Description grammars

A general notation

Stouffs, Rudi

DOI
10.1177/0265813516667300

Publication date
2016

Document Version
Accepted author manuscript

Published in
Environment and Planning B: Planning & Design

Citation (APA)
Stouffs, R. (2016). Description grammars: A general notation. *Environment and Planning B: Planning & Design*, *45 (2018)*(1), 106-123. https://doi.org/10.1177/0265813516667300

Description grammars: a general notation

Abstract

A description grammar, in conjunction with a shape grammar, serves to generate verbal descriptions of designs, next to the spatial descriptions. These verbal descriptions can also assist in guiding the generative process. This paper presents a general notation for descriptions and description rules that accounts, extensively if not entirely, for many of the applications of description grammars found in literature. Specifically, a review of the notation with respect to these description schemes supports the explication of its strengths and limitations and the identification of future work. A follow-up paper will revisit selected applications of description grammars and demonstrate the applicability of this general notation to these case studies.

Introduction

"Designers work with descriptive devices of many kinds. These may be spatial or symbolic" (Stiny, 1991, p. 171). Descriptions may serve to compare designs to find similarities and dissimilarities. Descriptions may also be generated. Shape grammars have been used extensively for both. Shape grammars are a formal rewriting system for producing languages of shapes (Stiny, 1980); they have been used, to name just a few, by Stiny and Mitchell (1978) to generate Palladio's villa ground plans as a partial definition of the Palladian style, by Downing and Flemming (1981) to allow differences between bungalows of Buffalo "to be explained as different geometric realizations of a shared set of conventions," and by Çağdaş (1996) to characterize formal compositional aspects of traditional Turkish houses.

When we describe architecture, we are interested in both the description of the specific architectural object and its relation to other, similar architectural objects. Although shape grammars have been extensively used for this purpose, shape descriptions of architectural objects can be deficient. Stiny (1981, p. 257) noted that "main details of the functional elements comprising designs in these languages are provided in the informal, verbal descriptions of the shape [rewriting] rules used." To address this deficiency, Stiny proposed to augment a shape grammar with a description function in order to construct the verbal descriptions of designs. He illustrated the application of a description function with designs made up of blocks from Froebel's building gifts. However, he indicated that the formal representation of descriptions, together with the descriptions themselves, would "likely have to be worked out on a case-by-case basis" (Stiny, 1981, p. 258).

About thirty-five years onwards, we can find more than a few applications in literature of a description function, often denoted a description grammar, in conjunction with a shape grammar, to qualify designs both spatially and descriptionally. While similarities can be discerned, especially among researchers who worked closely together, a formal representation of descriptions was still lacking until now. Moreover, few of these applications include an implementation of the description function or grammar. Only Duarte and Correia (2006) describe the implementation of a description grammar—codifying the Portuguese housing design guidelines—and they specifically encode (hard-code) the description rules in order to handle custom description structures. Later, Duarte et al (2012, p. 84) identify the lack of a description grammar interpreter as one of two reasons for adopting a different strategy considering an ontology to represent urban program formulation rules and an ontology editor as the rule interpreter, the other reason being the complexity of the urban formulation problem.

This lack of a description grammar interpreter may hamper the development of sound description grammars. Eloy (2012a) reflects on a grammar implementation for housing rehabilitation. She acknowledges that the description rules were first developed in an abbreviated form, which "proved to be insufficient in terms of implementing the grammar in computer software since it does not have all the information required" (Eloy 2012a, p. 320). She subsequently defined a detailed description but, admitting to a purely manual elaboration, she only elaborated a few sample rules and illustrated a few derivational steps (Eloy, 2012b, p. 150). Similarly, Correia (2013)—while describing the implementation of a shape grammar interpreter for Duarte's (2001) Malaguiera grammar—reflects on the difficulty of implementing the grammar's description rules. He indicates "many ambiguities and even some errors" and illustrates these with a few examples (Correia, 2013, pp. 60-61).

Neither the availability of a formal representation, nor the availability of a description grammar interpreter, will ensure a painless elaboration or a completely accurate grammar. However, the constraints that a formal representation imposes, and the ability to test the specification of description rules with a description grammar interpreter, can yield a quicker and better understanding of the requirements for a consistent representation and a robust rule set. Despite Stiny's (1981) conjecture that a formal representation of descriptions likely has to be worked out on a case-by-case basis, we aim to demonstrate that a formal representation, with corresponding description grammar interpreter, is able to account extensively, even if not entirely, for many of the existing applications of description grammars and, thus, can serve the development of new applications.

Recent work has yielded an extensive overview of applications of description grammars (or functions) in literature (reference withheld), a formal notation for descriptions and

description rules that builds upon this overview, and the implementation of this notation in a description grammar interpreter (reference withheld). Absent is a thorough analysis of the formal notation with respect to the breadth of applications of description grammars and the variability in concepts, components and notations. Also lacking are detailed studies—other than Stiny's (1981) example illustrating the application of a description function with designs made up of blocks from Froebel's building gifts (reference withheld)—of how some of these applications found in literature can be recast and redeveloped to make optimal use of the available notation and implementation.

This paper exactly addresses this lacuna, focusing on qualifying the generality of the notation and on addressing its limitations. Firstly, we briefly present the overview of description schemes in literature as a foundation for subsequent analysis. Secondly, we review the formal notation and explicate its strengths and limitations with respect to the description schemes presented. Finally, we discuss any omissions. In a follow-up paper, we will revisit these description schemes and demonstrate how they can be recast and redeveloped to make use of the available notation and implementation. This is both meant as an illustration and as a confirmation of the analysis results.

Description schemes

Stiny references his 1981 article in quite a few subsequent articles, however, most references only touch upon the subject of shape descriptions, other than visual descriptions, or, alternatively, extend on the subject of parallel descriptions, – and grammars – a corollary of the description function. Only once does Stiny (2006) revisit the topic with a new example, though he twice (Stiny, 1987, p. 182; March and Stiny,

1985) alluded to a forthcoming paper that would treat description schemes in more detail. However, including Stiny's original paper, we can identify at least eighteen distinct accounts of description schemes or their illustrations presented in about forty publications. We're omitting any description schemes that rely on (spatial) topological, ontological or graph structures that require specific, non-textual representational structures and, as such, cannot be easily accounted for by a textual notation.

