-exterior-stair-extend

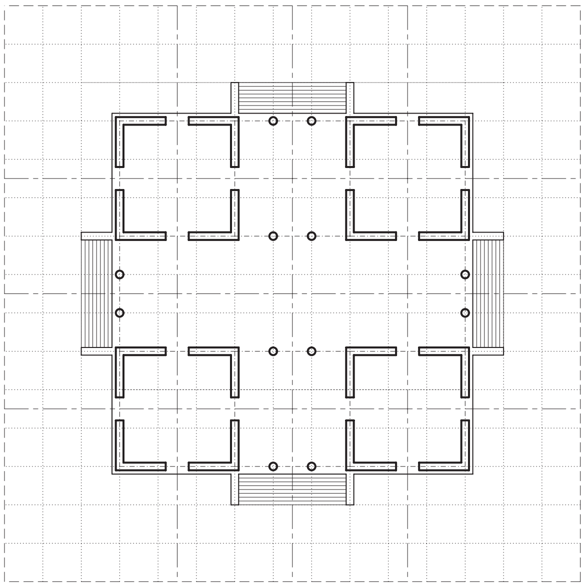
rule-durand-end-grammar

3x [rule-durand-exterior-stair-platform, rule-durand-exterior-stair-platform-fix]

6.31 Derivation of Durand grammar continued.

⇓ 3x [rule-durand-exterior-stair-platform, rule-durand-exterior-stair-platform-fix]

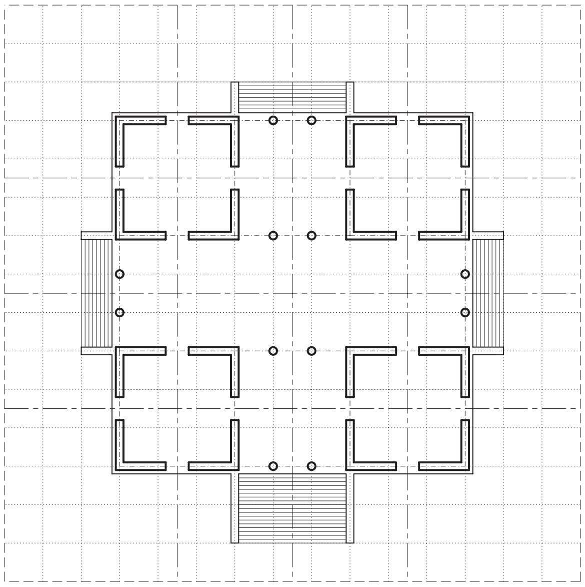
39



rule-durand-grid-expand-u
rule-durand-grid-expand-uv
rule-durand-exterior-stair-platform
rule-durand-exterior-stair-wall
rule-durand-exterior-stair-extend
rule-durand-end-grammar

⇓ rule-durand-exterior-stair-extend

40



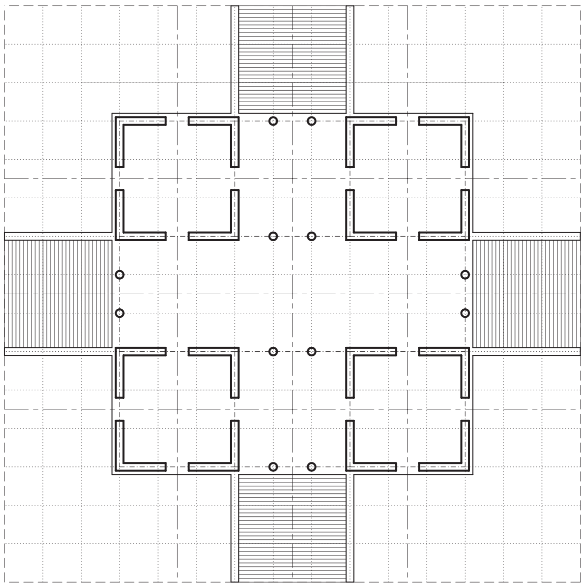
rule-durand-grid-expand-u
rule-durand-grid-expand-uv
rule-durand-exterior-stair-platform
rule-durand-exterior-stair-wall
rule-durand-exterior-stair-extend
rule-durand-end-grammar

⇓ 7x [rule-durand-exterior-stair-extend]

6.31 Derivation of Durand grammar continued.

7x [rule-durand-exterior-stair-extend]

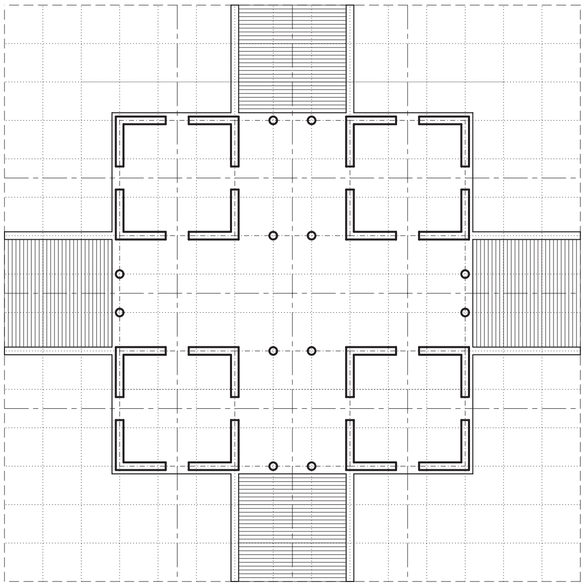
41



rule-durand-grid-expand-u
rule-durand-grid-expand-uv
rule-durand-exterior-stair-platform
rule-durand-exterior-stair-wall
rule-durand-exterior-stair-extend
rule-durand-end-grammar

rule-durand-end-grammar

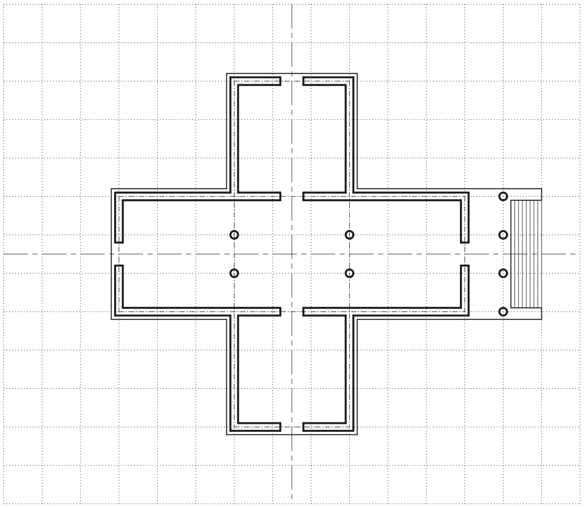
42



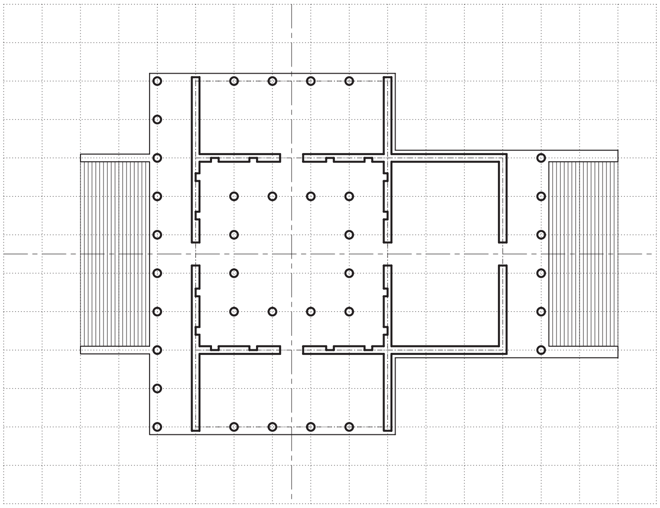
6.31 Derivation of Durand grammar continued.

6.4 Examples of the Durand Grammar

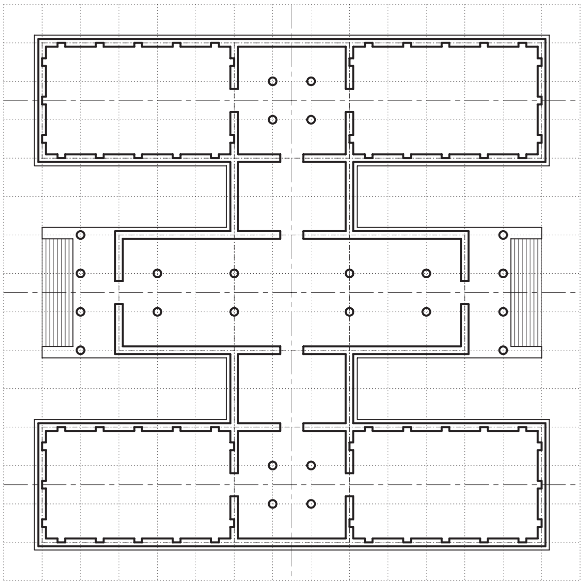
This section will show some sample two-dimensional floor plans generated from the Durand grammar. The grammar emulates the design process but does not add all the restrictions that Durand implies in his drawings such as the adherence to symmetry. This allows the grammar to produce designs that are not in *Précis of the Lectures on Architecture*. The decision to have a symmetrical building is left up to the user. Figures 6.32 – 6.37 shows some sample floor plan designs that can be made using the grammar. The floor plans range from simple symmetrical plans to complex asymmetrical multi-room drawings.



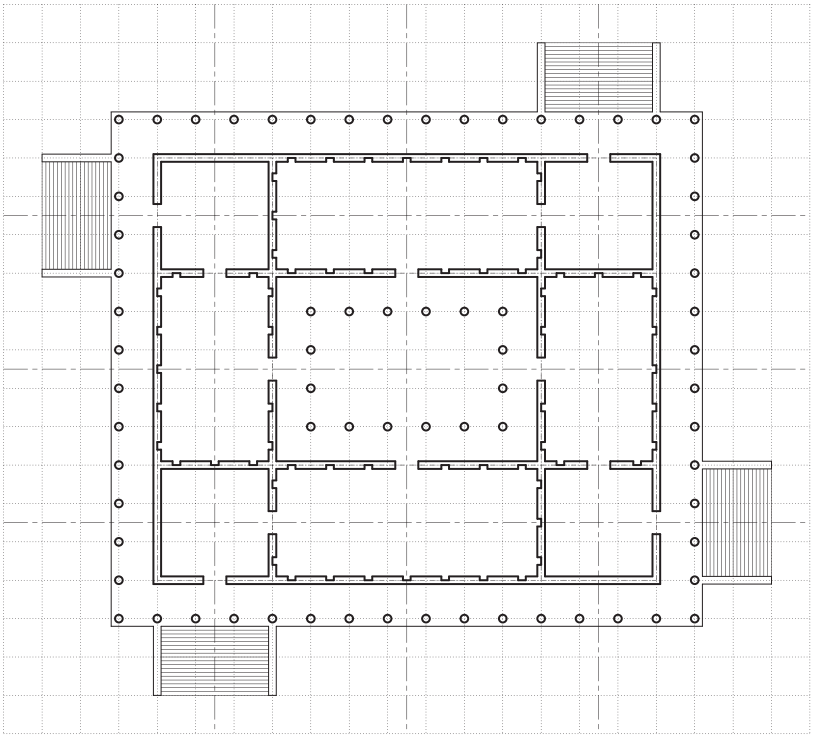
6.32 A sample floor plan using the Durand grammar.



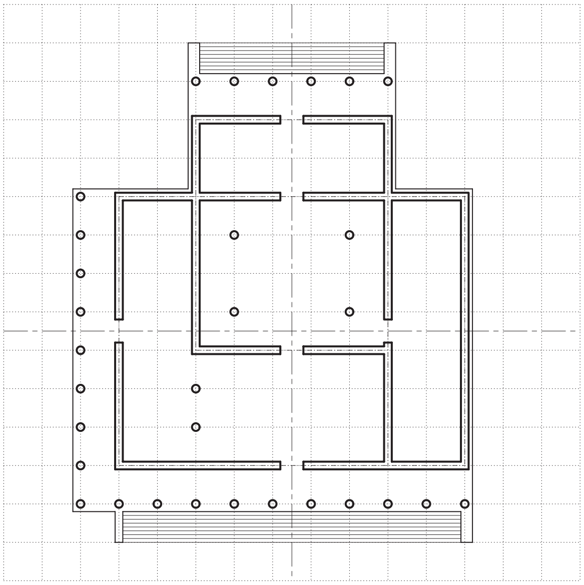
6.33 A sample floor plan using the Durand grammar.



