Implementing a Description Grammar Interpreter

A Notation for Descriptions and Description Rules

Rudi Stouffs[1]
[1]*National University of Singapore*
[1]*stouffs@nus.edu.sg*

Description grammars represent a formalism for generating verbal descriptions of designs, used in conjunction with shape grammars. A description grammar constitutes a set of description rules that define a language of descriptions. A description grammar interpreter implements the mechanisms to interpret descriptions and description rules, apply description rules to descriptions to derive new descriptions, and generate the members of the corresponding language of descriptions. In this paper, we argue the need for the development of a (general) description grammar interpreter, offer a formal notation for descriptions and description rules, describe the representation and implementation of descriptions and description rules, and provide an example implemented using the description grammar interpreter.

Keywords: *Shape grammars, Description grammars, Grammar interpreter*

INTRODUCTION

Description grammars represent a formalism for generating verbal descriptions of designs, used in conjunction with shape grammars. They were conceived by Stiny (1981) in the form of a description function that augments a shape grammar. Subsequent authors have either used the term description function or description grammar. In the latter notion, the description grammar can be considered as a parallel grammar to the shape grammar.

Stiny (1981) illustrates the application of a description function with designs made up of blocks from Froebel's building gifts. Stouffs (2015) presents an extensive overview of applications of description grammars in literature. Few of these include an implementation of their description grammar. Duarte and Correia (2006) describe the implementation of a description grammar that codifies the Portuguese housing design guidelines, where the description rules are specifically encoded to handle custom description structures. Duarte et al. (2012, p. 84) identify the lack of a (general) 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.

The lack of a general description grammar interpreter can impede the development of sound description grammars. Eloy (2012a) applies a discursive grammar, incorporating a shape grammar and a description grammar (and a set of heuristics) to housing rehabilitation. She reflects on a possible implementation and 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 then 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). Correia (2013) describes the implementation of a shape grammar interpreter for Duarte's (2001) Malaguiera grammar, omitting the description grammar part. However, he does discuss the description aspect of the Malagueira grammar, in particular, the difficulty of implementing the rules of the Malagueira grammar because of the "many ambiguities and even some errors" these rules present, which he illustrates with a few examples (Correia 2013, p. 60-61). While some inaccuracies can always be expected, we argue that the ability to implement a description grammar can be an important aid in improving rigor and reducing ambiguity. This is the case even if the description grammar interpreter cannot fully support all features of the proposed description grammar and, therefore, only allows for the implementation of a somewhat simplified or approximated grammar.

In this paper, we report on the implementation of a (general) description grammar interpreter that builds upon the generalized specification for descriptions and description rules presented by Stouffs (2014). First, we offer a formal notation for descriptions and description rules. Next, we describe the representational structure used to represent descriptions and description rules within the grammar interpreter. Subsequently, we revisit Stiny's (1981) illustration of a description function with designs made up of blocks from Froebel's building gifts in an application of the description grammar interpreter.

FORMAL TEXTUAL DESCRIPTIONS

We consider descriptions as textual in nature. In his overview, Stouffs (2015) omits any "descriptions in the form of (spatial) topological, ontological or graph structures that require specific, non-textual representational structures." Nevertheless, textual descriptions are not limited to plain text; they can include numbers as well as strings, lists, and sets (of descriptions). When part of a description rule, either as the left-hand-side or right-hand-side, a description may also include operators and functions, parameters, conditionals, and references. Within the left-hand-side of a description rule, operators, functions and references generally only occur within conditionals, though the concatenation operator on strings is a prominent exception as it can be used not only to concatenate strings but also to identify substrings in the matching process.