Not all accounts offer the same level of detail in describing either a description scheme or its illustration. Nevertheless, all are included to retain completeness. The eighteen accounts can be classified according to their role in the grammatical design generation process: as reflecting on the spatial elements and their composition, as expressing some property, such as volume, cost or manufacturing plan, as a design brief and as a generative guide (other than design brief).

Descriptions as reflections

Stiny (1981) proposes a description function in order to construct intended descriptions of designs. His descriptions reflect on the spatial elements—made up of blocks from Froebel's building gifts—that constitute the design and the way these are combined. These descriptions are derived from the generation process and, as such, do not impose any conditions on the respective shape descriptions. Furthermore, Stiny's (1981) functions are not explicitly dependent on the shape rules they reflect upon. Though they collect coordinate pairs specifying boundary points of (linear) wall elements, the relative coordinates of subsequent coordinate pairs are hardcoded in the functions, considering a distance of one between adjacent boundary points.

Li (2001; also, 2004) applies a description function to the specification of a shape grammar for (teaching) the architectural style of the *Yingzao fashi* (Chinese building

manual from 1103). The descriptions that are generated are taken from the annotated *Yingzao fashi* (Liang, 1983) and, similarly to Stiny (1981), the descriptions reflect on the composition of spatial elements that constitute the design. Li considers various descriptions (nine in total, specifying measures and descriptions of width, depth, height), as well as drawings (seven, from plan diagram to plan, section and elevation), in parallel.

Zamenopoulos (2012) considers the mathematical characterization of the organizational complexity of intentionality and proposes a category theoretic account of the semantic content of design intentionality, using descriptions of shape configurations to express interpretations of languages of designs. Note that Zamenopoulos only exemplifies descriptions, not the underlying description rules.

Descriptions as expressions

A few authors consider description functions in the context of spatial grammars applied to mechanical engineering. These accounts invariable consider descriptions as expressing some property, such as volume, cost or manufacturing plan.

Brown et al (1996; also, Brown and Cagan, 1997) consider a description function that generates process plans for the manufacturing of objects with a turning tool. The objects themselves are generated by a parametric attributed set grammar, but redefining the grammar instead as a shape grammar (with constraint specifications) would not impact the description function as such. Separately, Brown (1997) exemplifies volume calculation as a description function for a grammar specifying a language of stepped grooved shafts. In contrast to Stiny (1981), both description schemes consider description rules that are explicitly dependent on the conjunctive shape rules. For example, Brown's (1997) description rules for volume calculation require the shape rule

to provide values for the diameter and length of the section when adding a new section to the shaft.

Agarwal (1999; also, Agarwal et al, 1999) considers a description function that yields cost expressions or equations that can be evaluated to reveal the cost of a design as the design develops through the generation process. While these expressions make explicit reference to characteristics of the shape under rule application, such as its dimensions, these are not evaluated during rule application. Instead, the cost expressions are intended to be evaluated on the corresponding shape again and again, at any time during the generation process, in order to assess the evolving cost. These cost assessments can be used to gain insight into how design changes affect the cost and thus providing feedback on the generation process itself; but it can also be used to guide the generation process by cost preferences or constraints.

Descriptions as design brief

Duarte (2001; also, 2005a) considers a discursive grammar to incorporate a shape grammar, a description grammar and a set of heuristics, at least from a technical viewpoint. The use of heuristics is intended to constrain the rules that are applicable at each step of the design generation. From an operational viewpoint, a discursive grammar combines a programming grammar generating design briefs based on user and site data and a designing grammar using the design brief(s) to generate designs in a particular style. Both programming grammars and designing grammars utilize description grammars, though only the designing grammar complements the description grammar with a shape grammar. Duarte and colleagues apply discursive grammars to the Portuguese housing program guidelines and evaluation system (PAHP) and the houses designed by the architect Alvaro Siza at Malagueira (Duarte, 2001), to urban

design (Beirão, 2012) and to housing rehabilitation (Eloy, 2102a; 2012b; also, Eloy and Duarte, 2014). Duarte (2005b; also, 2001) presents a simplification of the Malagueira designing grammar, with descriptions representing functional zones and their adjacency relations.

Descriptions as generative guide

Many other accounts consider descriptions as generative guide, other than design briefs.

Knight (2003) proposes state descriptions to guide an optimization process. A compound shape/description rule specifies a fitness function that computes the state description from the shape under consideration. The actual shape is left unchanged in the application of this rule. Only if the resulting state description is, e.g., 1, another (compound) shape rule will consequently modify the shape. Knight (2003) proposes the fitness function to encode an algorithm that directly accesses the actual shape under rule application.

Liew (2004) proposes a number of 'descriptors' to guide the rule application process, one of which—the Zone descriptor—considers an application of Knight's (2003) functions encoding algorithms. For example, a 'void' function checks whether a specified area is void of all shapes, and an 'exclude' function checks whether a specified area excludes shapes from a specified list of shapes. Liew (2004) also considers a Rule-set descriptor, where the description specifies a set of rule labels such that a rule only applies if the rule label is present in the current description.

Stiny (2006) presents description rules for Palladian villa plans that count the number of rooms and assign plans to equivalence classes. He explores the use of such descriptions to set goals to guide and control the design process. Ahmad (2009; also, Ahmad and

Chase, 2006) proposes to augment a shape grammar with a style description scheme based on the concept of semantic differential to map the style characteristics of shape rules. While these style descriptions do not directly guide the generative process, they do serve to guide the grammar transformation process for the purpose of stylistic change.

Al-kazzaz (2011; also, Al-kazzaz et al, 2010) considers descriptions in shape grammars for hybrid design, where the descriptions provide feedback on rule application based on comparisons between the generated design and the antecedents in the corpus. Additionally, he considers a user guide specified as sets of antecedent labels. Muslimin (2013) uses descriptions to investigate how meaning is embedded in the Passura' carvings of the Toraja people in South Sulawesi, Indonesia. He exemplifies the descriptions both as a result of the shape generation and as a guide to the shape generation. Note that neither Al-kazzaz (2011) nor Muslimin (2013) offer any explication of description rules, describing them only conceptually or as rule types.