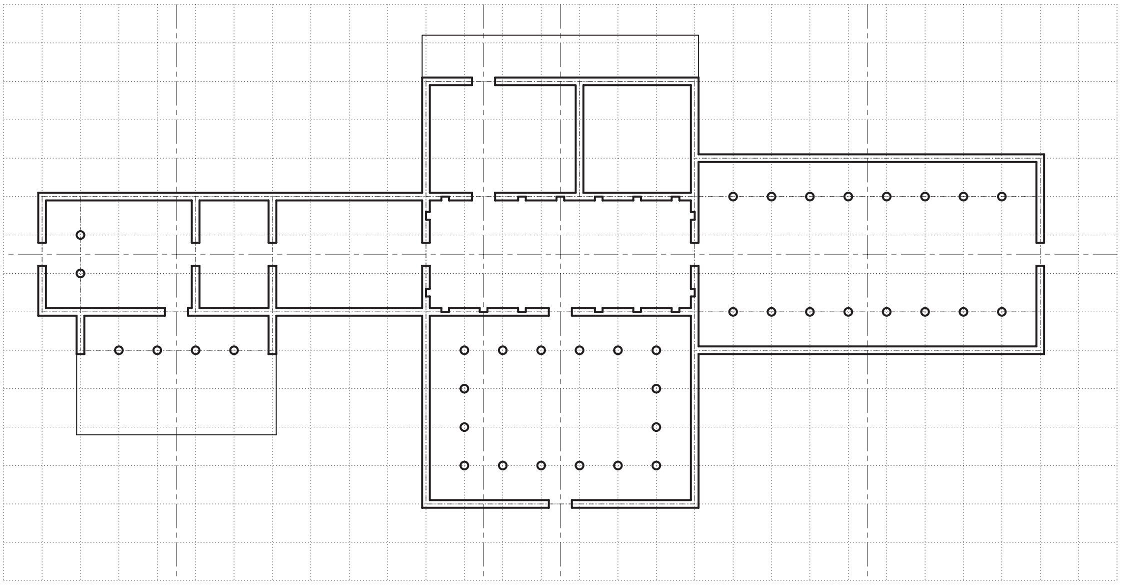
6.34 A sample floor plan using the Durand grammar.



6.35 A sample floor plan using the Durand grammar.



6.36 A sample floor plan using the Durand grammar.

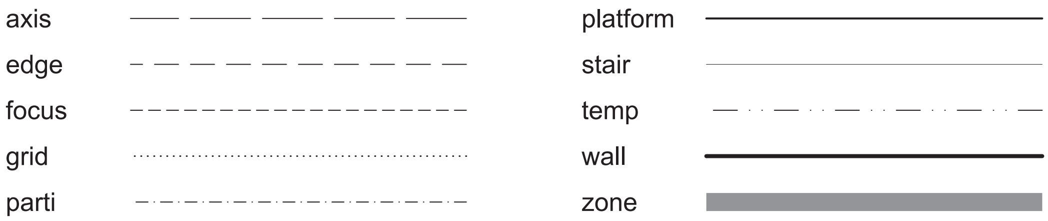


6.37 A sample floor plan using the Durand grammar.

6.5 Rules of the Durand grammar

There are a total of 74 rules in the Durand grammar (figure 6.39). The rules are listed in this section in alphabetical order (figure 6.40). Each rule is divided into two parts. The left half of the frame gives the name of the rule along with the geometric representation of the schema. The pertinent parameters are noted by dimension lines. The right half of the frame lists the components of a rule. It begins with a general description of the rule and includes the parameters g() and the components for the meta-language descriptors. For all the rules, the transformation t() has been omitted since there are no transformation restrictions unless otherwise noted. The variable "gu" used in the rules stores the grid unit width and height in the drawing and is set at the beginning of the grammar.

The linetypes used in the rules are the same as those used in the derivation of the Durand grammar (figure 6.30). The different linetypes correspond to the different labeled lines used in the grammar. The only change is the addition of a gray shaded area to represent the zone descriptor (figure 6.38).



6.38 The linetype chart for each of the different labeled lines in the Durand grammar.

rule-durand-axis-bisect-grid
rule-durand-axis-erase
rule-durand-axis-make
rule-durand-axis-move
rule-durand-axis-pair

rule-durand-end-grammar

rule-durand-exterior-stair-erase
rule-durand-exterior-stair-erase-step2
rule-durand-exterior-stair-erase-terminate
rule-durand-exterior-stair-extend
rule-durand-exterior-stair-platform
rule-durand-exterior-stair-platform-fix
rule-durand-exterior-stair-wall

rule-durand-grid-expand-u
rule-durand-grid-expand-u-fill
rule-durand-grid-expand-u-terminate
rule-durand-grid-expand-uv
rule-durand-grid-expand-uv-fill
rule-durand-grid-expand-uv-terminate
rule-durand-grid-extend-axis

rule-durand-opening-all-axis
rule-durand-opening-all-axis-step2
rule-durand-opening-all-axis-step3
rule-durand-opening-all-axis-step4
rule-durand-opening-all-axis-step5
rule-durand-opening-complete
rule-durand-opening-one-axis
rule-durand-opening-one-axis-step2
rule-durand-opening-one-axis-step3

rule-durand-parti-fix1
rule-durand-parti-fix2
rule-durand-parti-generate
rule-durand-parti-lengthen
rule-durand-parti-move-along-axis
rule-durand-parti-narrow
rule-durand-parti-shift-with-axis
rule-durand-parti-shorten
rule-durand-parti-widen

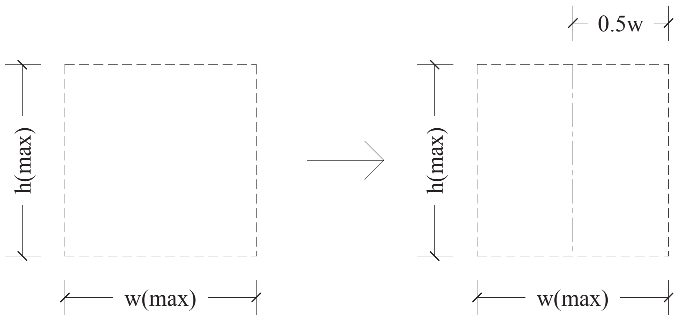
rule-durand-platform-complete
rule-durand-platform-complete-with-columns
rule-durand-platform-erase-all
rule-durand-platform-erase-one
rule-durand-platform-extend-one
rule-durand-platform-make-one
rule-durand-platform-make-ring
rule-durand-platform-make-ring-step2
rule-durand-platform-make-ring-step3
rule-durand-platform-make-ring-step4
rule-durand-platform-retract-one

rule-durand-room-all-columns
rule-durand-room-all-columns-step2
rule-durand-room-columns-complete
rule-durand-room-corner-columns
rule-durand-room-corner-columns-step2
rule-durand-room-divide
rule-durand-room-divide-step2
rule-durand-room-erase-columns
rule-durand-room-erase-columns-step2
rule-durand-room-indent
rule-durand-room-indent-complete
rule-durand-room-indent-step2
rule-durand-room-remove-bay
rule-durand-room-terminate
rule-durand-room-unindent
rule-durand-room-unindent-step2

rule-durand-wall-column
rule-durand-wall-column-step2
rule-durand-wall-column-step3
rule-durand-wall-complete
rule-durand-wall-erase
rule-durand-wall-generate
rule-durand-wall-trim-+shape
rule-durand-wall-trim-Lshape
rule-durand-wall-trim-Tshape

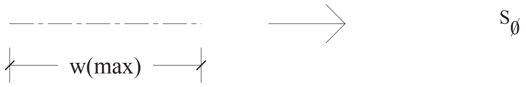
6.39 Alphabetical listing of all 74 rules in the Durand grammar.

rule-durand-axis-bisect-grid



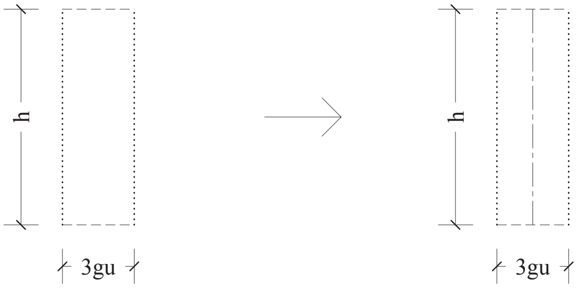
DESCRIPTION:	Rule to make an axis that bisects the entire grid.
CONSTRAINT:	w > 0, h > 0
LABEL-FILTER:	on
APPLY-MODE:	single
SUCCESS:	nil
FAILURE:	nil
RULE-SET:	nil

rule-durand-axis-erase



DESCRIPTION:	Rule to erase one maximal axis line.
CONSTRAINT:	w > 0
LABEL-FILTER:	on
APPLY-MODE:	single
SUCCESS:	nil
FAILURE:	nil
RULE-SET:	nil

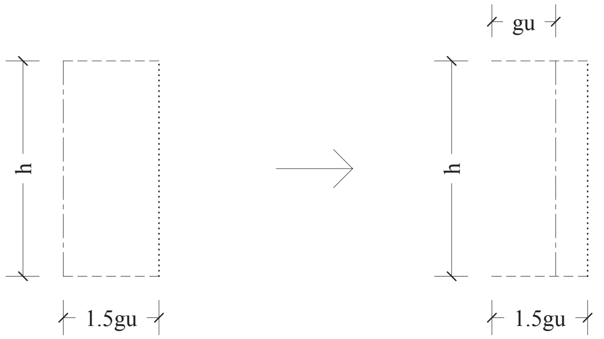
rule-durand-axis-make



DESCRIPTION:	Rule to make a new axis in the grid.
CONSTRAINT:	h > 0
LABEL-FILTER:	on
APPLY-MODE:	single
SUCCESS:	nil
FAILURE:	nil
RULE-SET:	nil

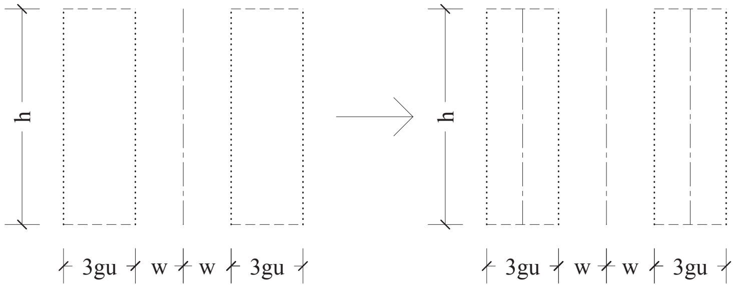
6.40 Rules of the Durand grammar.

rule-durand-axis-move



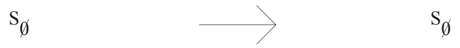
DESCRIPTION:	Rule to move the axis by one gridunit in the positive x direction.
CONSTRAINT:	h > 0
LABEL-FILTER:	on
APPLY-MODE:	single
SUCCESS:	nil
FAILURE:	nil
RULE-SET:	nil

rule-durand-axis-pair



DESCRIPTION:	Rule to create a pair of axis based on a central axis.
CONSTRAINT:	w > 0, h > 0
LABEL-FILTER:	on
APPLY-MODE:	single
SUCCESS:	nil
FAILURE:	nil
RULE-SET:	nil

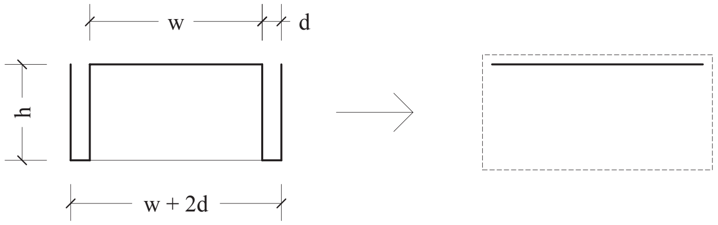
rule-durand-end-grammar



DESCRIPTION:	Rule to end the grammar by having no rules in the rule-set.
CONSTRAINT:	nil
LABEL-FILTER:	nil
APPLY-MODE:	nil
SUCCESS:	nil
FAILURE:	nil
RULE-SET:	set-rule ()

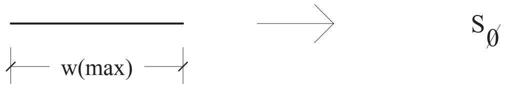
6.40 Rules of the Durand grammar continued.

rule-durand-exterior-stair-erase



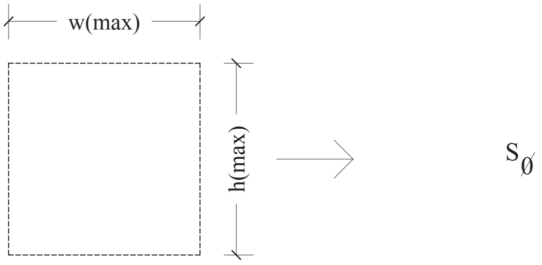
DESCRIPTION:	Rule to erase an existing exterior staircase.
CONSTRAINT:	w > 0, d > 0, h > 0
LABEL-FILTER:	on
APPLY-MODE:	single
SUCCESS:	nil
FAILURE:	nil
RULE-SET:	nil

rule-durand-exterior-stair-erase-step2



DESCRIPTION:	Rule to erase all stair labeled lines.
CONSTRAINT:	w > 0
LABEL-FILTER:	on
APPLY-MODE:	parallel
SUCCESS:	rule-durand-exterior-stair-erase-terminate
FAILURE:	rule-durand-exterior-stair-erase-terminate
RULE-SET:	nil

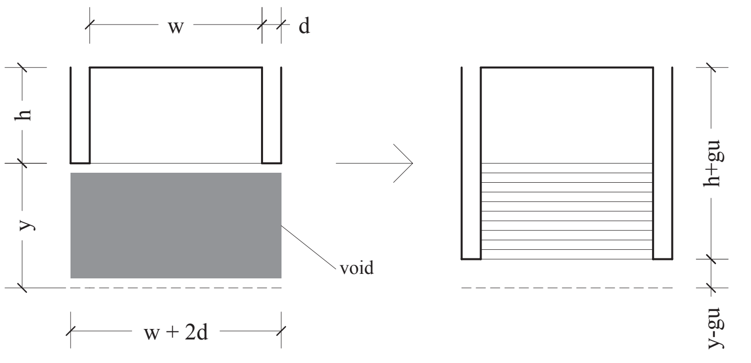
rule-durand-exterior-stair-erase-terminate