Examples of descriptions and description rules in literature mostly adopt a notation that is favorable to a human reader, not necessarily machine-readable. For example, Li (2001) merely distinguishes parameters, from among (unquoted) literal text and numeric expressions, as single letters (possibly with a superscript number) in italics, e.g., a_1 (though the Chinese versions - using the Latin alphabet - of descriptions and description rules are entirely presented in italics). As another example, Duarte (2001) distinguishes parameters from enumerated terms as uppercase letters (possibly with a number), e.g., F1. Instead, we adopt a more explicit, machine-readable notation, distinguishing literals, including numbers and (quoted) strings, and expressions, including numeric and string expressions, with explicit operators. We conceive that this explicit notation could be parsed and presented in a more human readable form, though we do not explore it. The opposite operation of interpreting an informal presentation is in most instances grammar or author specific.

Below, we describe the formal textual notation for description entities, specifically, literals, functions, numeric expressions, string expressions, tuples, tuple expressions, parameters and references. For each entity type, we provide a formal specification in Extended Backus-Naur-Form (EBNF), using single quotation marks to delimit literal text, square brackets ('[...]') to delimit an optional construct, braces ('{...}') to indicate zero or more repetitions of the enclosed con-

struct, parentheses ('(...)') to indicate a simple grouping of constructs, and a vertical bar ('|') to indicate a choice of one from many.

Description entities may be defined differently when used within a description, within the left-hand-side (lhs) of a description rule, or within the right-hand-side (rhs) of a description rule. Nevertheless, there are also a lot of similarities. For this reason, we draw from the same pool of nonterminals in defining the production rules for all three cases, and specify alternative production rules for the same nonterminal only when needed. We identify the three cases with the terms *description*, *lhs* and *rhs*, and add the respective term, enclosed within angle brackets ('<...>') as a prefix to the respective production rule.

Literals

Numbers can be integral or decimal, positive or negative. Literal strings are (double) quoted; the escape character ('\') can be used to include a double quote within a string. A few literals are predefined, including *e*, *nil*, *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). Though we do not support logical expressions, in conformity with Duarte (2001), we assume the literals *true* and *false* to represent 1 and 0, respectively. Various authors consider other enumerated (unquoted) terms, for example, denoting functions, spaces, qualifications, rule labels, and ontological terms. However, except for *true* and *false*, all these enumerations are grammar-specific. Therefore, these are represented as strings. Unquoted terms, other than predefined literals, are assumed to define a function, a parameter, or part of a reference.

---

literal = *keyword-literal* | *number* | *string* .
keyword-literal = 'nil' | 'e' | '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-char-except-quote* | '\' ' " ' .

Functions

Functions offer specific, often complex, functionality that cannot be expressed simply through an operator. For example, Stiny (1981) considers retrieving the last coordinate pair of a list, determining the number of distinct coordinate pairs in a list, retrieving loops of coordinate pairs in a list, etc. Though not explicated as functions by Stiny, these operations are here considered as functions.

Functions can occur anywhere in the rhs of a description rule but, within the lhs of a description rule, only functions returning numbers can occur and onlyin a conditional. For each expression-entity type (numeric, string or tuple), a number of functions are predefined. Numeric functions include square root(*sqrt*), *sin*e, *co*sine and *tan*gent. These functions take a numeric expression as input and return a literal number upon evaluation. String functions include the *length* of a string, and a *left* and *right* substring. The length function takes a string or string expression (if used in the rhs of a description rule) and returns a literal number; the substring functions take a string expression and a numeric expression, specifying the number ofcharacters for the substring, and return a literal string. Tuple functions include the *length* of a tuple, the *first* and *last* entity of a tuple, and a number of functions derived from Stiny (1981): 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.

---

function = *function-returns-number* | *function-returns-string* | *function-returns-tuple* .
function-returns-number = *numeric-function* | *length-function* | *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* | *tuple-function-untyped* .
string-function-returns-string = ('left' | 'right') '(' *string-argument* ',' *numeric-expression* ')' .
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* | *tuple-function-untyped* .
tuple-function-returns-tuple = ('unique' | 'segments' | 'pairwise' | 'loops') '(' *tuple-argument* ')' | 'adjacencies' '(' *tuple-argument* ',' *tuple-argument* ')' .