Coutinho (2014) considers descriptions from Alberti's *De re aedificatoria* to guide the generation of Alberti's column system. Finally, Stouffs and Tunçer (2015) consider a description scheme to generate an instance of a historical architectural typology from an ontological description thereof. Descriptions come in two forms, as an XML description and as a set of ontological terms.

A formal notation

(reference withheld) presents a formal notation for descriptions and description rules (Table 1). Here, we offer a synopsis of this notation and explicate its strengths and limitations with respect to the description schemes presented above.

Table 1: Formal notation for descriptions and the left-hand-side (lhs) and right-hand-side (rhs) of description rules in Extended Backus-Naur-Form (EBNF), including examples. The same non-terminals serve to define the production rules for a description, an lhs, and an rhs. Only when necessary are alternative production rules defined for the same non-terminal; these are then identified by adding the terms *description*, *lhs*, and *rhs*, respectively, enclosed within angle brackets (‘<...>’), as a prefix to the respective production rule.

description = description-entity | description-sequence . description-entity = literal | top-level-tuple . description-sequence = ‘#’ description-entity ‘#’ { description-entity ‘#’ } .
literal = keyword-literal | number | string . keyword-literal = ‘e’ | ‘nil’ | ‘pi’ | ‘true’ | ‘false’. number = [‘–’] digit-sequence [‘.’ digit-sequence] . digit-sequence = digit { digit } . digit = ‘0’ | ‘1’ | ‘2’ | ‘3’ | ‘4’ | ‘5’ | ‘6’ | ‘7’ | ‘8’ | ‘9’ . string = ‘“’ { string-character } ‘”’ . string-character = any-character-except-quote | ‘\’ ‘“’ .

Example **description-entity**:
`“centrally divided, double 1-rafter beam in front and back”`
 	
Example **description-sequence**:
`#e#0#“nothing”#`

top-level-tuple = tuple | unmarked-tuple . tuple = ‘(’ tuple-entities ‘)’ | ‘<’ [tuple-entities] ‘>’ | ‘[’ [tuple-entities] ‘]’ .

<description>tuple-entities = tuple-entity-sequence .

<lhs>tuple-entities = tuple-entity-sequence | tuple-expression .

<rhs>tuple-entities = tuple-entity-sequence | tuple-expression .

tuple-entity-sequence = tuple-entity ({ ‘,’ tuple-entity } | { ‘;’ tuple-entity }) .

<description>tuple-entity = literal | tuple .

<lhs>tuple-entity = numeric-expression | string-expression | tuple .

<rhs>tuple-entity = numeric-expression | string-expression | tuple | function-returns-tuple .

unmarked-tuple = tuple-expression | tuple (tuple | keyword-literal) { tuple-entity } .

Example **tuple**:

```
("l:", 10, "c:", (0, 0), "r:", 0)
```

Example **unmarked-tuple**:

```
<" ", "O", "R0", "R1"> <"O", 1, 1, 1> <"R0", 1, 1, 0> <"R1", 1, 0, 1>
```

description-rule-side = description-rule-entity | description-rule-sequence .

<lhs>description-rule-entity = literal | parameter [‘?’ conditional] | string-expression | top-level-tuple .

<rhs>description-rule-entity = numeric-expression | string-expression | function-returns-tuple | tuple-expression .

description-rule-sequence = ‘#’ description-rule-entity ‘#’ { description-rule-entity ‘#’ } .

parameter = identifier .

identifier = (letter | underscore) { (letter | underscore | digit) } .

letter = ‘A’ | ‘B’ | ‘C’ | ‘D’ | ‘E’ | ‘F’ | ‘G’ | ‘H’ | ‘I’ | ‘J’ | ‘K’ | ‘L’ | ‘M’ | ‘N’ | ‘O’ | ‘P’ |

‘Q’ | ‘R’ | ‘S’ | ‘T’ | ‘U’ | ‘V’ | ‘W’ | ‘X’ | ‘Y’ | ‘Z’ | ‘a’ | ‘b’ | ‘c’ | ‘d’ | ‘e’ | ‘f’ | ‘g’ | ‘h’ | ‘i’ | ‘j’ | ‘k’ | ‘l’ | ‘m’ | ‘n’ | ‘o’ | ‘p’ | ‘q’ | ‘r’ | ‘s’ | ‘t’ | ‘u’ | ‘v’ | ‘w’ | ‘x’ | ‘y’ | ‘z’ .

underscore = ‘_’ .

Example **<lhs>description-rule-entity**:

```
<"Fixed", var1> <var2, var3> remainder
```

Example **description-rule-sequence**:

```
#a1#a2#a3#a4#a5#a6#a7#a8#
```

conditional = enumeration | equation .

enumeration = ‘{’ (number-sequence | string-sequence) ‘}’ .

number-sequence = number { ‘,’ number } .

string-sequence = string { ‘,’ string } .

equation = comparator comparand .

comparator = ‘=’ | ‘<>’ | ‘<’ | ‘<=’ | ‘>’ | ‘>=’ .

comparand = number | ‘(’ numeric-expression ‘)’ | parameter | reference .

Example **<lhs>description-rule-entity** with **enumeration**:

```
yard?{nil, "default"}
```

Example **<lhs>description-rule-entity** with **equation**:

```
<nrooms?>2, rooms>
```

numeric-expression = term { addition-operator term } .

term = factor { multiplication-operator factor } .

factor = base { exponentiation-operator exponent } .

exponent = base .

base = keyword-literal | number | ‘(’ numeric-expression ‘)’ | function-returns-number |

parameter | reference .

exponentiation-operator = '^' .

multiplication-operator = '*' | '/' | '%' .

addition-operator = '+' | '–' .

Example **numeric-expression**:

```
vol – pi^2 * radius * (length / 2)^2 + 4 / 3 * pi * (length / 2)^3
```

string-expression = string-expression-entity { '.' string-expression-entity } .

<lhs>string-expression-entity = literal | parameter ['?' conditional] .

<rhs>string-expression-entity = base | string | function-returns-string .

Example **<rhs>string-expression**:

```
"with ".(c + 1)." columns"
```

Example **<lhs>string-expression**:

```
"with ".c?=(be21 + be22)." columns"
```

<lhs>tuple-expression = tuple-append | tuple-prepend .