DESCRIPTION:	Rule to remove a rectangular focus polygon from the drawing.
CONSTRAINT:	w > 0, h > 0
LABEL-FILTER:	on
APPLY-MODE:	random
SUCCESS:	nil
FAILURE:	nil
RULE-SET:	nil

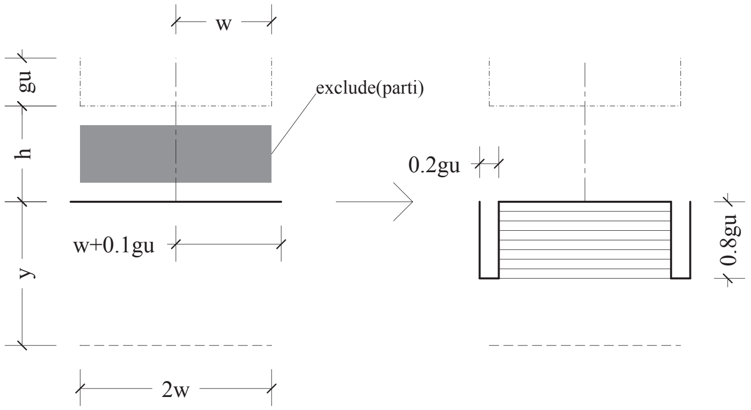
6.40 Rules of the Durand grammar continued.

rule-durand-exterior-stair-extend



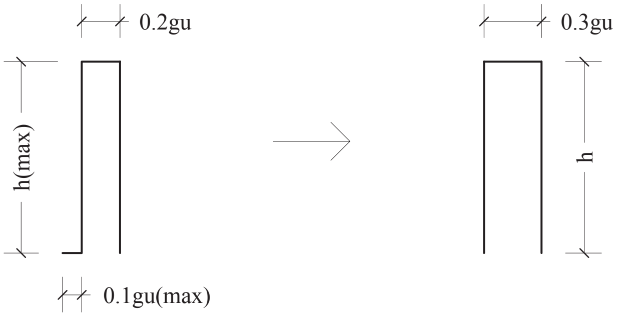
DESCRIPTION:	Rule to extend an existing exterior staircase.
CONSTRAINT:	w > 0, d > 0, h > 0, y > 0.9gu
LABEL-FILTER:	on
APPLY-MODE:	single
SUCCESS:	nil
FAILURE:	nil
RULE-SET:	nil

rule-durand-exterior-stair-platform



DESCRIPTION:	Rule to make an exterior stair on axis against a platform.
CONSTRAINT:	w > 0, h > 0, y > 0.8gu
LABEL-FILTER:	on
APPLY-MODE:	single
SUCCESS:	rule-durand-exterior-stair-platform-fix
FAILURE:	nil
RULE-SET:	nil

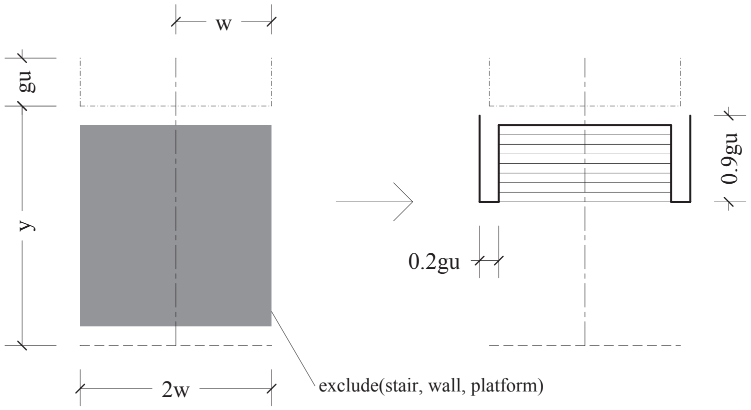
rule-durand-exterior-stair-platform-fix



DESCRIPTION:	Rule to fix a little notch in the platform that occurs when the platform surrounds the building.
CONSTRAINT:	h > 0
LABEL-FILTER:	on
APPLY-MODE:	parallel
SUCCESS:	nil
FAILURE:	nil
RULE-SET:	nil

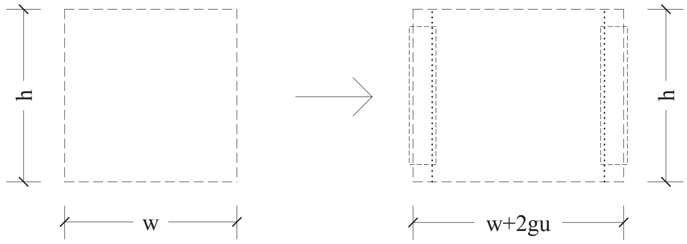
6.40 Rules of the Durand grammar continued.

rule-durand-exterior-stair-wall



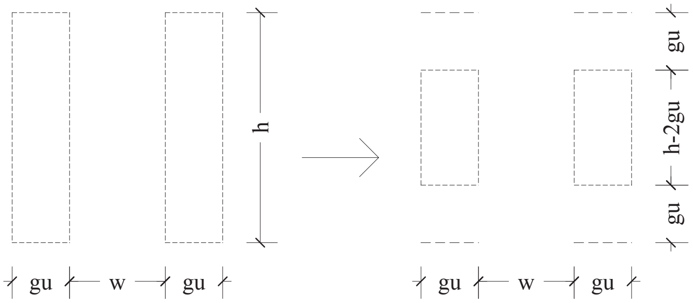
DESCRIPTION:	Rule to make an exterior stair on axis against a wall.
CONSTRAINT:	w > 0, h > 0, y > 0.9gu
LABEL-FILTER:	on
APPLY-MODE:	single
SUCCESS:	nil
FAILURE:	nil
RULE-SET:	nil

rule-durand-grid-expand-u



DESCRIPTION:	Rule to expand the grid in the u direction.
CONSTRAINT:	w > 0, h > 0
LABEL-FILTER:	on
APPLY-MODE:	single
SUCCESS:	rule-durand-grid-expand-u-fill
FAILURE:	nil
RULE-SET:	nil

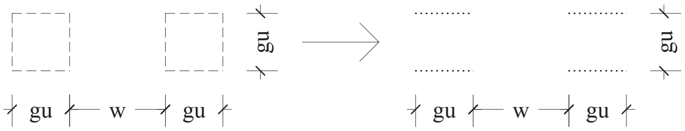
rule-durand-grid-expand-u-fill



DESCRIPTION:	Rule to fill in the expanded grid.
CONSTRAINT:	w > 0, h > 2gu
LABEL-FILTER:	on
APPLY-MODE:	single
SUCCESS:	rule-durand-grid-expand-u-fill
FAILURE:	rule-durand-grid-expand-u-terminate
RULE-SET:	nil

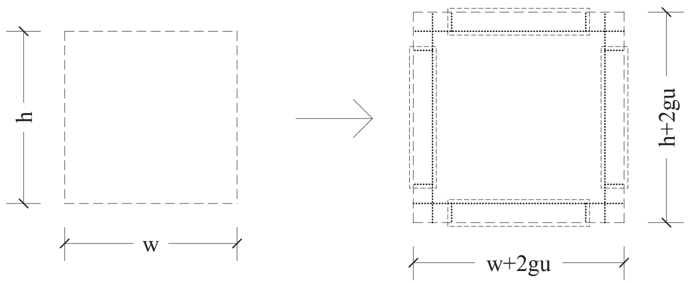
6.40 Rules of the Durand grammar continued.

rule-durand-grid-expand-u-terminate



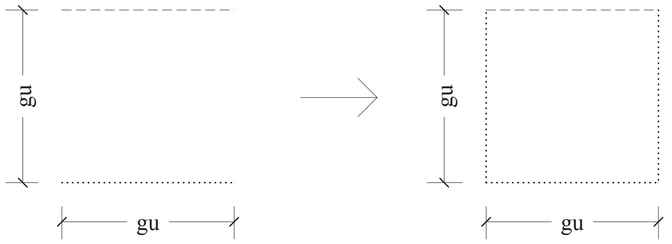
DESCRIPTION:	Rule to fill in the expanded grid.
CONSTRAINT:	w > 0, h > 2gu
LABEL-FILTER:	on
APPLY-MODE:	single
SUCCESS:	rule-durand-grid-expand-u-fill
FAILURE:	rule-durand-grid-expand-u-terminate
RULE-SET:	nil

rule-durand-grid-expand-uv



DESCRIPTION:	Rule to expand the grid in both the u and v direction.
CONSTRAINT:	w > 0, h > 0
LABEL-FILTER:	on
APPLY-MODE:	random
SUCCESS:	rule-durand-grid-expand-uv-fill
FAILURE:	nil
RULE-SET:	nil

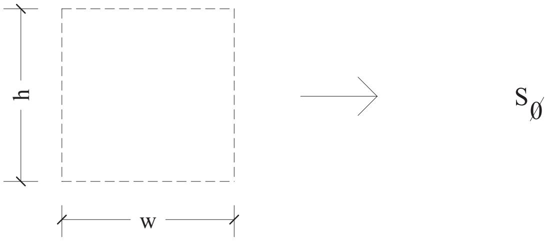
rule-durand-grid-expand-uv-fill



DESCRIPTION:	Rule to fill in the intermediate grid lines.
CONSTRAINT:	nil
LABEL-FILTER:	on
APPLY-MODE:	parallel
SUCCESS:	rule-durand-grid-expand-uv-terminate
FAILURE:	nil
RULE-SET:	nil

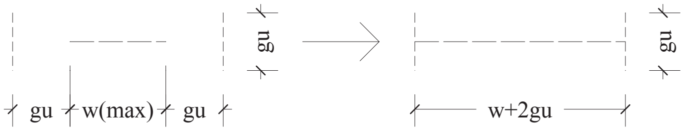
6.40 Rules of the Durand grammar continued.

rule-durand-grid-expand-uv-terminate



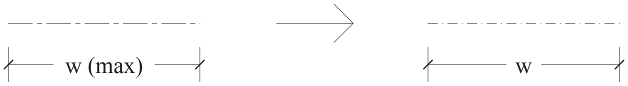
DESCRIPTION:	Rule to erase the focus lines.
CONSTRAINT:	w > 0, h > 0
LABEL-FILTER:	on
APPLY-MODE:	parallel
SUCCESS:	rule-durand-grid-extend-axis
FAILURE:	rule-durand-grid-extend-axis
RULE-SET:	nil

rule-durand-grid-extend-axis



DESCRIPTION:	Rule to terminate the expanded grid.
CONSTRAINT:	w > 0
LABEL-FILTER:	on
APPLY-MODE:	parallel
SUCCESS:	nil
FAILURE:	nil
RULE-SET:	nil

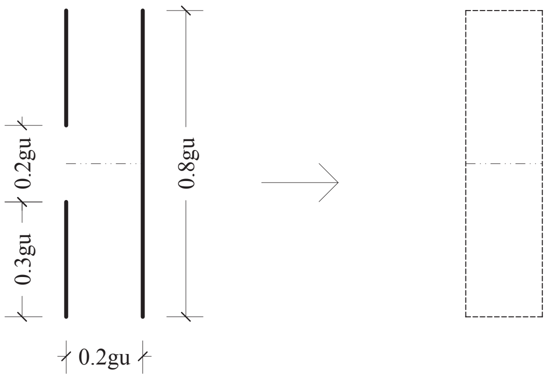
rule-durand-opening-all-axis



DESCRIPTION:	Primary rule of a macro to cut out openings on all the walls that intersect the selected axis which is first converted to a temp label.
CONSTRAINT:	w > 0
LABEL-FILTER:	on
APPLY-MODE:	single
SUCCESS:	rule-durand-opening-all-axis-step2
FAILURE:	nil
RULE-SET:	nil

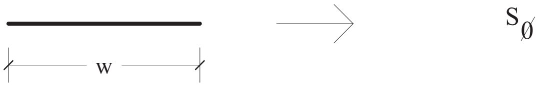
6.40 Rules of the Durand grammar continued.

rule-durand-opening-all-axis-step2



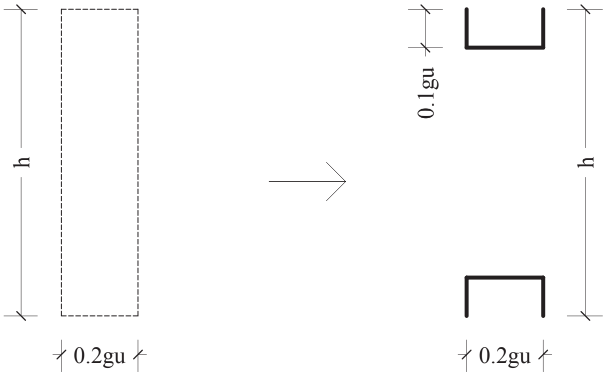
DESCRIPTION:	Rule to make an opening along a temp labeled axis. Works for both normal and indented walls.
CONSTRAINT:	nil
LABEL-FILTER:	on
APPLY-MODE:	parallel
SUCCESS:	rule-durand-opening-all-axis-step3
FAILURE:	rule-durand-opening-all-axis-step5
RULE-SET:	nil

rule-durand-opening-all-axis-step3



DESCRIPTION:	Rule to erase all wall labeled lines.
CONSTRAINT:	w > 0
LABEL-FILTER:	on
APPLY-MODE:	parallel
SUCCESS:	rule-durand-opening-all-axis-step4
FAILURE:	rule-durand-opening-all-axis-step4
RULE-SET:	nil