Numeric expressions

Similar to functions, numeric expressions can only occur within a conditional in the lhs of a description rule, and anywhere in the rhs of a description rule. The definition of a numeric expression is entirely the same in both cases. Parameters and numbers (as well as references, functions returning numbers, and predefined literals) can be combined in numeric expressions. All numeric expressions require explicit operators: plus ('+'), minus ('-'), times ('*'), divided-by ('/'), modulo ('%') and to-the-power-of ('^'). The usual operator precedence rules apply and parentheses can be used to override these rules.

numeric-expression = *term* { *addition-op term* } .
term = *factor* { *multiplication-op factor* } .
factor = *base* { *exponentiation-op exponent* } .
exponent = *base* .
base = *keyword-literal* | *number* | '(' *numeric-expression* ')' | *function-returns-number* | *parameter* | *reference* .
exponentiation-op = '^' .
multiplication-op = '*' | '/' | '%' .
addition-op = '+' | '-' .

String expressions

While string expressions can occur both in the lhs and rhs of a description rule, within an lhs, only literals and parameters are allowed, though the parameters may be each augmented with a conditional. Within an rhs, on the other hand, a string expression is any concatenation of description entities other than tuples (or tuple expressions). Numeric expressions within string expressions must be enclosed within parentheses. As stated before, Li (2001) adopts a more informal notation, omitting parentheses around numeric expressions, double quotes around literal strings and the concatenation operator ('.'). However, we require a machine-readable format that can be readily disambiguated. Such disambiguation may still be needed, for example, when a string expression contains a floating-point number or, alternatively, a concatenation of two literal numbers; the former interpretation will take precedence over the latter.

string-expression = *string-expression-entity* { '.' *string-expression-entity* } .
<lhs>*string-expression-entity* = *literal* | *parameter* ['?' *conditional*] .
<rhs>*string-expression-entity* = *base* | *string* | *function-returns-string* .

Tuples

A tuple is any list or sequence of description entities. Tuples can be nested. Most authors use tuples in descriptions; they variably use parentheses, angle brackets and square brackets as enclosing brackets to identify tuples, and commas or semicolons to separate entities within a tuple. Sometimes, enclosing brackets are omitted altogether, if at the top level of a tuple structure, and, in a few cases, separation marks are omitted as well, leaving only spaces to separate the entities. We accommodate all these variations, considering disambiguation rules where and when necessary. These disambiguation rules intend to mimic as much as possible how we, humans, might interpret such situations, though such mimicry might

imply some subjectivity.

As an example, a minus sign separating two numerical entities, the first one of which might be represented as a parameter, is interpreted as such, specifying a subtraction, even when it might be possible to consider it instead as a unary negation within a tuple of (at least two) numbers, with separation marks omitted. If the latter interpretation is required, parentheses can be used to enclose the unary negation. Note that these parentheses themselves could be misinterpreted as defining a tuple within the larger tuple. However, we are in agreement with all authors when we recognize a list of length one (or zero) only if it uses angle or square brackets.

---

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* } .

Tuple expressions

Common operators on tuples are append and prepend. Duarte (2001) additionally proposes the addition ('+') of tuples that have the same structure, that is, have the same (fixed) length and corresponding entity types. 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.

Most authors adopt an implicit append or prepend operator, merely identified by the absence of a separation mark. We concur and, in order to distinguish the append or prepend operator from simply omitting the separation marks within a tuple, we require a structural similarity between, on the one hand, the entity to be appended or prepended and, on the other hand, the entities within the larger tuple. In a similar way, we define a join operator on tuples.