<rhs>tuple-expression = tuple-addition | tuple-extension .

tuple-append = { tuple-entity } parameter ('*' | '+') tuple-entity { tuple-entity } [tuple-expression] .

tuple-prepend = [tuple-expression] { tuple-entity } tuple-entity parameter ('*' | '+') { tuple-entity } .

tuple-addition = [parameter] '+' basic-tuple-argument .

tuple-extension = { tuple-entity } parameter { tuple-entity } [tuple-expression] .

Example **tuple-prepend**:

```
h1 h2 H*
```

Example **tuple-extension**:

```
a1 last(a1) + (0, 1)
```

Example **tuple-addition**:

```
bedrooms + <1, [("couple", 0), ("double", 0), ("single", 1)]>
```

function = function-returns-number | function-returns-string | function-returns-tuple .

function-returns-number = numeric-function | length-function | string-function-untyped | tuple-function-untyped .

numeric-function = (‘sqrt’ | ‘sin’ | ‘cos’ | ‘tan’) ‘(’ numeric-expression ‘)’ .

length-function = ‘length’ ‘(’ (string-argument | tuple-argument) ‘)’ .

<lhs>string-argument = string | function-returns-string | parameter | reference .

<rhs>string-argument = string-expression .

function-returns-string = string-function-returns-string | string-function-untyped | tuple-function-untyped .

string-function-returns-string = (‘left’ | ‘right’) ‘(’ string-argument ‘,’ numeric-expression ‘)’ .

string-function-untyped = ‘eval’ ‘(’ string-argument ‘)’ .

tuple-function-untyped = (‘first’ | ‘last’ | ‘min’ | ‘max’) ‘(’ tuple-argument ‘)’ .

<lhs>tuple-argument = basic-tuple-argument .

<rhs>tuple-argument = basic-tuple-argument | tuple-expression .

basic-tuple-argument = tuple | function-returns-tuple | parameter | reference .

function-returns-tuple = tuple-function-returns-tuple | string-function-untyped | tuple-function-untyped .

tuple-function-returns-tuple = (‘unique’ | ‘segments’ | ‘pairwise’ | ‘loops’) ‘(’ tuple-

argument ‘)’ | ‘adjacencies’ ‘(’ tuple-argument ‘,’ tuple-argument ‘)’ .

Example **function-returns-number**:

```
length("room")
```

Example **function-returns-tuple**:

```
adjacencies(a4, a5 a6)
```

reference = reference-to-lhs | reference-to-rhs .

reference-to-lhs = [‘lhs.’] reference-designator ‘.’ (‘value’ | parameter | property) [‘:’ filter] .

reference-to-rhs = ‘rhs.’ reference-designator ‘.’ property [‘:’ filter] .

reference-designator = identifier .

property = identifier .

filter = reference-designator ‘.’ property filter-operator (number | vector | string) .

filter-operator = ‘=’ | ‘<>’ | ‘<=’ | ‘>=’ .

vector = [rational] ‘(’ rational ‘,’ rational ‘,’ rational ‘)’ .

rational = [‘–’] digit-sequence [‘/’ digit-sequence] .

Example **reference-to-lhs**:

```
indices.value
```

Example **reference-to-rhs**:

```
rhs.sections.radius:labels.label="S"
```

Descriptions

While most description schemes consider multiple descriptions handled in parallel (e.g., Li, 2001; Duarte, 2001), Stiny (1981) considers descriptions containing multiple sections separated by the ‘#’ symbol. Thus, a description is formally specified either as

a single description entity, that is, a literal or a tuple, or as a sequence of description entities separated and enclosed by the '#' symbol. A literal may be a number, a (double quoted) string or a literal identifier. The latter include *e*, *nil*, *pi*, *true* and *false*. *E* and *nil* are equivalent, both define an 'empty' entity, that is, zero, an empty string, or an empty tuple; *e* is used by Stiny (1981), *nil* by Duarte (2001). Note that Li (2001), Duarte (2001; 2005b), Beirão (2012) and Eloy (2012a) also adopt the symbol 'Ø' for an empty tuple. We suggest the use of *e* or *nil* instead. The literal *pi* represents the number 'π'. The literals *true* and *false* represent 1 and 0, respectively, in conformity with Duarte (2001).

Numbers

Numbers are non-controversial. Stiny (1981; 2006), Li (2001) and Beirão (2012) consider descriptions expressed as integers, e.g., for counting. Brown (1997), Duarte (2001), Al-kazzaz (2011) and Beirão (2012) consider descriptions expressed as real or floating-point numbers, specifying area, volume or cost values, among others. Many authors consider numbers as part of tuples.

Strings

Strings are slightly more contentious. Li (2001) considers triples of alphanumeric descriptions specifying the disposition of beams. Each part is constructed through concatenation and replacement of smaller alphanumeric (or numeric) entities. Similarly, Stouffs and Tunçer (2015) consider an alphanumeric description—expressed in XML—built up through concatenation and replacement. Many other description schemes consider alphanumeric terms, however, these are fixed terms—though a description rule may replace one term by another—that are never combined through concatenation, only collected in tuples. Duarte (2001) considers names of people; Brown et al (1996) consider labels for description entities to improve readability; Duarte (2001; 2005b),

Ahmad (2009), Al-kazzaz (2011), Beirão (2012), Eloy (2012a) and Stouffs and Tunçer (2015) all consider enumerations of terms, for example, denoting functions, spaces, qualifications, rule labels, or ontological terms.

Only Stouffs and Tunçer (2015) use (double) quotes to identify alphanumeric description entities. Li (2001) omits any quotes, as well as enclosing brackets and separation marks for the triples, allowing the description to be read either as a single statement or as a triple of strings. While we appreciate such informality from a human's point of view, double quotes are required to identify strings for machine-readability. We anticipate that this explicit notation can always be parsed and presented in a more human readable form. While strictly speaking, enumerated terms do not require quotes to be recognized as such, since all enumerations found are grammar-specific in nature—with exception of true and false—we require all terms to be quoted, eliminating the need to predefine any enumerations. We reserve unquoted terms for literal identifiers, parameters, references and function designators (see later).

Tuples

A tuple is a sequence (or list) of description entities. Tuples can be nested. Most description schemes adopt tuples; for example, Stiny (1981) considers, among others, coordinate pairs and tuples of coordinate pairs; Li (2001) considers tuples of integers and triples of strings; Duarte (2005) implies the use of a single tuple structure, combining a number of different entity types, where a general description contains a number of instances of this tuple structure; Brown et al (1996), on the other hand, considers the use of a single tuple with fixed length at the top level, where each entity in the tuple expands into a tuple of arbitrary length and, possibly, a nested tuple of tuples.