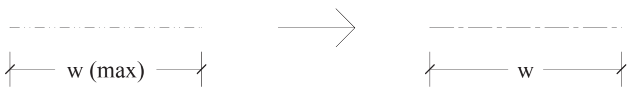
rule-durand-opening-all-axis-step4



DESCRIPTION:	Rule to remove the focus lines and insert the ends of the walls.
CONSTRAINT:	h > 0
LABEL-FILTER:	on
APPLY-MODE:	parallel
SUCCESS:	rule-durand-opening-all-axis-step5
FAILURE:	rule-durand-opening-all-axis-step5
RULE-SET:	nil

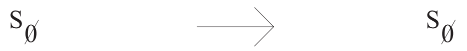
6.40 Rules of the Durand grammar continued.

rule-durand-opening-all-axis-step5



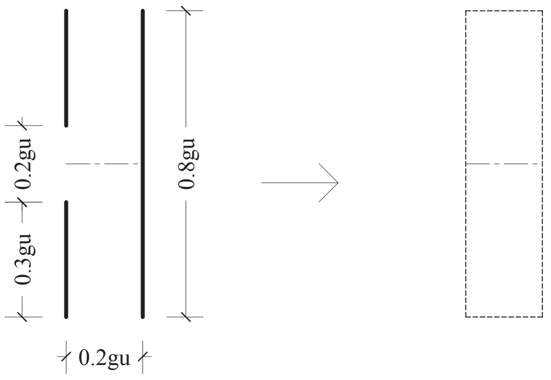
DESCRIPTION:	Rule to replace temp label lines back to axis label lines.
CONSTRAINT:	w > 0
LABEL-FILTER:	on
APPLY-MODE:	random
SUCCESS:	nil
FAILURE:	nil
RULE-SET:	nil

rule-durand-opening-complete



DESCRIPTION:	Rule to move onto the next stage which is Platforms.
CONSTRAINT:	nil
LABEL-FILTER:	nil
APPLY-MODE:	nil
SUCCESS:	nil
FAILURE:	nil
RULE-SET:	set-rule (rule-durand-grid-expand-u, rule-durand-grid-expand-uv, rule-durand-platform-make-one, rule-durand-platform-make-ring, rule-durand-platform-complete)

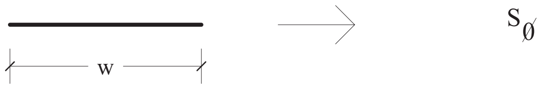
rule-durand-opening-one-axis



DESCRIPTION:	Rule to make an opening along an axis. Works for both normal and indented walls.
CONSTRAINT:	nil
LABEL-FILTER:	on
APPLY-MODE:	single
SUCCESS:	rule-durand-opening-one-axis-step2
FAILURE:	nil
RULE-SET:	nil

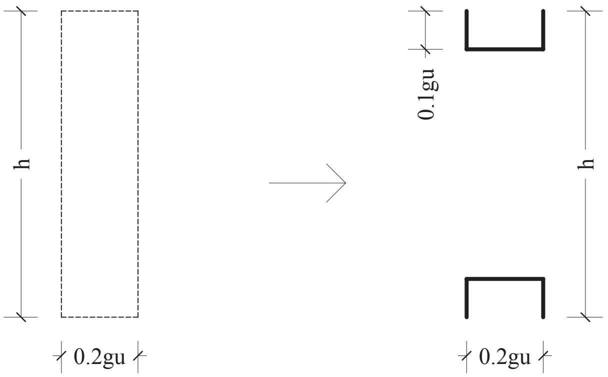
6.40 Rules of the Durand grammar continued.

rule-durand-opening-one-axis-step2



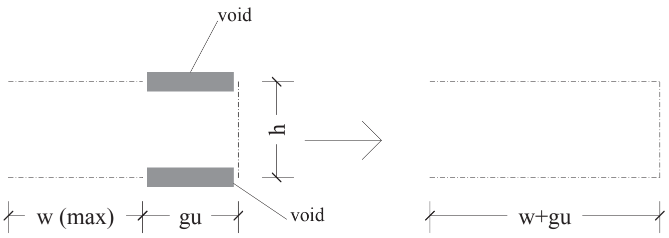
DESCRIPTION:	Rule to erase all wall labeled lines.
CONSTRAINT:	w > 0
LABEL-FILTER:	on
APPLY-MODE:	parallel
SUCCESS:	rule-durand-opening-one-axis-step3
FAILURE:	rule-durand-opening-one-axis-step3
RULE-SET:	nil

rule-durand-opening-one-axis-step3



DESCRIPTION:	Rule to remove the focus lines and insert the ends of the walls.
CONSTRAINT:	h > 0
LABEL-FILTER:	on
APPLY-MODE:	random
SUCCESS:	nil
FAILURE:	nil
RULE-SET:	nil

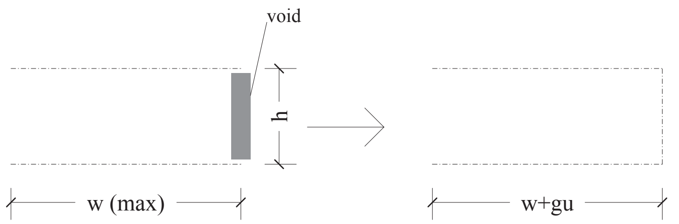
rule-durand-parti-fix1



DESCRIPTION:	Rule to fix problematic parti arrangements.
CONSTRAINT:	w > 0, h > 0
LABEL-FILTER:	on
APPLY-MODE:	parallel
SUCCESS:	rule-durand-parti-fix2
FAILURE:	rule-durand-parti-fix2
RULE-SET:	nil

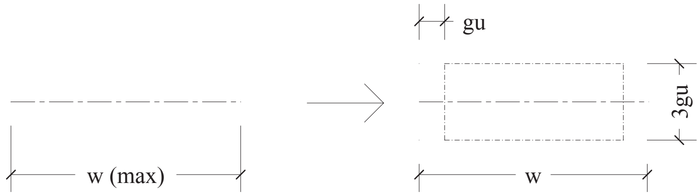
6.40 Rules of the Durand grammar continued.

rule-durand-parti-fix2



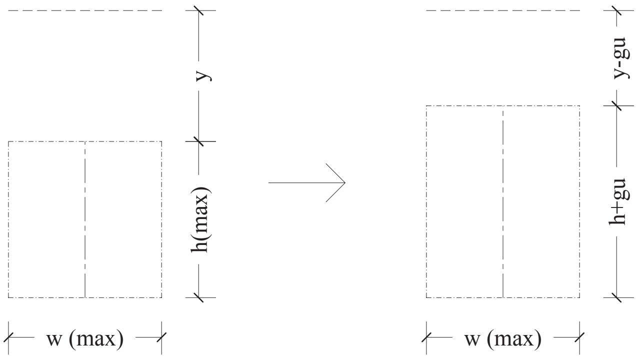
DESCRIPTION:	Rule to fix problematic parti arrangements.
CONSTRAINT:	w > 0, h > 0
LABEL-FILTER:	on
APPLY-MODE:	parallel
SUCCESS:	nil
FAILURE:	nil
RULE-SET:	nil

rule-durand-parti-generate



DESCRIPTION:	Rule to generate an initial parti from the axes.
CONSTRAINT:	w > 0
LABEL-FILTER:	on
APPLY-MODE:	parallel
SUCCESS:	nil
FAILURE:	nil
RULE-SET:	set-rule(rule-durand-parti-shorten, rule-durand-parti-lengthen, rule-durand-parti-widen, rule-durand-parti-narrow, rule-durand-parti-move-along-axis, rule-durand-parti-shift-with-axis, rule-durand-wall-generate)

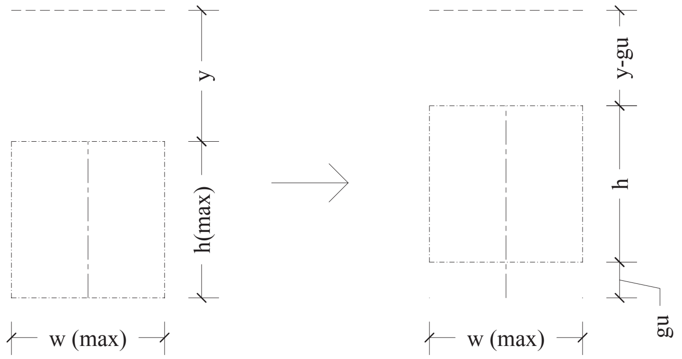
rule-durand-parti-lengthen



DESCRIPTION:	Rule to lengthen a rectangular parti by one grid unit.
CONSTRAINT:	w > 0, h > w-gu, y > 1.5gu
LABEL-FILTER:	on
APPLY-MODE:	single
SUCCESS:	nil
FAILURE:	nil
RULE-SET:	nil

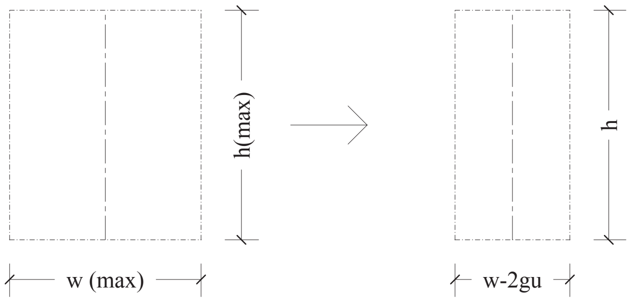
6.40 Rules of the Durand grammar continued.

rule-durand-parti-move-along-axis



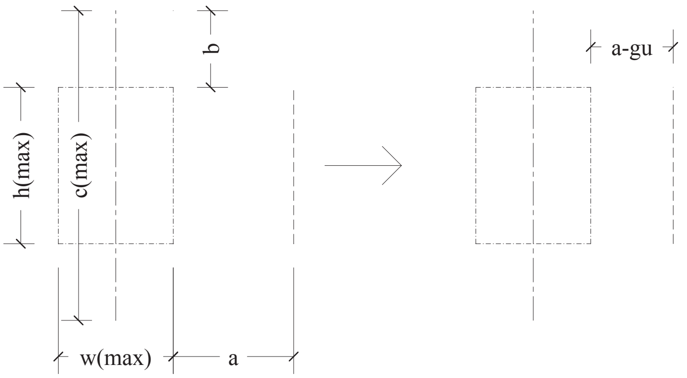
DESCRIPTION:	Rule to move a rectangular parti along an axis by one grid unit.
CONSTRAINT:	w > 0, h > w-gu, y > 1.5gu
LABEL-FILTER:	on
APPLY-MODE:	single
SUCCESS:	nil
FAILURE:	nil
RULE-SET:	nil

rule-durand-parti-narrow



DESCRIPTION:	Rule to narrow a rectangular parti by one grid unit.
CONSTRAINT:	w > 2gu, h > w-gu
LABEL-FILTER:	on
APPLY-MODE:	single
SUCCESS:	rule-durand-parti-fix1
FAILURE:	nil
RULE-SET:	nil

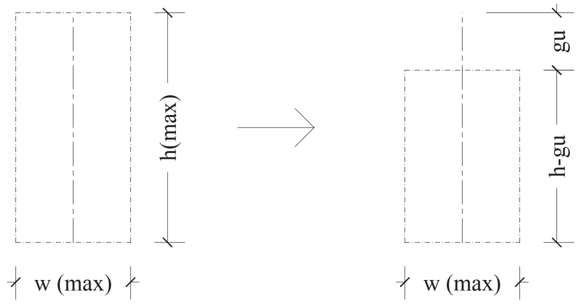
rule-durand-parti-shift-with-axis



DESCRIPTION:	Rule to move a parti rectangle with the center axis line one grid unit in the u direction.
CONSTRAINT:	w>0, h>w-gu, a>gu, b>0, c>0
LABEL-FILTER:	on
APPLY-MODE:	single
SUCCESS:	rule-durand-parti-fix1
FAILURE:	nil
RULE-SET:	nil

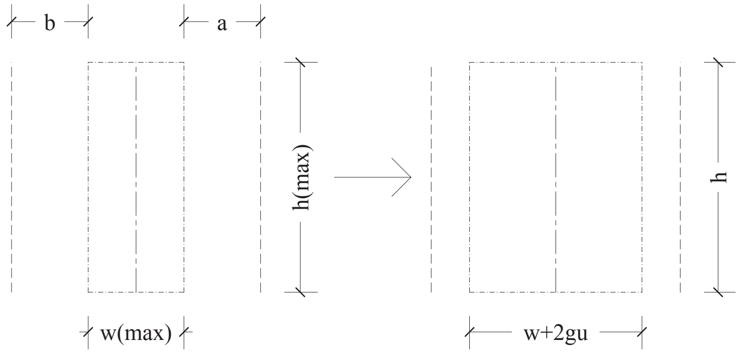
6.40 Rules of the Durand grammar continued.