---

top-level-tuple-expression = *tuple-addition* | *tuple-expression* .
tuple-addition=[*parameter*] '+' *basic-tuple-argument*.
<lhs>*tuple-expression*=*tuple-append* | *tuple-prepend* .
<rhs>*tuple-expression* = { *tuple-entity* } *parameter* { *tuple-entity* } [*tuple-expression*] .
tuple-append = { *tuple-entity* } *parameter* ('*' | '+') *tuple-entity* { *tuple-entity* } [*tuple-expression*] .
tuple-prepend = [*tuple-expression*] { *tuple-entity* } *tuple-entity parameter* ('*' | '+') { *tuple-entity* } .

Parameters

Parameters are variable terms. When used in the lhs of a description rule, a parameter is matched to a literal in a description during rule application. If the parameter is included in a string expression (concatenation), this literal can be any part of a literal string. We borrow from regular expressions the ability to collect any number of entities from a tuple in a parameter by adding a postfix kleene star ('*') or kleene plus ('+') to the parameter specification, denoting a tuple of zero, one or more, respectively, one or more entities.

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.

---

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 = '_' .
identifier = (*letter* | *underscore*) { (*letter* | *underscore* | *digit*) } .
parameter = *identifier* .

Within the lhs of a description rule, a parameter may be assigned a conditional that constrains the possible values of the parameter. This conditional may be enumerative or equational. If enumerative, the conditional explicates a finite set of possible values (all of the same type, either numbers or strings) for the parameter, for instance, an enumeration of terms. If equational, the conditional specifies a numeric equality or inequality on the parameter. The equation or inequality is specified as a conditional operator (one of '=', '<>', '<', '<=', '>', '>=') and operand, either a literal number, parameter or reference, or a numerical expression enclosed within parentheses.

---

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 .

References

References are similar to parameters; they are also variable terms. However, whereas parameters are local within a description rule, references refer to values from other, parallel grammars. For example, Li (2001) considers nine parallel descriptions, specifying measures and descriptions of width, depth, height, with corresponding rule sets. A compound rule combines rules operating on different descriptions and, within a compound rule, one rule may reference the current value from another rule. 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. Similarly, Duarte (2001) defines one description specifying the number of bedrooms, another the number of floors, and a compound rule specifying the latter with respect to the former. Differently, the number of bedrooms is specified as part of a tuple and is retrieved from this tuple via a parameter within a rule over this description.

Brown (1997) additionally considers references to shape descriptions; his description rules for volume calculation require, among others, values for the diameter and length of a newly added section. These values are accessible as properties of the shape, i.e., the diameter and length of a cylinder, and can be retrieved from the shape by specifying this property together with the element type this property belongs to, and by filtering the shape from among other shapes of the same type by additional information, such as its label and whether it occurs in the lhs or results from the rhs of the rule.

We adopt an object-oriented notation (using a dot (".) to separate object and method) for references, where the 'object' refers to the parallel description grammar and the 'method' is either the term *value*, referencing the entire value of the description matched to a description rule, the name of the parameter of a description rule, or the name of a property of a shape element.

---

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-op* (*number* | *vector* | *string*) .
filter-op = '=' | '<>' | '<=' | '>=' .
vector = [*rational*] '(' *rational* ',' *rational* ',' *rational* ')' .
rational = ['-'] *digit-sequence* ['/' *digit-sequence*] .

Descriptions

A description is either a literal (a number, a string or any of the predefined literals) or a tuple, or a sequence of literals and/or tuples separated and enclosed by a special character, e.g., '#' (as suggested by Stiny (1981)). In the latter case, the description can be considered as a compound description, incorporating different parallel descriptions.

---

description=description-entity | *description-sequence* .
description-entity = literal | *top-level-tuple* .
description-sequence = '#' *description-entity* '#' { *description-entity* '#' } .

Description rules

A description rule has an lhs and an rhs. Any description can form the lhs of a description rule. Additionally, a parameter or a string expression can be a substitute for a literal or serve as tuple entities within the lhs of a description rule. A parameter may additionally be specified a conditional; a string expression can itself only be composed of literals and parameters (with or without conditional).