There exists a large variety in how description schemes present tuples; yet, there are also many similarities. They variably use parentheses, angle brackets and square brackets as enclosing brackets to identify tuples, and commas or semicolons to separate elements within a tuple. Sometimes, enclosing brackets are omitted altogether, but only at the top level of a nested tuple structure, and, in a few cases, separation marks are omitted as well, leaving only spaces to separate the elements.

In contrast to strings, that potentially may contain any kind of characters and tokens, it is possible to accommodate all these notational variations for tuples and to consider disambiguation rules where and when necessary. For example, a minus sign separating two numerical entities is interpreted as a subtraction, instead of as a unary negation within a tuple of (at least two) numbers, with separation marks omitted. As another example, a parenthesized expression of a single description entity is interpreted simply as this entity, instead of as a tuple. A tuple of length one (or zero) is only recognized as such if it is enclosed within angle or square brackets, a practice all authors adopt.

Left-hand-side of a description rule

Any description can form the left-hand-side (lhs) of a description rule – either a single description entity, that is, a literal or a tuple, or a sequence of description entities separated and enclosed by the '#' symbol. Additionally, a parameter or a string expression can be a substitute for a literal or serve as entities within a tuple.

Parameters

A parameter is a variable term; it is matched to a literal or a tuple in a description under rule application. If the parameter forms part of a string expression, this literal can be any part of a literal string (see 'string expressions'). If the parameter forms part of a tuple, it matches a specific element of the tuple, unless it is signified by a kleene star ('*') or a

kleene plus (‘+’), in which case it can match any subsequence of elements of the tuple, respectively, including or excluding an empty subsequence.

Authors commonly adopt a convention to identify parameters from other alphanumeric components. For instance, Li (2001) uses single, lowercase letters, possibly with a superscript number, in italics to denote parameters, e.g., *a*1; Duarte (2001) distinguishes parameters from enumerated terms using single, uppercase letters (possibly followed by a number), e.g., F1. Since we require strings (including enumerated terms) to be double quoted, any identifier other than literal identifiers, function designators and reference designators—all of which are known ahead of time—is assumed to denote a parameter.

Some description schemes consider description tuples of variable length, containing any number of elements. These elements typically adhere to the same structure, i.e., they are all numbers, all strings, or all tuples of the same length and with corresponding element types. Different from fixed-length tuples, elements from variable-length tuples cannot be matched to individual parameters without the use of an additional construct—such as the kleene star or kleen plus signifiers—, as the length of the tuple may not be known in advance. For example, Stiny (1981) considers variable-length tuples of coordinate pairs, and rules (denoted functions) that identify—though not explicate—the last coordinate pair from the tuple (see ‘functions’). Eloy (2012a) considers a variable-length tuple but—informally—identifies only the individual elements of concern in the lhs of the description rules. An alternative and better approach would be to consider the tuple instead as a set of description instances, each element specifying a single instance. The ordering of the elements/instances is then no longer important.

Brown et al (1996) suggest a notation borrowed from logic programming (using the separator ‘|’), in order to distinguish the first element or elements from the remainder of

the tuple. Additionally, they consider a function to reverse a tuple, applied in the right-hand-side of a description rule, so as to allow the subsequent distinguishing of the last element or elements of the original tuple. The only disadvantage of this notation is that it emphasizes the prepending of elements over the appending of elements. Since the kleene star signifier avoids this asymmetry, we ignore Brown et al's (1996) suggestion.

Conditionals

Li (2001), Duarte (2001), Beirão (2012) and Eloy (2012a) all consider conditional specifications that constrain rule application and cannot simply be captured in an explication of the lhs of the rule. For instance, a description rule may apply in a number of different cases that correspond to different values for a single description entity. Short of specifying different rules corresponding the different values, which could work in the case of an enumeration but would fail in the case of a real numeric interval, conditional specifications may allow a parameter to be constrained beyond a single value. For example, Duarte (2001), Beirão (2012) and Eloy (2012a) present numerous examples where parameters can take a limited set of values. Li (2001) and Eloy (2012a) both consider numerical conditions constraining one numeric value in function of another numeric value, or values, all part of the same description. Brown et al (1996) also consider rule variants that include conditional specifications; however, these can easily be captured in a further explication of the lhs of the rule.

Any parameter may be specified a conditional that constrains the possible values of this parameter. This conditional may be either enumerative or equational. An enumerative conditional explicates a finite set of possible values—either all numbers or all strings—, for instance, an enumeration of terms. An equational conditional specifies a numeric equality or inequality on the parameter, in the form of a conditional operator ('=', '<>', '<', '<=', '>' or '>=') and operand. The operand must be either a number or a numerical

expression operating on numbers, parameters—previously defined—, functions and/or references. Functions and references are addressed later.

We opt to integrate conditionals within the description, rather than specifying them separately. The conditional expressions are also restricted in form; a parameter value is simply compared with a single value, a set of values, or a range of (numeric) values. These comparison values must be either literal values or computable as literal values; that is, any parameters within a conditional expression must have been previously matched—where matching occurs from left to right in the lhs of the description rule. These restrictions seem to concur largely with examples of conditional specifications adopted within example description schemes, even if the conditional must be reformulated and formatted according to the required notation. In a subsequent paper, we will revisit some of the eighteen descriptions schemes for a more detailed assessment (forthcoming).

String expressions

A string expression enables the identification of substrings in the matching process. A string expression is a concatenation of literals and parameters (with or without conditional). A parameter can match any substring, conditioned by the literal components (and the conditional, if present). A concatenation of two parameters, without a literal separating the two parameters, is not possible, unless the first parameter has an enumerative conditional.

Only Li (2001) and Stouffs and Tunçer (2015) consider string expressions. Besides omitting quotes, Li also omits any explicit concatenation operator. However, the use of an explicit concatenation operator (‘.’) is necessary in order to distinguish string concatenations from tuples of strings.