rule-durand-parti-shorten



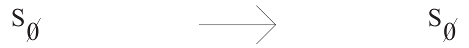
DESCRIPTION:	Rule to shorten a rectangular parti by one grid unit.
CONSTRAINT:	w > 0, h > w-gu
LABEL-FILTER:	on
APPLY-MODE:	single
SUCCESS:	nil
FAILURE:	nil
RULE-SET:	nil

rule-durand-parti-widen



DESCRIPTION:	Rule to widen a rectangular parti by one grid unit.
CONSTRAINT:	w>0, h>w-gu, a>1.1gu, b>1.1gu
LABEL-FILTER:	on
APPLY-MODE:	single
SUCCESS:	rule-durand-parti-fix1
FAILURE:	nil
RULE-SET:	nil

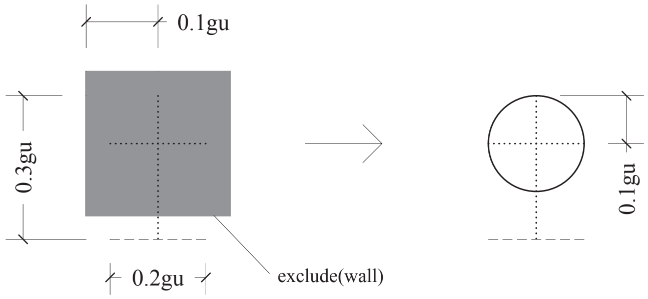
rule-durand-platform-complete



DESCRIPTION:	Rule to move onto the next stage which is Exterior Stairs.
CONSTRAINT:	nil
LABEL-FILTER:	nil
APPLY-MODE:	nil
SUCCESS:	nil
FAILURE:	nil
RULE-SET:	set-rule (rule-durand-grid-expand-u, rule-durand-grid-expand-uv, rule-durand-exterior-stair-wall, rule-durand-exterior-stair-platform, rule-durand-exterior-stair-extend, rule-durand-exterior-stair-erase, rule-durand-end-grammar)

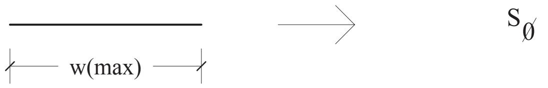
6.40 Rules of the Durand grammar continued.

rule-durand-platform-complete-with-columns



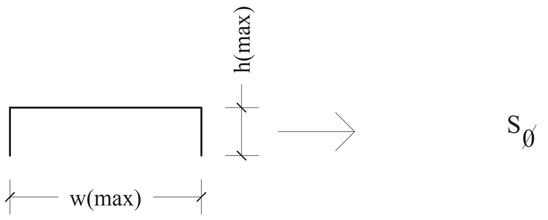
DESCRIPTION:	Rule to move onto the next stage which is Exterior Stairs after adding a series of columns along the edge of all platforms.
CONSTRAINT:	nil
LABEL-FILTER:	nil
APPLY-MODE:	nil
SUCCESS:	nil
FAILURE:	nil
RULE-SET:	set-rule (rule-durand-grid-expand-u, rule-durand-grid-expand-uv, rule-durand-exterior-stair-wall, rule-durand-exterior-stair-platform, rule-durand-exterior-stair-extend, rule-durand-exterior-stair-erase, rule-durand-end-grammar)

rule-durand-platform-erase-all



DESCRIPTION:	Rule to erase all platform labeled lines.
CONSTRAINT:	w > 0
LABEL-FILTER:	on
APPLY-MODE:	parallel
SUCCESS:	nil
FAILURE:	nil
RULE-SET:	nil

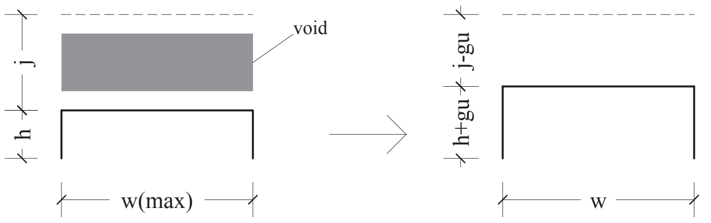
rule-durand-platform-erase-one



DESCRIPTION:	Rule to erase one platform.
CONSTRAINT:	w > 0, h > 0
LABEL-FILTER:	on
APPLY-MODE:	user
SUCCESS:	nil
FAILURE:	nil
RULE-SET:	nil

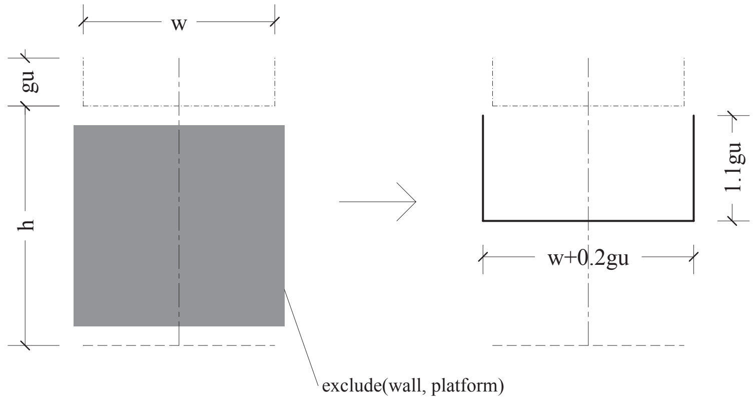
6.40 Rules of the Durand grammar continued.

rule-durand-platform-extend-one



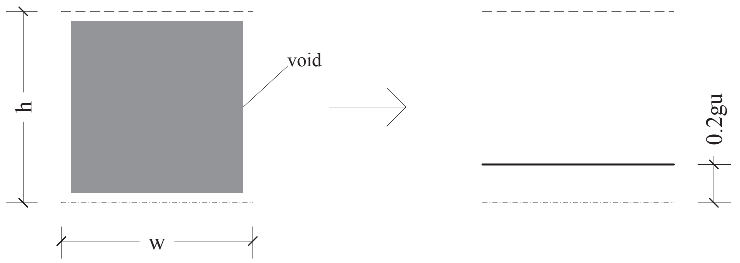
DESCRIPTION:	Rule to extend one platform by one gridunit.
CONSTRAINT:	w > 0, h > 0, j > 1.1gu
LABEL-FILTER:	on
APPLY-MODE:	single
SUCCESS:	nil
FAILURE:	nil
RULE-SET:	nil

rule-durand-platform-make-one



DESCRIPTION:	Rule to make a platform on axis against a wall.
CONSTRAINT:	w > 0, h > 1.1gu
LABEL-FILTER:	on
APPLY-MODE:	single
SUCCESS:	nil
FAILURE:	nil
RULE-SET:	nil

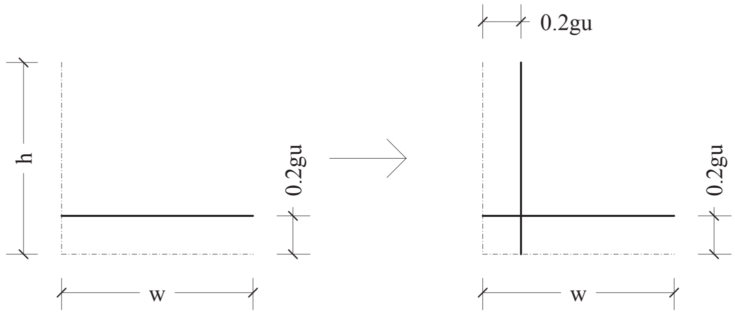
rule-durand-platform-make-ring



DESCRIPTION:	Primary rule of a macro to place a platform around the entire building footprint.
CONSTRAINT:	w > 0, h > 0
LABEL-FILTER:	on
APPLY-MODE:	parallel
SUCCESS:	rule-durand-platform-make-ring-step2
FAILURE:	nil
RULE-SET:	set-rule(rule-durand-grid-expand-u, rule-durand-grid-expand-uv, rule-durand-platform-erase-all, r-d-platform-extend-one, r-d-platform-retract-one, r-d-p-complete-with-columns, rule-durand-platform-complete)

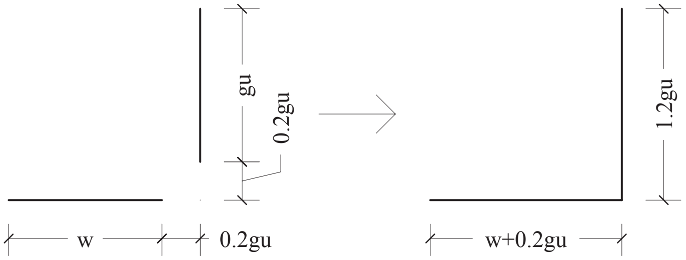
6.40 Rules of the Durand grammar continued.

rule-durand-platform-make-ring-step2



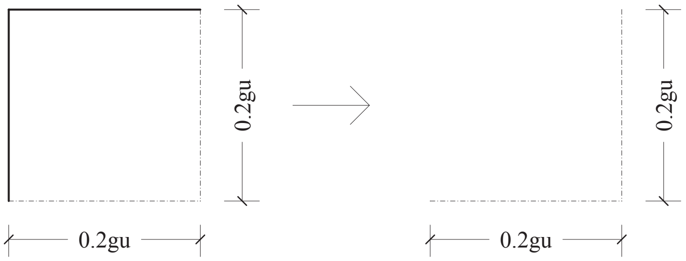
DESCRIPTION:	rule to place a platform along a s-shaped parti line.
CONSTRAINT:	w > 0, h > 0
LABEL-FILTER:	on
APPLY-MODE:	parallel
SUCCESS:	rule-durand-platform-make-ring-step3
FAILURE:	rule-durand-platform-make-ring-step3
RULE-SET:	nil

rule-durand-platform-make-ring-step3



DESCRIPTION:	Rule to fillet two platform lines together to make a corner.
CONSTRAINT:	w > 0
LABEL-FILTER:	on
APPLY-MODE:	parallel
SUCCESS:	rule-durand-platform-make-ring-step4
FAILURE:	rule-durand-platform-make-ring-step4
RULE-SET:	nil

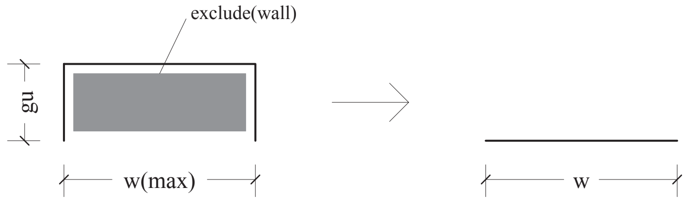
rule-durand-platform-make-ring-step4



DESCRIPTION:	Rule to trim the extended platform lines against the parti.
CONSTRAINT:	nil
LABEL-FILTER:	on
APPLY-MODE:	parallel
SUCCESS:	nil
FAILURE:	nil
RULE-SET:	nil

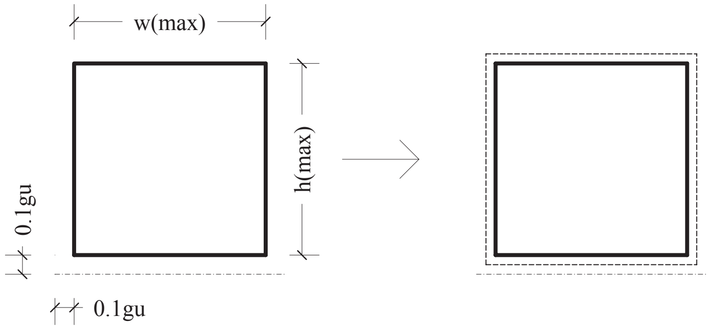
6.40 Rules of the Durand grammar continued.

rule-durand-platform-retract-one



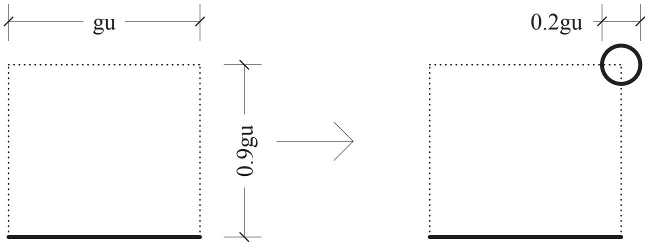
DESCRIPTION:	Rule to retract one platform by one gridunit.
CONSTRAINT:	w > 0
LABEL-FILTER:	on
APPLY-MODE:	user
SUCCESS:	nil
FAILURE:	nil
RULE-SET:	nil

rule-durand-room-all-columns



DESCRIPTION:	Primary rule of a macro to place columns around a room.
CONSTRAINT:	w > 0, h > 0
LABEL-FILTER:	on
APPLY-MODE:	single
SUCCESS:	rule-durand-room-all-columns-step1
FAILURE:	nil
RULE-SET:	nil

rule-durand-room-all-columns-step2



DESCRIPTION:	Rule to place columns around the edge of the room.
CONSTRAINT:	nil
LABEL-FILTER:	on
APPLY-MODE:	parallel
SUCCESS:	rule-durand-room-terminate
FAILURE:	rule-durand-room-terminate
RULE-SET:	nil

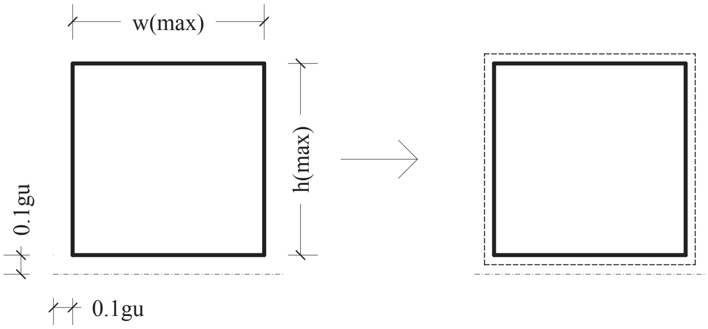
6.40 Rules of the Durand grammar continued.