Any description entity can form part of the rhs of a description rule, except that parameters cannot have a conditional specified. The rhs of a description rule can exceptionally take the form of an addition of two tuples with the first tuple omitted, as suggested by Duarte (2001). In this case, the first operand of the addition operation refers to the tuple resulting from the matching of the lhs.

---

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* | *top-level-tuple-expression* .
description-rule-sequence = '#' *description-rule-entity* '#' { *description-rule-entity* '#' } .

REPRESENTATIONAL STRUCTURE

Though the descriptions are textual in nature, we adopt a tree structure for the representation of description entities (Figure 1), in order to support the matching of description entities and, thus, the matching of (the lhs of) a description rule onto a description. Tree leaves consist of literals and parameters (in the case of a lhs), where literals represent numeric and string entities that serve as constraints in the matching process, and parameters specify variables that can be matched to leaf nodes and tree (sub)structures. Additionally, conditions can be imposed on parameters to further constrain matching. Non-leaf nodes represent tuples and string concatenations. In the case of a tuple, each entity may correspond to a child node, or a single 'variable' leaf node may imply a collection of zero, one or more entities from the tuple. Similarly, in the case of a string concatenation, the non-leaf node should match a single leaf node, expressing the actual string to be matched.

We have developed an implementation of a description grammar interpreter, adhering to the formal notation presented above, in the context of SortalGI [1], a *sortal* grammar interpreter developed for the Processing programming environment. *Sortal* grammars are a formalism (or rather, a class of formalisms) for design grammars, extending on shape grammars. *Sortal* grammars utilize *sortal* structures (Stouffs 2008) 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.

A description is represented either as a tree structure, corresponding a description entity, or as a sequence of tree structures, corresponding a compound description. Descriptions can be collected into forms. This provides support for sets of description entities. A (description) rule is represented as a pair of forms, one for the lhs and one for the rhs. From a representational point of view, no distinction is made between descriptions and either the lhs or rhs of description rules, unlike the formal notation

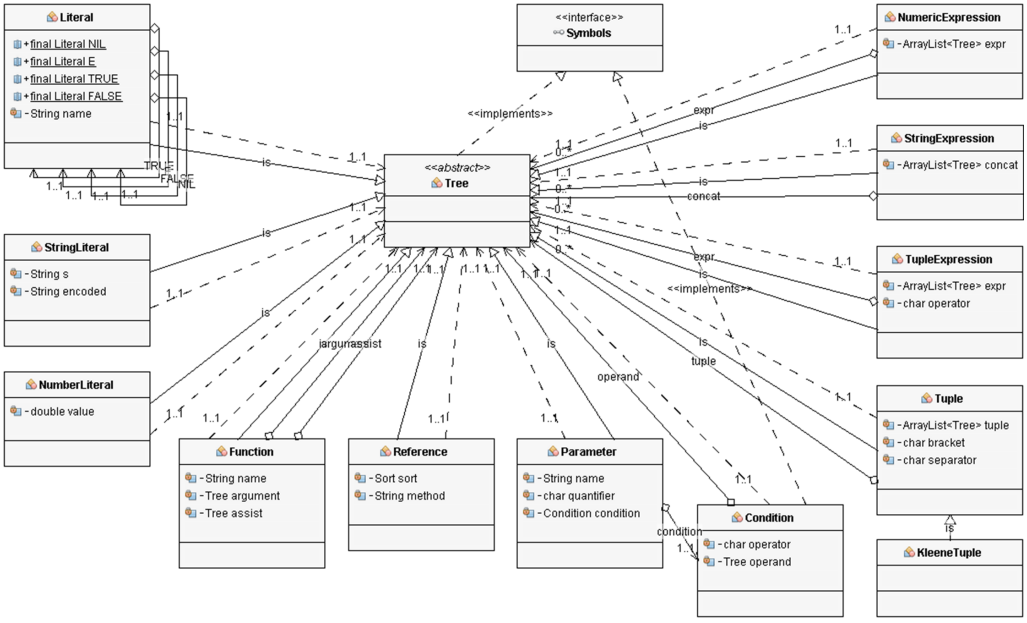

Figure 1
UML diagram of the (Java) classes defining a description tree (corresponding a description entity).

above. Any inaccuracies with respect to the context dependency of the formal notation are caught as run-time errors upon rule application.