Right-hand-side of a description rule

Any description can form the right-hand-side (rhs) of a description rule. Additionally, an rhs can include parameters (without signifiers and conditionals), references, numerical expressions, string expressions, tuple expressions and functions. Parameters have been addressed previously; when used in the rhs of a description rule, the same parameter must also occur in the lhs of the same description rule and the parameter value will be the literal(s) matched to the parameter during rule application. We will address references as last.

Numerical expressions

A numerical expression can operate on literal identifiers, numbers, numerical functions, parameters and references. Stiny (1981; 2006), Li (2001) and Beirão (2012) consider operations of addition, subtraction and/or multiplication, on integers. Brown (1997), Duarte (2001), Al-kazzaz (2011) and Beirão (2012) consider mathematical operations on real or floating-point numbers, including division and exponentiation. We consider the operators plus ('+'), minus ('–'), times ('*'), divided-by ('/'), modulo ('%') and to-the-power-of ('^'), with the usual operator precedence rules applying, and the use of parentheses to override these rules where necessary. Other operations are considered in the form of numerical functions, including square root (*sqrt*), *sin*e, *cos*ine and *tan*gent.

String expressions

String expressions in the rhs of a description rule can include, other than literals and parameters (excluding signifiers and conditionals), references, numerical expressions (when enclosed in parentheses) and functions returning either numbers or strings. The result is the concatenation of all components upon their evaluation into literal numbers or strings. Li (2001) omits parentheses around numerical expressions, as well as the

concatenation operator. We require parentheses for the sake of readability, to visually collect the numerical expression as a single component within the concatenation. The use of parentheses is not required in the case of a single number, even if it is a floating-point number. Though both use the same symbol ('.'), a floating-point number takes precedence over a concatenation of numbers.

Tuple expressions

Common operators on tuples are append and prepend. For example, Stiny (1981) considers an append operation on tuples, simply using a space to separate the tuple and the element to be added. Brown et al (1996) consider an operation to prepend one or more elements to a tuple, using a shorthand notation borrowed from logic programming.

We adopt Stiny's notation and extend it to the operations of prepend and join as well. In order to distinguish these operations from simply omitting the separation marks within a tuple, like most authors, we require a structural similarity between, on the one hand, the tuple from the entity to be appended or prepended and, on the other hand, the entities within the larger tuple. For example, a number and a tuple of numbers will yield a tuple of numbers with the single number prepended (or appended, depending on the order). If the single entity is itself a tuple and the other tuple is a nested tuple, then the single entity will be prepended or appended to the nested tuple only if it has the same structure as the first element of the nested tuple. If both operands are nested tuples, and the elements of both tuples have the same structure, then a join operation will be assumed, combining the elements from both tuples in a new, single tuple. If no structural similarity can be determined, than the expression will instead be interpreted as specifying a tuple while omitting enclosing brackets and separator.

Duarte (2001) additionally proposes the addition of tuples that have the same structure. Adding two tuples adds the respective entities: if both entities are numbers they are summed; if both entities are strings (or enumerated terms in the case of Duarte (2001)) they must be identical; if both entities are tuples and have the same structure, then addition is applied recursively. Exceptionally, the first tuple may be omitted if the operation is at the top level of the rhs of the description rule (as exemplified by Duarte (2001)). In this case, the first operand of the addition operation is considered to be the literal tuple resulting from the matching of the lhs.

Functions

Functions can operate on numbers, strings and tuples, or a combination thereof, and return any one of these three entity types. Besides the numerical functions *sqrt*, *sin*, *cos* and *tan*, taking a single number as argument and returning a number, other predefined functions operate on strings and tuples. Functions operating on strings include determining the *length* of a string, and determining a *left* and *right* substring. The length of the substring is specified as an additional argument.

The functions operating on tuples are inspired by Stiny (1981). Though he does not explicate these as functions, he considers retrieving the last coordinate pair from a list, determining the number of distinct coordinate pairs in a tuple, retrieving the distinct number of adjacent coordinate pairs in a tuple, retrieving loops of coordinate pairs in a tuple, etc. These are generalized to functions determining the *length* of a tuple, retrieving the *first* or *last* element of a tuple, retrieving a tuple of only *unique* elements, a tuple of pairs (*segments*) such that the *i*th pair is made up of the *i*th and (*i*+1)th elements of the operand tuple, a tuple of tuples identifying the *loops* in the operand tuple, and a tuple of tuples representing an *adjacencies* matrix. The latter function takes two arguments, a tuple of ‘enclosures’ and a tuple of ‘connecting’ elements.

Duarte (2001) defines two grammar-specific functions, one that updates a tuple of 'current spaces' with user-prompted data about the solar orientation of the dwelling, another that allows the user to reset the relative weights of qualities and then normalizes their sum to one hundred. Brown et al (1996) define five functions, three of which are actually specified as description rules, though not (necessarily) operating on the same or similar description. These three are grammar-specific, the other two functions are more generally applicable: one function reverses a tuple, the other returns the maximum value from among the elements of the tuple. Knight (2003) and Liew (2004) propose functions encoding algorithms that operate on the shape under the conjunctive shape rule application.

Extending the collection of predefined functions is straightforward, e.g., with a function to reverse a tuple. User-defined functions can be specified as class methods, where the class is added to the implementation and the names of the class and method are provided to the function's definition. Then, the description grammar interpreter can automatically retrieve and apply the function to the given arguments, using computational reflection. User-defined functions that operate on the shape under the conjunctive shape rule application can be similarly defined, encoding the name of the grammar or drawing into the method to allow the method to retrieve the specified shape.

References

References are similar to parameters; they are also variable terms. However, whereas parameters must be defined within the lhs of the same description rule, references refer to parameters and values from other, parallel description rules. For example, Li (2001) considers nine parallel descriptions, specifying measures and descriptions of width, depth, height, and corresponding rule sets. A compound rule combines a number of description rules operating on different descriptions, thus, taken from different rule sets.

Within a compound rule, the rhs of a description rule may reference the current value of another description. Taking an example from Li (2001), one description counts the number of rafters, another description describes the disposition of the beams and includes the resulting number of rafters. Duarte (2001) similarly considers the rhs of a description rule to reference specific parameters within other, parallel descriptions. While both Li (2001) and Duarte (2001) simply specify the parameter in question, we include a designator for the parallel description grammar to precede the parameter or, alternatively, the term *value* to reference the entire value.