rule-durand-room-columns-complete

$$S_{\emptyset} \longrightarrow S_{\emptyset}$$

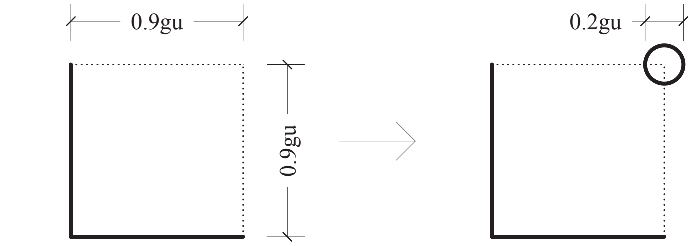
DESCRIPTION:	Rule to go to the next stage which is Room Indentations.
CONSTRAINT:	nil
LABEL-FILTER:	nil
APPLY-MODE:	nil
SUCCESS:	nil
FAILURE:	nil
RULE-SET:	set-rule(rule-durand-room-indent, rule-durand-room-unindent, rule-durand-indent-complete)

rule-durand-room-corner-columns



DESCRIPTION:	Primary rule of a macro to place columns at the corners of a room.
CONSTRAINT:	w > 0, h > 0
LABEL-FILTER:	on
APPLY-MODE:	single
SUCCESS:	rule-durand-room-corner-columns-step1
FAILURE:	nil
RULE-SET:	nil

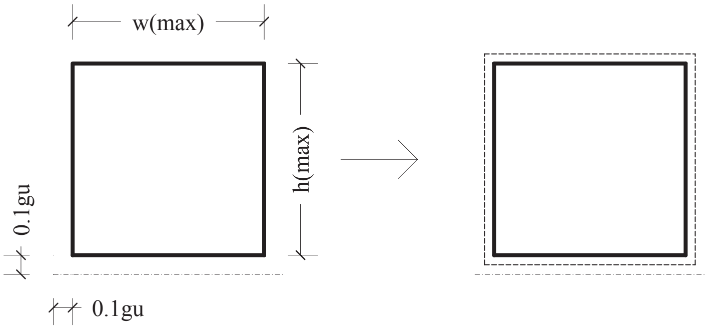
rule-durand-room-corner-columns-step2



DESCRIPTION:	Rule to place a column at the corners of a room.
CONSTRAINT:	w > 0, h > 0
LABEL-FILTER:	off
APPLY-MODE:	parallel
SUCCESS:	rule-durand-room-terminate
FAILURE:	rule-durand-room-terminate
RULE-SET:	nil

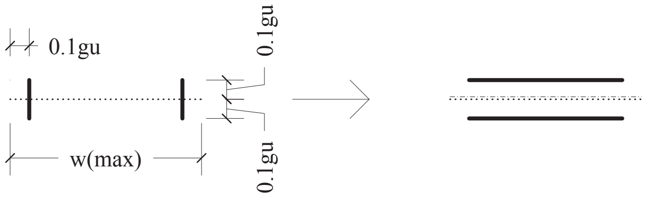
6.40 Rules of the Durand grammar continued.

rule-durand-room-divide



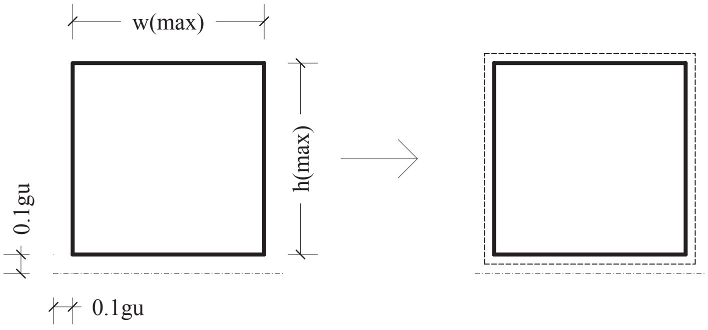
DESCRIPTION:	Primary rule of a macro to divide a room.
CONSTRAINT:	w > 0, h > 0
LABEL-FILTER:	on
APPLY-MODE:	single
SUCCESS:	rule-durand-room-divide-step1
FAILURE:	nil
RULE-SET:	nil

rule-durand-room-divide-step2



DESCRIPTION:	Rule to add a wall along a grid line.
CONSTRAINT:	w > 0
LABEL-FILTER:	on
APPLY-MODE:	single
SUCCESS:	rule-durand-room-terminate
FAILURE:	rule-durand-room-terminate
RULE-SET:	nil

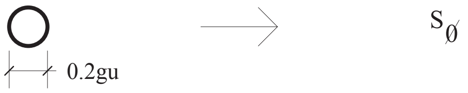
rule-durand-room-erase-columns



DESCRIPTION:	Primary rule of a macro to erase all the columns in a room.
CONSTRAINT:	w > 0, h > 0
LABEL-FILTER:	on
APPLY-MODE:	single
SUCCESS:	rule-durand-room-erase-columns-step1
FAILURE:	nil
RULE-SET:	nil

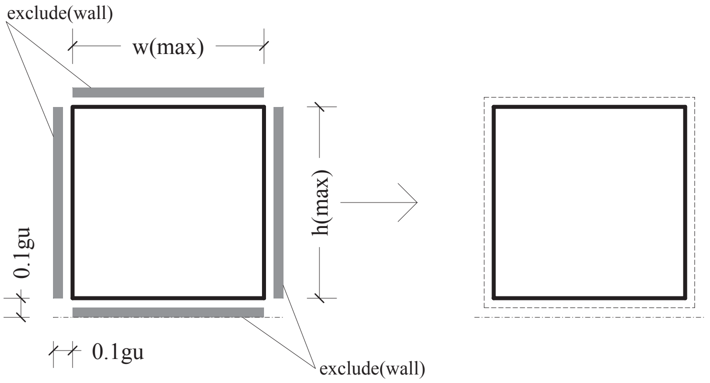
6.40 Rules of the Durand grammar continued.

rule-durand-room-erase-columns-step2



DESCRIPTION:	Rule to erase a column.
CONSTRAINT:	nil
LABEL-FILTER:	on
APPLY-MODE:	parallel
SUCCESS:	nil
FAILURE:	nil
RULE-SET:	nil

rule-durand-room-indent



DESCRIPTION:	Primary rule of a macro to place indentations around a room.
CONSTRAINT:	w > 0, h > 0
LABEL-FILTER:	on
APPLY-MODE:	single
SUCCESS:	rule-durand-room-indent-step2
FAILURE:	nil
RULE-SET:	nil

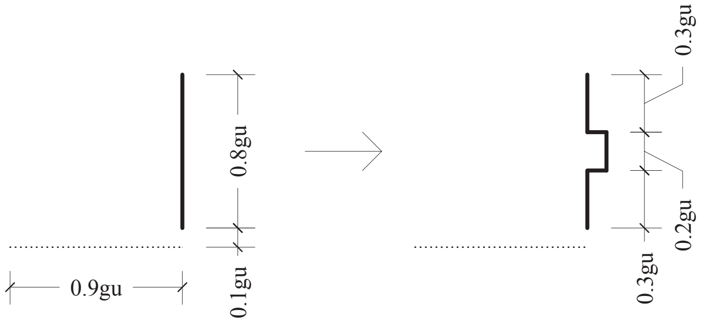
rule-durand-room-indent-complete

$$S_{\emptyset} \longrightarrow S_{\emptyset}$$

DESCRIPTION:	Rule to go to the next stage which is Openings.
CONSTRAINT:	nil
LABEL-FILTER:	nil
APPLY-MODE:	nil
SUCCESS:	nil
FAILURE:	nil
RULE-SET:	set-rule(rule-durand-axis-make, rule-durand-opening-one-axis, rule-durand-opening-all-axis, rule-durand-opening-complete)

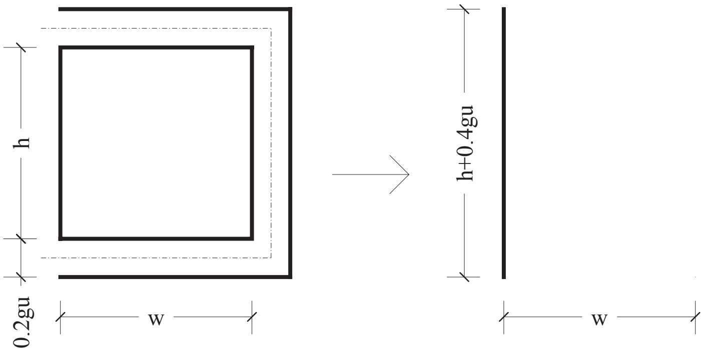
6.40 Rules of the Durand grammar continued.

rule-durand-room-indent-step2



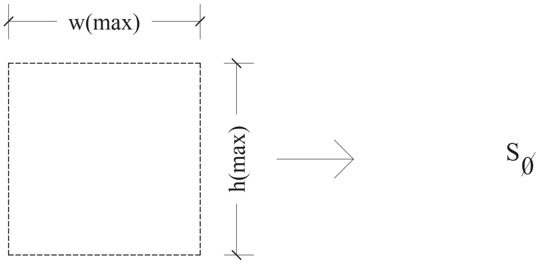
DESCRIPTION:	Rule to make an indentation in the walls.
CONSTRAINT:	nil
LABEL-FILTER:	on
APPLY-MODE:	parallel
SUCCESS:	rule-durand-room-terminate
FAILURE:	rule-durand-room-terminate
RULE-SET:	nil

rule-durand-room-remove-bay



DESCRIPTION:	Rule to erase a protruding bay.
CONSTRAINT:	w > 0, h > 0
LABEL-FILTER:	on
APPLY-MODE:	single
SUCCESS:	nil
FAILURE:	nil
RULE-SET:	nil

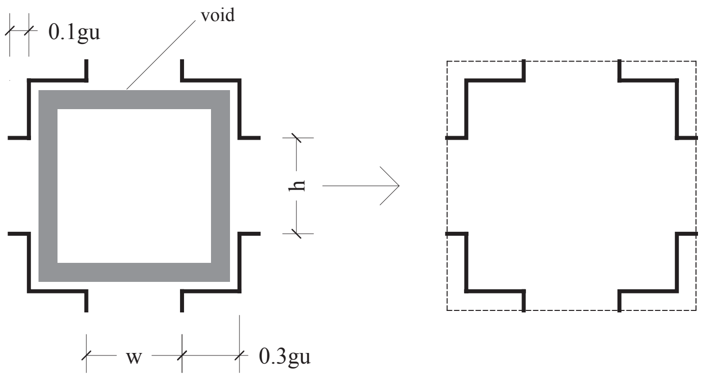
rule-durand-room-terminate



DESCRIPTION:	Rule to remove a rectangular focus polygon from the drawing.
CONSTRAINT:	w > 0, h > 0
LABEL-FILTER:	on
APPLY-MODE:	random
SUCCESS:	nil
FAILURE:	nil
RULE-SET:	nil

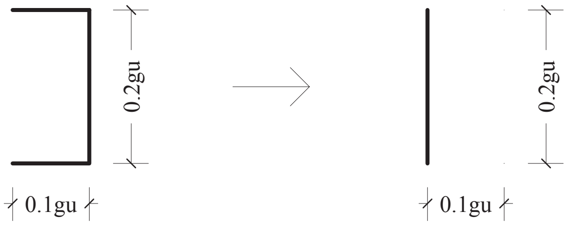
6.40 Rules of the Durand grammar continued.

rule-durand-room-unindent



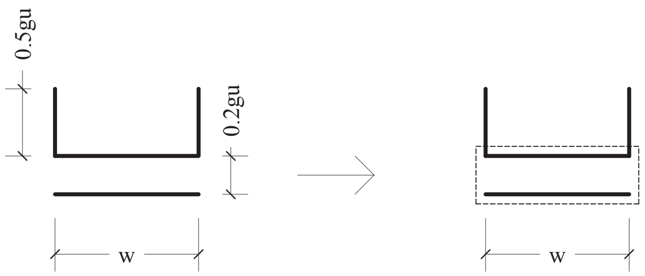
DESCRIPTION:	Primary rule of a macro to remove the indentations around a room.
CONSTRAINT:	w > 0, h > 0
LABEL-FILTER:	on
APPLY-MODE:	single
SUCCESS:	rule-durand-room-unindent-step2
FAILURE:	nil
RULE-SET:	nil

rule-durand-room-unindent-step2



DESCRIPTION:	Rule to fill in the indentations in a wall.
CONSTRAINT:	nil
LABEL-FILTER:	on
APPLY-MODE:	parallel
SUCCESS:	rule-durand-room-terminate
FAILURE:	rule-durand-room-terminate
RULE-SET:	nil