EXAMPLE

We revisit Stiny's (1981) example illustrating the application of a description function with designs made up of blocks from Froebel's building gifts. We refer to Stiny (1981) for the specification of the description rules (which he denotes functions) and the example (Stiny 1981, p. 262). We present the example here in the Sortal Description Language (SDL), an interpretive language for describing *sortal* representational structures, data constructs and rules.

Stiny defines a description having the form

$\#a_1\#a_2\#a_3\#a_4\#a_5\#a_6\#a_7\#a_8\#$

where a_1 is either a sequence of coordinate pairs indicating the positions of the pillars visited, or a single coordinate pair indicating the position of the pillar currently visited; a_2 is an integer specifying the number of pillars and a_3 an integer specifying the number of wall elements; a_4 is a list of rooms (enclosed spaces) not containing other rooms, with each room specified as a tuple of coordinate pairs indicating the positions of the pillars at the corners of the room; a_5 is a list of stiles, with each stile specified by the two coordinate pairs indicating the pillars it lies between; similarly, a_6 is a list of doors, and a_7 a list of windows; finally, a_8 is an adjacency matrix based on the rooms, stiles and doors.

Firstly, we define a *sortal* structure termed 'description' to represent the descriptions:

sort description : [Description];

Secondly, we define each of the seventeen functions from Stiny's example as description rules. Each rule is assigned a name ('g1' through 'g17'), a brief explanation, and an lhs and rhs description. The descriptions are specified confirm the formal notation defined above. They are enclosed within backquotes (`...`) to distinguish the descriptions from the remaining SDL code. Rules g_1 through g_4 construct a sequence of coordinate pairs indicating the positions