Brown et al (1996), Brown (1997) and Agarwal (1999) consider explicit references to shapes and shape rules. For example, Brown's (1997) description rules for volume calculation require the conjunctive shape rule to provide values for the diameter and length properties of the section when adding a new section to the shaft. In this case, the reference specifies the drawing or shape type, and property. Additionally, the specific shape may be distinguished by providing a filter, e.g., the value of a label the shape may have assigned. Furthermore, if the shape is a product of the conjunctive shape rule, or its property value may be affected by the shape rule, the reference may be specified to refer to the rhs of the shape rule.

Agarwal's (1999) cost equations also make explicit reference to characteristics of the shape under rule application, such as its dimensions, however, these are not evaluated during rule application. Nevertheless, in order to provide feedback on how design changes affect the cost during the generation process, the cost equations must be able to be evaluated on the corresponding shape at any time. To achieve such delayed evaluation, the cost equations may be constructed as strings, and an *eval* method may interpret the string as a numeric expression with references to the current shape.

Sets

Brown et al (1996), Duarte (2005b), Al-kazzaz (2011) and Stouffs and Tunçer (2015) consider sets of instances of descriptions. The use of a set, rather than a tuple, allows for the identification, alteration and removal of instances of descriptions without having to be concerned with the size of the tuple or the ordering of the elements in the tuple. For example, Eloy (2012a) uses a variable-length tuple but identifies only the individual elements of concern in the lhs of the description rules. A set representation is undoubtedly more appropriate here. Duarte (2001) considers tables as fixed descriptions, containing dimensional and cost information. Here too, each table can be represented as a set, of tuples, where each tuple specifies the various table indices and the corresponding cell value.

Considering instances of descriptions as members of a set is straightforward. Similar to a shape being represented as a collection of maximal elements, a description *form* is represented as a collection of individual descriptions or instances. Where a parametric shape rule in the form of $a \rightarrow b$ applies to a shape s, under a parametric assignment g and a transformation f, if $f(g(a)) \leq s$, yielding the shape $s' = s - f(g(a)) + f(g(b))$, a parametric description rule $A \rightarrow B$, with A and B sets of parametric description entities, applies to a set S if $g(A) \subseteq S$ and yields the set $S' = (S / g(A)) \cup g(B)$. Instead, when descriptions are not considered as sets, this degenerates to a description rule $a \rightarrow b$ applying to a description d, under a parametric assignment g, only if $g(a) = d$, yielding the description $d' = g(b)$. That is, description rules commonly do not apply under a part relationship but require the entire—single—description to be matched, parametrically, by the left-hand-side of the rule and specify the complete replacement of the description

according to the parametric assignment of the right-hand-side of the rule. Fortunately, both behaviours rely upon the same mechanism.

Discussion

Some of the differences between the formal notation reviewed above and each of the eighteen description schemes this notation is based upon are mainly cosmetic in nature, e.g., the need to quote strings and enumerated terms, to use an explicit concatenation operator, and to specify a description designator in a reference. Even the way conditionals are integrated into the description is broadly of a cosmetic nature. We could even consider a more informal presentation or visualization of descriptions and description rules, though not for the—unambiguous—specification thereof.

Some other differences are due to convention, for example, considering the use of a kleene star and kleene plus signifier to collect a sub-tuple of elements instead of Brown et al's (1996) notation borrowed from logic programming.

However, there is also at least one omission in the formal notation. Li (2001), Duarte (2001; 2005), Beirão (2012) and Eloy (2012a) allow rules to request or necessitate user input. Specifically, Li (2001) and Beirão (2012) identify a series of variables for input by the user, the input for which can be provided beforehand or, if missing, upon rule application. In the case of Duarte (2001; 2005) and Eloy (2012a), however, this input is often required upon rule application, because the same rule might apply more than once, each time with a possibly different input value, or values. The formal notation does not yet include the ability to signify a parameter for user input. The implementation could be envisioned to provide a default interface for requesting and storing user input, while

allowing for the application developer to overwrite the default interface with an application-specific interface.

Not all the differences are readily specifiable, for example the adopted notation for conditionals may not support all variations in conditionals found in literature. Without further examination, it is hard to predict how well (or badly) this notation for conditionals performs in light of the various description schemes. For this reason, and to identify other dissimilarities that may have remained unnoticed, we need to revisit some or all of the description schemes in literature and attempt to recast and redevelop these descriptions and rules according to the formal notation and its implementation. For example, (reference withheld) has done so for Stiny's (1981) description scheme. One obvious alteration in this example is the explicit use of functions to retrieve the last coordinate pair of a tuple, determine the number of distinct coordinate pairs in a tuple, etc. Less obvious, is the need to explicate the initial coordinate pair as a tuple of this coordinate pair and the empty entity *e*, because, otherwise, extracting the last element of the tuple would yield a single coordinate rather than the coordinate pair. Stiny (1981), instead, relegates the extraction of the last coordinate pair to the explanation provided with the rule.

Revisiting the other description schemes is the subject of another paper (forthcoming). This follow-up paper will revisit selected applications of description grammars and demonstrate the applicability of this general notation to these case studies. It is both meant as an illustration and as a confirmation of the strengths and limitations reviewed here.

We have developed an implementation of a description grammar interpreter, adhering to the formal notation presented above, in the context of a *sortal* grammar interpreter.

Sortal grammars (reference withheld) are a formalism (or rather, a class of formalisms) for design grammars, extending on shape grammars. *Sortal* grammars utilize *sortal* structures (reference withheld) as representational structures, benefiting from the fact that every component *sort* specifies a partial order relationship on its individuals and forms, defining both the matching operation and the arithmetic operations for rule application. The *sortal* grammars framework supports parallel descriptions and the association of descriptions to shapes, the latter in support of Beirão (2012).

Conclusion

We reviewed a formal notation for descriptions and description rules and explicated its strengths and limitations with respect to eighteen description schemes found in literature. Revisiting the description schemes with respect to this notation is the subject of a follow-up paper (forthcoming).