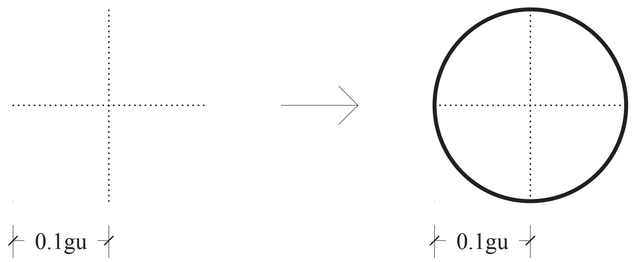
rule-durand-wall-column



DESCRIPTION:	Rule to pick a wall and convert it into a series of columns.
CONSTRAINT:	w > gu
LABEL-FILTER:	on
APPLY-MODE:	single
SUCCESS:	rule-durand-wall-column-step2
FAILURE:	nil
RULE-SET:	nil

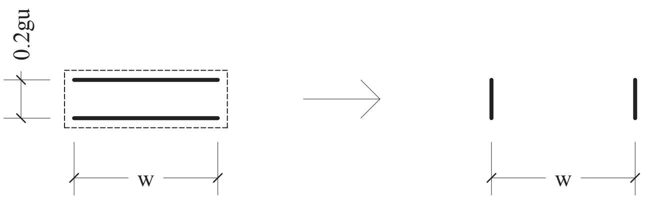
6.40 Rules of the Durand grammar continued.

rule-durand-wall-column-step2



DESCRIPTION:	Rule to add a circle at the grid intersections.
CONSTRAINT:	nil
LABEL-FILTER:	on
APPLY-MODE:	parallel
SUCCESS:	rule-durand-wall-column-step3
FAILURE:	rule-durand-wall-column-step3
RULE-SET:	nil

rule-durand-wall-column-step3



DESCRIPTION:	Rule to remove focus lines and replace walls.
CONSTRAINT:	w > 0
LABEL-FILTER:	on
APPLY-MODE:	random
SUCCESS:	nil
FAILURE:	nil
RULE-SET:	nil

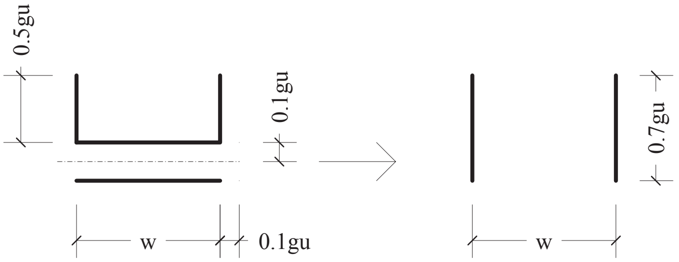
rule-durand-wall-complete

$$S_{\emptyset} \longrightarrow S_{\emptyset}$$

DESCRIPTION:	Rule to go to the next stage which is Room Columns.
CONSTRAINT:	nil
LABEL-FILTER:	nil
APPLY-MODE:	nil
SUCCESS:	nil
FAILURE:	nil
RULE-SET:	set-rule(rule-durand-room-all-columns, rule-durand-room-corner-columns, rule-durand-room-erase-columns, rule-durand-room-columns-complete)

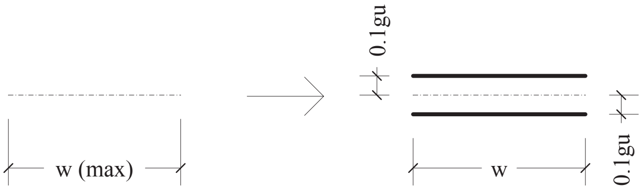
6.40 Rules of the Durand grammar continued.

rule-durand-wall-erase



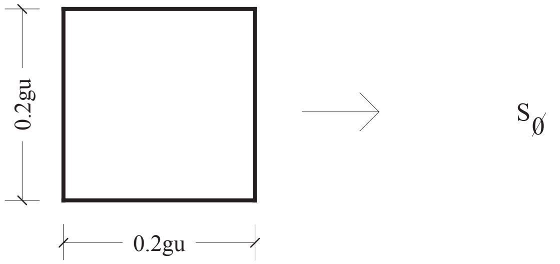
DESCRIPTION:	Rule to erase a wall and its associated parti line.
CONSTRAINT:	nil
LABEL-FILTER:	on
APPLY-MODE:	single
SUCCESS:	nil
FAILURE:	nil
RULE-SET:	nil

rule-durand-wall-generate



DESCRIPTION:	Rule to generate the walls from the parti and to move on to the next stage which is Walls.
CONSTRAINT:	w > 0
LABEL-FILTER:	on
APPLY-MODE:	parallel
SUCCESS:	rule-durand-wall-trim-+shape
FAILURE:	nil
RULE-SET:	set-rule(rule-durand-wall-erase, rule-durand-wall-column, rule-durand-room-divide, rule-durand-room-remove-bay, rule-durand-wall-complete)

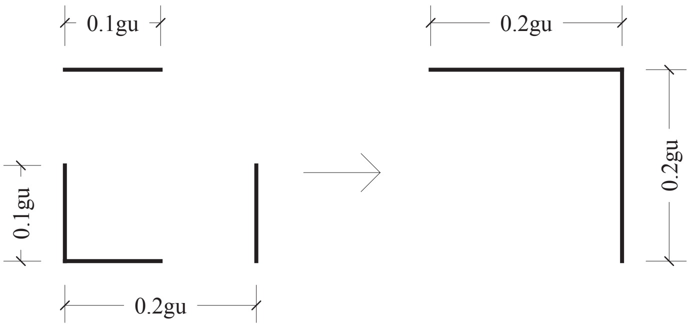
rule-durand-wall-trim-+shape



DESCRIPTION:	Rule to trim out cross shaped intersections.
CONSTRAINT:	nil
LABEL-FILTER:	on
APPLY-MODE:	parallel
SUCCESS:	rule-durand-wall-trim-Tshape
FAILURE:	rule-durand-wall-trim-Tshape
RULE-SET:	nil

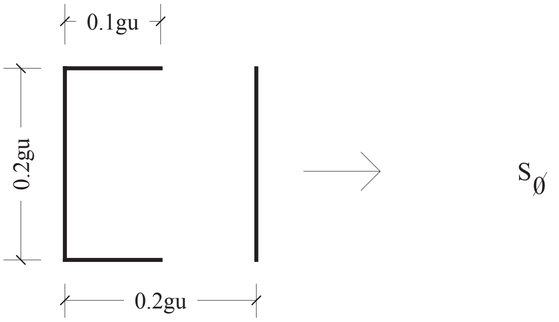
6.40 Rules of the Durand grammar continued.

rule-durand-wall-trim-Lshape



DESCRIPTION:	Rule to trim out L shaped intersections.
CONSTRAINT:	nil
LABEL-FILTER:	on
APPLY-MODE:	parallel
SUCCESS:	nil
FAILURE:	nil
RULE-SET:	nil

rule-durand-wall-trim-Tshape



DESCRIPTION:	Rule to trim out T shaped intersections.
CONSTRAINT:	nil
LABEL-FILTER:	on
APPLY-MODE:	parallel
SUCCESS:	rule-durand-wall-trim-Lshape
FAILURE:	rule-durand-wall-trim-Lshape
RULE-SET:	nil

6.40 Rules of the Durand grammar continued.

7.0 Contribution

The key contribution of this thesis is the seven descriptors that make up the shape grammar meta-language (figures 3.02 and 4.01). The five major benefits are:

(1) Organize a set of rules for the user to choose from.
The rule-set descriptor uses a parallel description to present the user with a salient set of rules to choose from during the derivation of the grammar. The parallel description of rules guides the user through the design process described by the grammar.

(2) Group together a series of rules.
The directive descriptor explicitly links together a series of rules to create a macro. Macros are a logical set of rules that perform a single design transformation and are used to encapsulate design knowledge.

(3) Filter information in a drawing.
The s() function is used to filter information in a drawing to restrict where a rule can apply. The two descriptors that are examples of the s() function are label-filter and focus. The label-filter descriptor removes specified labeled shapes from the drawing. The focus descriptor removes specified areas of the drawing. In the shape grammar language, the s() function is applied to the drawing C.

$$t(g(A)) \leq s(C)$$

(4) Constrain where a rule can apply.
The p() function adds context as a constraint on the matching conditions between the schema and the drawing. The two descriptors that are examples of the p() function are maxline and zone. The maxline descriptor constrains a line in the subshape to be a maximal line the drawing. The zone descriptor associates a predicate function with an area of the schema. An example predicate function is the void function which identifies empty spaces. In the shape grammar language, the p() function is applied to the schema A.

$$p(t(g(A))) \leq C$$

(5) Control how a rule is applied.
The apply-mode descriptor specifies how a set of subshapes is applied to the drawing. Typically only a single subshape is applied to the drawing but the apply-mode descriptor has other options that can apply a rule in parallel or at random.

8.0 Summary

The shape grammar meta-language is a set of seven descriptors that can modify the conditions surrounding the rule application process in the shape grammar formalism. These conditions include rule selection, drawing state, matching conditions, and application method. Each condition is a phase in the rule application process. The matching conditions can be further subdivided into parameter requirements, transformation requirements, and contextual requirements phases. The entire rule application process can be explained with a series of six phases: rule selection, drawing state, parameter requirements, transformation requirements, contextual requirements, and application method.

The first phase of the rule application process is the rule selection phase. In this phase, the user determines which rule to apply. The meta-language has two descriptors that can manipulate which rule gets selected. The first descriptor is the directive which links a series of rules together to make a macro. There are two components to the directive: the success rule and the failure rule. The success rule determines which rule to apply next if the given rule applies successfully to the drawing. The failure rule determines which rule to apply next if the given rule can not be applied to the drawing. The rules specified by the directive can be any rule in the grammar.

The other descriptor in the rule selection phase is rule-set which provides the user with a set of rules to choose from. Any rule outside of the rule-set can not be used even if the rule is applicable. There are three control options to the rule-set descriptor: set-rule, add-rule, sub-rule. The set-rule option defines a rule-set. The add-rule option adds rules to the rule-set. And the sub-rule option removes a rule from the rule-set. The rule-set acts as a guide to help the user navigate through the grammar. It is also a means to differentiate the primary rule of a macro from the secondary rules of a macro.

After selecting a rule, the user must decide where in the drawing to apply the rule. This is determined in the drawing state phase. The meta-language has two descriptors that filter information in the drawing to restrict where a rule can apply. The first descriptor is label-filter which filters out any labeled shape in the drawing that is not a labeled shape in the left-hand schema of the given rule. This temporary state is persistent until the rule is finished applying. The second descriptor is focus. Whereas the label-filter descriptor filters out elements of the drawing, the focus descriptor filters out areas of the drawing. The descriptor uses special labeled lines to mark off areas of the drawing that can be used to apply a rule. The portions of the drawing outside of the marked area can not be used for rule application.

Once the rule has been selected and the drawing state determined, the matching conditions of the rule must be evaluated. There are three phases that determine the matching conditions between the left-hand schema of the rule and the drawing: parameter requirements, transformation requirements, and contextual requirements. The parameter and transformation requirements are part of the original shape grammar formalism.

These phases determine what values are assigned to the parameters of a schema and what set of transformations are necessary in order to have a subshape match.

The meta-language adds a third phase, contextual requirements. In this phase, tests are performed to determine if the context of the subshape is satisfactory. There are two descriptors in this phase: maxline and zone. The maxline descriptor restricts a line of the subshape to be a maximal line. The context, in this case, is the fact that a subshape line must not be a smaller portion of a larger line in the drawing. The second descriptor is zone which tests if specified areas of the subshape in the drawing satisfy a predicate function. By using the zone descriptor, a schema can take into account the conditions surrounding the subshape. Two predicate functions are given as examples. The first example is the void function which tests if the marked area of the subshape is void of any shapes. The second example is the exclude function which tests if the marked area is void of specific labeled shapes.

The last decision a user makes in applying a rule is determining how to apply the rule. After going through the first five phases, a set of possible subshapes is created. This set may contain only one subshape or many subshapes. The apply-mode descriptor specifies the number of subshapes to apply. The options are single, parallel, and random. The single option allows the user to select a single subshape for rule application. The parallel option selects all the subshapes for rule application. And the random option randomly selects one subshape to apply.

The shape grammar meta-language makes some changes to the original formulas in the shape grammar formalism. The original formulas are:

$$\text{For all } t \text{ and } g \text{ such that } t(g(A)) \leq C \qquad (3)$$
$$C' = \sum(C - t(g(A)) + t(g(B))) \qquad (4)$$

The meta-language changes the formulas to be:

$$\text{For all } t \text{ and } g \text{ such that } p(t(g(A))) \leq s(C) \qquad (9)$$
$$C' = \sum (s(C) - p(t(g(A))) + t(g(B))) \qquad (10)$$

Where p() is the contextual requirements phase and s() is the drawing state phase.

This thesis gives three examples of how the shape grammar meta-language can be used to describe a process of design. The first example is the Bilateral Grid grammar. This grammar is a rewrite of the first stage of the Palladian grammar where a bilateral grid is created. The Bilateral Grid grammar uses the meta-language descriptors to create macros that give the user design choices to increase or decrease the size of the grid while removing the possibility to make dead-end states. The original grammar used a series of steps to create a bilateral grid. At each step, the user is given two or three different choices. If the user made the wrong choice, a dead-end state would be created. Instead of having the user guess the implicit sequencing of rules, the directive descriptor explicitly links together a logical series of rules to create a macro. Each macro is designed to adjust one aspect of the grid size. The macros can increase or decrease either the width or height of the grid. The Bilateral Grammar also uses the rule-set descriptor to prevent the user from selecting the secondary rules of a macro and the void function of the zone descriptor is used to test if the grid is complete.