of the pillars visited. Rather than extracting the *x* and *y* coordinates, a direction vector in the form of a coordinate pair is added to the last coordinate pair and the result is appended to the existing tuple. Rule g_5 counts the number of pillars and the number of wall elements, creates a list of rooms, and reduces the list of coordinate pairs to the last position. It makes use of the tuple functions *last*, *length*, *unique*, *segments* and *loops*. Rules g_6 through g_9 update the coordinate pair indicating the current position. Rules g_{10} and g_{11} create stiles, rules g_{12} and g_{13} create doors, and rules g_{14} and g_{15} create windows. Rule g_{16} creates the adjacency matrix from the rooms tuple (a_4) and the merged stiles (a_5) and doors (a_6) tuples. Rule g_{17} does not alter the description.

```
rule g1 "adds a coordinate pair to the North"
<< description: {`#a1#a2#a3#a4#a5#a6#a7#a8#`}
>> description: {`#a1 last(a1) + (0, 1)#a2#a3#a4#a5#a6#a7#a8#`};
rule g2 "adds a coordinate pair to the South"
<< description: {`#a1#a2#a3#a4#a5#a6#a7#a8#`}
>> description: {`#a1 last(a1) + (0, -1)#a2#a3#a4#a5#a6#a7#a8#`};
rule g3 "adds a coordinate pair to the East"
<< description: {`#a1#a2#a3#a4#a5#a6#a7#a8#`}
>> description: {`#a1 last(a1) + (1, 0)#a2#a3#a4#a5#a6#a7#a8#`};
rule g4 "adds a coordinate pair to the West"
<< description: {`#a1#a2#a3#a4#a5#a6#a7#a8#`}
>> description: {`#a1 last(a1) + (-1, 0)#a2#a3#a4#a5#a6#a7#a8#`};
rule g5 "counts the number of pillars and the number of wall elements, and builds the list of rooms"
<< description: {`#a1#a2#a3#a4#a5#a6#a7#a8#`}
>> description: {`#last(a1)#length(unique(a1))#length(unique(segments(a1)))#loops(a1)#a5#a6#a7#a8#`};
rule g6 "moves the current position to the North"
<< description: {`#a1#a2#a3#a4#a5#a6#a7#a8#`}
>> description: {`#a1 + (0, 1)#a2#a3#a4#a5#a6#a7#a8#`};
rule g7 "moves the current position to the South"
<< description: {`#a1#a2#a3#a4#a5#a6#a7#a8#`}
>> description: {`#a1 + (0, -1)#a2#a3#a4#a5#a6#a7#a8#`};
rule g8 "moves the current position to the East"
<< description: {`#a1#a2#a3#a4#a5#a6#a7#a8#`}
>> description: {`#a1 + (1, 0)#a2#a3#a4#a5#a6#a7#a8#`};
rule g9 "moves the current position to the West"
<< description: {`#a1#a2#a3#a4#a5#a6#a7#a8#`}
>> description: {`#a1 + (-1, 0)#a2#a3#a4#a5#a6#a7#a8#`};
rule g10 "adds a stile facing East"
<< description: {`#a1#a2#a3#a4#a5#a6#a7#a8#`}
>> description: {`#a1#a2#a3 - 1#a4#a5 <a1, a1 + (0, 1)>#a6#a7#a8#`};
rule g11 "adds a stile facing South"
<< description: {`#a1#a2#a3#a4#a5#a6#a7#a8#`}
>> description: {`#a1#a2#a3 - 1#a4#a5 <a1, a1 + (1, 0)>#a6#a7#a8#`};
rule g12 "adds a door facing East"
<< description: {`#a1#a2#a3#a4#a5#a6#a7#a8#`}
>> description: {`#a1#a2#a3 - 1#a4#a5#a6 <a1, a1 + (0, 1)>#a7#a8#`};
rule g13 "adds a door facing South"
<< description: {`#a1#a2#a3#a4#a5#a6#a7#a8#`}
>> description: {`#a1#a2#a3 - 1#a4#a5#a6 <a1, a1 + (1, 0)>#a7#a8#`};
rule g14 "adds a window facing East"
<< description: {`#a1#a2#a3#a4#a5#a6#a7#a8#`}
>> description: {`#a1#a2#a3 - 1#a4#a5#a6#a7 <a1, a1 + (0, 1)>#a8#`};
rule g15 "adds a window facing South"
<< description: {`#a1#a2#a3#a4#a5#a6#a7#a8#`}
>> description: {`#a1#a2#a3 - 1#a4#a5#a6#a7 <a1, a1 + (1, 0)>#a8#`};
rule g16 "builds the adjacency matrix"
<< description: {`#a1#a2#a3#a4#a5#a6#a7#a8#`}
>> description: {`#e#a2#a3#a4#a5#a6#a7#adjacencies(a4, a5 a6)#`};
rule g17 "identity function"
<< description: {`#a1#a2#a3#a4#a5#a6#a7#a8#`}
>> description: {`#a1#a2#a3#a4#a5#a6#a7#a8#`};
```

Thirdly, we construct a derivation as a series of rule applications. Each intermediate result is captured in an SDL variable. Note that the initial description differs from Stiny's initial description in that the first part of the description (corresponding to a_1) is explicated as a tuple consisting of a coordinate pair and an 'empty' entity, resulting in a tuple of a single coordinate pair. Otherwise, extracting the last element of the tuple would yield a single coordinate rather than

the coordinate pair.

```
form $initial = description: {`#(0, 0) e#e#e#e#e#e
#e#e#`};
form $pillars = description: $initial|g3|g1|g4|g2|
g3|g3|g1|g4;
form $rooms = description: $pillars|g5;
form $final = description: $rooms|g9|g13|g7|g14|
g11|g8|g14|g8|g10|g16;
```