References

Agarwal M, 1999, *Supporting Automated Design Generation: Function Based Shape Grammars and Insightful Optimization* PhD thesis, Department of Mechanical Engineering, Carnegie Mellon University, Pittsburgh, PA

Agarwal M, Cagan J, Constantine K G, 1999, "Influencing generative design through continuous evaluation: associating costs with the coffeemaker shape grammar" *Artificial Intelligence for Engineering Design, Analysis and Manufacturing* **13**(4) 253–275

Ahmad S, 2009, *A framework for strategic style change using goal driven grammar transformations* PhD thesis, Department of Architecture, University of Strathclyde, Glasgow, UK

Ahmad S, Chase S, 2006, "Grammar representations to facilitate style innovation: with an example from mobile phone design", in *Communicating Space(s)* Eds V Bourdakis, D Charitos (eCAADe, Brussels) pp 320–323

Al-kazzaz D A A, 2011, *Shape grammars for hybrid component-based design* PhD thesis, Department of Architecture, University of Strathclyde, Glasgow, UK

Al-kazzaz D, Bridges A, Chase S, 2010, "Shape grammars for innovative hybrid typological design", in *Future Cities* Eds G Schmitt, L Hovestadt, L Van Gool, F Bosché, R Burkhard, S Coleman, J Halatsch, M Hansmeyer, S Konsorski-Lang, A Kunze, M Sehmi-Luck (eCAADe, Brussels) pp 187–195

Beirão J N, 2012, *CItyMaker: Designing Grammars for Urban Design* PhD thesis, Faculty of Architecture, Delft University of Technology

Brown K, 1997, "Grammatical design" *IEEE Expert* **12**(2) 27–33

Brown K N, Cagan J, 1997, "Optimized process planning by generative simulated annealing" *Artificial Intelligence for Engineering Design, Analysis and Manufacturing* **11** 219–235

Brown K N, McMahon C A, Sims Williams J H, 1996, "Describing process plans as the formal semantics of a language of shape" *Artificial Intelligence in Engineering* **10**(2) 153–169

Çağdaş G, 1996, “A shape grammar: the language of traditional Turkish houses” *Environment and Planning B: Planning and Design* **23**(4) 443–464

Correia R C, 2013, “DESIGNA - a shape grammar interpreter”, MSc thesis, Instituto Superior Técnico, Universidade de Lisboa, Lisbon, Portugal

Coutinho P F, 2014, *Gramática da Forma da Sistematização da Coluna de Alberti* volume 1 and 2, PhD thesis, Department of Architecture, Universidade de Coimbra, Coimbra, Portugal

Downing F, Flemming U, 1981, “The bungalows of Buffalo” *Environment and Planning B: Planning and Design* **8**(3) 269–293

Duarte J P, 2001, *Customizing Mass Housing: A Discursive Grammar for Siza's Malagueira Houses* PhD thesis, Department of Architecture, MIT

Duarte J P, 2005a, “A discursive grammar for customizing mass housing: the case of Siza’s houses at Malagueira” *Automation in Construction* **14**(2) 265–275

Duarte J P, 2005b, “Towards the mass customization of housing: the grammar of Siza's houses at Malagueira” *Environment and Planning B: Planning and Design* **32**(3) 347–380

Duarte J P, Correia R, 2006, “Implementing a description grammar: generating housing briefs online” *Construction Innovation: Information, Process, Management* **6**(4) 203–216

Duarte J P, Beirão J N, Montenegro N, Gil J, 2012, “City induction: a model for formulating, generating, and evaluating urban designs”, in *Digital Urban Modeling and*

Simulation Eds S Müller Arisona, G Aschwanden, J Halatsch, P Wonka (Springer, Berlin) pp 73–98

Eloy S, 2012a, *A transformation grammar-based methodology for housing rehabilitation: meeting contemporary functional and ICT requirements* PhD thesis, Instituto Superior Técnico, TU Lisbon, Lisbon, Portugal.

Eloy S, 2012b, *A transformation grammar-based methodology for housing rehabilitation: meeting contemporary functional and ICT requirements: dwellings characterization and transformation rules* PhD thesis appendix, Instituto Superior Técnico, TU Lisbon, Lisbon, Portugal.

Eloy S, Duarte J P, 2014, "A transformation grammar-based methodology for housing rehabilitation", in *Design Computing and Cognition '12* Ed J S Gero (Springer, Dordrecht) pp 301-320

Knight T, 2003, "Computing with emergence" *Environment and Planning B: Planning and Design* **30**(1) 125–155

Li A I, 2001, *A shape grammar for teaching the architectural style of the Yingzao fashi* PhD thesis, Department of Architecture, MIT

Li A I, 2004, "Styles, grammars, authors, and users", in *Design Computing and Cognition '04* Eds J S Gero (Kluwer Academic, Dordrecht) pp 197-215

Liang S, 1983, *Yingzaofashi zhushi*, (Zhongguo jianzhu gongye, Beijing)

Liew H, 2004, *SGML: A Meta-Language for Shape Grammar* PhD thesis, Department of Architecture, MIT

March L, Stiny G, 1985, “Spatial systems in architecture and design: some history and logic” *Environment and Planning B: Planning and Design* **12**(1) 31–53.

Muslimin R, 2013, “Decoding Passura’ – representing the indigenous visual messages underlying traditional icons with descriptive grammar”, in *Open Systems* Eds R Stouffs, P Janssen, S Roudavski, B Tunçer (CAADRIA, Hong Kong) pp 781–790

Stiny G, 1980, “Introduction to shape and shape grammars” *Environment and Planning B: Planning and Design* **7**(3) 343–351

Stiny G, 1981, “A note on the description of designs” *Environment and Planning B: Planning and Design* **8**(3) 257–267

Stiny G, 1987, “Composition counts: A + E = AE” *Environment and Planning B: Planning and Design* **14**(2) 167–182

Stiny G, 1991, “The algebras of design” *Research in Engineering Design* **2**(3) 171–181

Stiny G, 2006, *Shape: Talking about Seeing and Doing* (MIT, Cambridge, MA)

Stiny G, Mitchell W J, 1978, “The Palladian grammar” *Environment and Planning B: Planning and Design* **5**(1) 5–18

Stouffs R, Tunçer B, 2015, “Typological descriptions as generative guides for historical architecture” *Nexus Network Journal* **17**(3)

Zamenopoulos T, 2012, “A complexity theory of design intentionality” *Artificial Intelligence for Engineering Design, Analysis and Manufacturing* **26**(1) 63–83

forthcoming, “Description grammars: precedents revisited” submitted to *Environment and Planning B: Planning and Design*