The second example is the Hexagon Path grammar. This grammar is developed as a game between two players to demonstrate that shape grammars can be used for more than just design generation, it can also be used as a scripting language to emulate a game. The objective of the game is to prevent your opponent from making a move. Each player can only place a piece that is adjacent to the head piece. The new piece can be either a new head piece or a block piece. If a new head piece is placed, the old head piece is converted into a tail piece. The meta-language is used to perform three basic operations of the game. The rule-set descriptor is used to alternate control between the two players and also determines which player has lost the game. The directive descriptor is used to create a macro that places a new head piece and subsequently changes the old head piece into a tail piece. The zone descriptor uses the void function to determine all the viable moves a player can make.

The third example is the Durand grammar which demonstrates how the meta-language can be used to emulate Durand's process for designing two-dimensional floor plans. His design process is a linear sequence of events. First establish the axes. From the axes, create the parti. Make the walls from the parti and then add other architectural elements that fit the grid and are collinear with other elements in the design. His linear sequence of design can be described in stages.

The rule-set descriptor divides the grammar into eight stages. At each stage of the design, only a select few rules are presented to the user, typically not more than ten. The rules in each stage present the user with salient choices to modify the design. By presenting only a select few, the user is not overwhelmed with the total number of rules in the grammar. The grammar also uses the directive descriptor to make several macros. There are two types of macros exemplified in the grammar: procedural and serial. The first type is procedural and is commonly used in conjunction with the focus descriptor.

The procedural macro can be broken down into three basic parts. The first part is to isolate the working area. In other words, where are the changes that need to occur? This is achieved by placing a focus polygon around the area that needs to be changed. The second part is to perform the design transformations. Since the focus polygon is in use, only the area inside the focus polygon will get changed. The last step is to remove the focus polygon and perform any other necessary "clean up" changes to the drawing. Some examples of procedural macros are r*ule-durand-wall-column*, *rule-durand-room-indent*, and *rule-durand-openings-all-axis*.

The second type is serial. Sometimes a design transformation requires several steps in order to achieve the desired design state. The directive is used to properly sequence the rules together. Two examples of a serial macro are *rule-durand-wall-generate* and *rule-durand-platform-make-ring*. The first macro generates the wall from the parti. Here the macro *rule-durand-wall-generate* creates a group of parallel wall labeled lines and then trims off, in sequence, the three intersection conditions that occurs between the lines. The second example is the generation of the platform around the building outline using macro *rule-durand-platform-make-ring*. Each rule in the sequence places a platform against a known possible condition of the building outline. The void function in the zone descriptor is used to test for these conditions. By filleting and trimming all the platform lines generated, a complete platform is made around the entire building.

9.0 Future works

The meta-language changes the original formulas to be:

$$\text{For all t and g such that } p(t(g(A))) \leq s(C) \qquad (9)$$
$$C' = \sum (s(C) - p(t(g(A))) + t(g(B))) \qquad (10)$$

Where p() is the contextual requirements phase and s() is the drawing state phase. Notice in formula 10, the p() function is used only on schema A. An alternative is to use the p() function for both schema A and B.

$$C' = \sum (s(C) - p(t(g(A))) + p(t(g(B)))) \qquad (11)$$

Placing a p() function on schema B suggests that there is a possibility for a post application evaluation process. After subtracting schema A and adding schema B, the p() function of schema B must be evaluated to determine if the p() function is satisfied. Not only does schema A have to be a part of the drawing before applying the rule (formula 7), schema B also has to be a part of the drawing after applying the rule (formula 12).

$$p(t(g(A))) \leq s(C) \qquad (7)$$

$$p(t(g(B)))) \leq C' \qquad (12)$$

If the p() function is satisfactory, there are no other changes that need to be made to the drawing. If the p() function returns false, then the most likely outcome is that the rule becomes invalidated and the drawing is returned to its original state. Whether or not a p() function on schema B is useful or beneficial for the use of shape grammars is a topic for further discussion.

An interesting augmentation to the Hexagon Path grammar would be the inclusion of rules that adds intelligence to the game. Instead of having two players play against each other, one could play against a computer. This, of course, requires the use of a shape grammar interpreter that can define parametric shapes as well as find subshapes in a drawing. There are a variety of shape grammar interpreters out there but none that are robust enough to handle both requirements (Gips 1999). Adding intelligence will also most likely require another meta-language to be developed for shape grammars that includes common programming constructs such as loops and functions. This addition could transform shape grammars into a programming language for visual and symbolic computation.

Durand's method of design is a prescriptive method. The design process he describes is meant to be a step by step process for beginning engineers to understand how architecture is made. But when an architect designs for himself, he does not necessarily always follow a prescribed sequence of events. He may use the same steps but in a dynamic manner. Schön describes the design process as an iterative see-move-see cycle where the designer has a conversation with the drawing. The designer sees the drawing in some manner which triggers a move which in turn allows the user to see the drawing in yet another way. At one moment the designer may see something in the drawing that triggers a move to place something in a room. In another moment he may decided to alter the parti arrangement.

What I am suggesting is that the order of the stages in the grammar should be flexible. Instead of having a linear sequence of stages, the grammar should have randomly accessible stages. This gives the user freedom to change any portion of the design at any time. But there are two major difficulties designing a grammar in this manner.

With a linear progression of stages, the state of the drawing is predictable from stage to stage. Thus rules can be designed to key in on specific characteristics of the drawing. But if the progression of the design is done in a random manner, there are no predictable characteristics a schema can match to in the drawing. For example, in the Durand grammar, the Openings stage comes after the Room Columns stage. If the Openings stage is executed before the Room Columns stage, the rules in the Room Columns stage will not find any rooms. The reason why this occurs is because the rules to find a room look for a wall labeled rectangle. Creating an opening in the walls puts a gap in the wall labeled rectangles causing the rule to not recognize a room.

One possible solution is to define a room by the four corners of the room instead of the four walls. That way, the walls of a room can have multiple holes and still be recognized as a room. There are, of course, potential problems that can occur. Depending on the arrangement of the rooms, the four corners that get picked may be the four corners of four different rooms that just happen to be in alignment with each other. The problem is being able to define a schema that is general enough to encompass many different possibilities yet specific enough to define a type. How can a schema be described without relying on specific characteristics? How can a schema be described that is flexible enough to accommodate a variety of possibilities? Is there such a thing as a fuzzy schema?

The second problem is that altering some portion of the drawing can have repercussions in other areas of the drawing. For example, the Durand grammar proceeds in a top down manner where the overall structure of the design is first created using the axis and parti lines and then the details, such as walls, columns, and stairs, are added to the structure. While adding the architectural details, the user may realize that the overall structure is not big enough to accommodate the new details. In order to remedy the situation, the user needs to adjust the overall structure of the design. This requires going back to the beginning of the grammar and manipulating the parti and axis lines which the grammar can not do. One means of resolving this problem is to develop software that allows the user to go back in the derivation and alter the parameters used and propagate those changes (Nagakura, 1995). Another possible solution is to develop relational macros which maintain the relationships between certain elements in the design. Whenever one portion of the drawing is changed, the other related portions are changed as well. For example, if a parti line is added to a floor plan in the Durand grammar, then the relational macro would add and modify the walls to fit the new parti line. This idea is predicated on having a robust shape grammar interpreter which does not yet exist. But that will all change with time.

The solution to these problems will make shape grammars a powerful paradigm for digital design tools. Designers can describe their ideas graphically and create an instrument where rules are used to alter the design. A collection of these rules creates a grammar or a personal design toolkit which can be used to modify any drawing.

10.0 References

Agarwal M, Cagan J: 1998, “A blend of different tastes: the language of coffeemakers” *Environmental and Planning B* **25**: 205-226.

Cagan J, Mitchell W J: 1993, “Optimally directed shape generation by shape annealing” *Environmental and Planning B* **20**: 5-12.

Carlson C, Woodbury R: 1992, “Structure grammars and their application to design” *in* D.C. Brown, M. Waldron, and H. Yoshikawa (eds), *Intelligent Computer Aided Design*, Elsevier Science Publishers, Amsterdam, pp. 107-132.

Chase S: 1989, “Shapes and shape grammars: from mathematical model to computer implementation” *Environmental and Planning B* **16**: 215-242.

Chase S: 2002, “A model for user interaction in grammar-based design systems” *Automation in Construction* **11**(2): 161-172.

Chiou S, Krishnamurti R: “The grammar of Taiwanese traditional vernacular dwellings” *Environmental and Planning B* **22**: 689-720.

Crowe N: 1995, *Nature and the Idea of a Man-made World*, MIT Press, Cambridge, Massachusetts.

Duarte J: 2001, *Customizing Mass Housing: A Discursive Grammar for Siza’s Malagueira Houses*. PhD Dissertation, Department of Architecture, Massachusetts Institute of Technology, Cambridge, Massachusetts.

Durand J: 2000, *Précis of the Lectures on Architecture*, Translated by David Britt, Getty Research Institute.

Flemming U: 1987, “More than the sum of parts: the grammar of Queen Anne houses” *Environmental and Planning B* **14**: 323-350.

Flemming U: 1994, “Get with the program: common fallacies in critiques of computer-aided architectural design” *Environmental and Planning B* **21**: S106-S116.

Frampton K: 2001, *Le Corbusier*, Thames & Hudson, New York.

Gips, J: 1999, “Computer Implementation of Shape Grammars” invited paper, Workshop on Shape Computation, MIT.

Knight T: 1994, *Transformations in Design*, Cambridge University Press, New York.

Knight T: 2003, Computing with emergence, *Environmental and Planning B* **30**: 125-155.

Koning H, Eizenburg J: 1981, “The language of the prairie: Frank Lloyd Wright’s prairie houses” *Environmental and Planning* B **8**: 295-323.

Kruft H: 1994, *A History of Architectural Theory From Vitruvius To The Present*, Princeton Architectural Press, New York.

Lawson B: 1997, *How Designers Think: The Design Process Demystified*, Architectural Press, Boston.

Li A: 2001, *A Shape Grammar for Teaching the Architectural Style of the Yingzao Fashi*. PhD Dissertation, Department of Architecture, Massachusetts Institute of Technology, Cambridge, Massachusetts.

Li, A: 2002, A prototype interactive simulated shape grammar, in K Koszewski and S Wrona (eds), Connecting the Real and the Virtual - design e-ducation, Proceedings of the 20th Conference on Education in Computer Aided Architectural Design in Europe, eCAADe, Warsaw, pp. 314-317.

Liew, H: 2002, Descriptive Conventions for Shape Grammars, in G Proctor (ed), Thresholds - Design, Research, Education and Practice, in the Space Between the Physical and the Virtual, Proceedings of the 2002 Annual Conference of the Association for Computer Aided Design In Architecture, ACADIA, Pomona, pp. 365-378.

Liew, H: 2003, SGML: A Shape Grammar Meta-Language, in W Dokonal and U Hirschberg (eds), Digital Design, Proceedings of the 21st Conference on Education in Computer Aided Architectural Design in Europe, eCAADe, Graz, pp. 639-647.

Nagakura T: 1995, *Form-processing: A System for Architectural Design*, PhD Dissertation, Harvard University, Cambridge, Massachusetts.

Mitchell W: 1990, *The Logic of Architecture*, MIT Press, Cambridge, Massachusetts.

Posner M: 1980, “Orienting of attention” *Quarterly Journal of Experimental Psychology* **32**: 3-25.

Rowe C: 1976, *The Mathematics of the Ideal Villa and Other Essays*, MIT Press, Cambridge, Massachusetts.

Rykwert J, Leach N, Tavernor R: 1988, *On the Art of Building in Ten Books*, MIT Press, Cambridge, Massachusetts.

Schön D: 1983, *The Reflective Practitioner: How Professionals Think in Action*, Basic Books, New York.

Schön D, Wiggins G: 1992, “Kinds of Seeing and Their Functions in Designing” *Design Study* **13**(2): 135-156.

Stiny G: 1977, “Ice ray: a note on the generation of Chinese lattice designs” *Environmental and Planning B* **4**: 89-98.

Stiny G: 1980a, “Introduction to shape and shape grammars” *Environmental and Planning B* **7**: 343-351.

Stiny G: 1980b, “Kindergarten grammars: designing with Froebel’s building gifts” *Environmental and Planning B* **7**: 409-462.

Stiny G: 1981, “A note on the description of designs” *Environmental and Planning B* **8**: 257-267.

Stiny G: 1990, “What is a design?” *Environmental and Planning B* **17**: 97-103.

Stiny G: 1991, “The Algebras of Design” *Research in Engineering Design* **2**: 171-181.

Stiny G: 1992, “Weights” *Environmental and Planning B*, **19**: 413-430.

Stiny G: 2001, “How to calculate with shapes” (working paper), MIT Department of Architecture.

Stiny G, Mitchell W: 1978, “The Palladian Grammar” *Environmental and Planning B* **5**: 5-18.

Twombly R, Menocal N: 2000, *Louis Sullivan: The Poetry of Architecture*, W. W. Norton & Company, New York.

Tzonis A, Lefaivre L: 1986, *Classical Architecture: The Poetics of Order*, MIT Press, Cambridge, Massachusetts.

Villari, S: 1990, *J.N.L. Durand (1760-1834) Art and Science of Architecture*, Rizzoli International Publications, New York.