The results are as follows, output in SDL format:

```
sort description : [Description];
form $initial = description: {`#(0, 0) e#e#e#e#e#e
#e#e#`};
form $pillars = description: {`#(0, 0) (1, 0) (1, 1) (0, 1)
(0, 0) (1, 0) (2, 0) (2, 1) (1, 1)#e#e#e#e#e#e#e#`};
form $rooms = description: {`#(1, 1)#6#7#<(0, 0), (1,
0), (1, 1), (0, 1)> <(1, 0), (2, 0), (2, 1), (1, 1)>#e#e#e#
e#`};
form $final = description: {`#e#6#2#<(0, 0), (1, 0), (1,
1), (0, 1)> <(1, 0), (2, 0), (2, 1), (1, 1)>#<(0, 0), (1, 0)>
<(2, 0), (2, 1)>#<(0, 1), (1, 1)>#<(0, 0), (0, 1)> <(1, 0),
(1, 1)>#<" ", "O", "R0", "R1"> <"O", 1, 1, 1> <"R0", 1, 1,
0> <"R1", 1, 0, 1>#`};
```

These results, other than the initial form, are in full agreement with Stiny's (1981, p. 262) example, except that Stiny offers a graphical depiction of the adjacency matrix and terms the rooms "W" and "E", instead of "R0" and "R1".

CONCLUSION

We have argued the need for the development of a (general) description grammar interpreter, have formally described descriptions and description rules, touched upon the representational structure of descriptions and description rules and its implementation, and revisited Stiny's (1981) example of a description function in an application of the description grammar interpreter. Future work will be focused on qualifying the generality of the description grammar interpreter and on addressing its limitations. Firstly, we need to explicate the limitations of the formal notation with respect to the description schemes presented in literature. The implementation of additional examples from literature may support this investigation. Secondly, we need to extend the implementation of the description grammar interpreter to address current limitations and shortcomings.

REFERENCES

Correia, RC 2013, *DESIGNA - A Shape Grammar Interpreter*, Ph.D. Thesis, Instituto Superior Técnico, TU Lisbon

Duarte, JP 2001, *Customizing Mass Housing: A Discursive Grammar for Siza's Malagueira Houses*, Ph.D. Thesis, Dept. of Architecture, MIT

Duarte, JP, Beirão, JN, Montenegro, N and Gil, J 2012, 'City Induction: A Model for Formulating, Generating, and Evaluating Urban Designs', in Müller Arisona, S, Aschwanden, G, Halatsch, J and Wonka, P (eds) 2012, *Digital Urban Modeling and Simulation*, Springer, Berlin, p. 73–98

Duarte, JP and Correia, R 2006, 'Implementing a description grammar: Generating housing briefs online', *Construction Innovation: Information, Process, Management*, 6(4), p. 203–216

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

Eloy, S 2012b, *A transformation grammar-based methodology for housing rehabilitation: meeting contemporary functional and ICT requirements, Appendix*, Ph.D. Thesis, Instituto Superior Técnico, TU Lisbon

Li, AI 2001, *A shape grammar for teaching the architectural style of the Yingzao fashi*, Ph.D. Thesis, Dept. of Architecture, MIT

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

Stouffs, R 2008, 'Constructing design representations using a sortal approach', *Advanced Engineering Informatics*, 22(1), p. 71–89

Stouffs, R 2015, 'Description grammars: an overview', in Ikeda, Y, Herr, CM, Holzer, D, Kajima, S, Kim, MJ and Schnabel, MA (eds) 2015, *Emerging Experience in Past, Present and Future of Digital Architecture*, CAADRIA, Hong Kong

Stouffs, R 2014, 'Towards a formal representation for description rules', in Thompson, M (eds) 2014, *Fusion, vol. 2*, eCAADe, Brussels, p. 347–356

[1] http://sortal.org/downloads/sortalgi.html