EUROGRAPHICS 2019 / P. Alliez and F. Pellacini
(Guest Editors)

Volume 38 (2019), Number 2

Local Editing of Procedural Models

M. Lipp[1], M. Specht[1], C. Lau[1], P. Wonka[2], and P. Müller[1]

[1]Esri R&D Center Zurich, Switzerland
[2] KAUST, Saudi Arabia

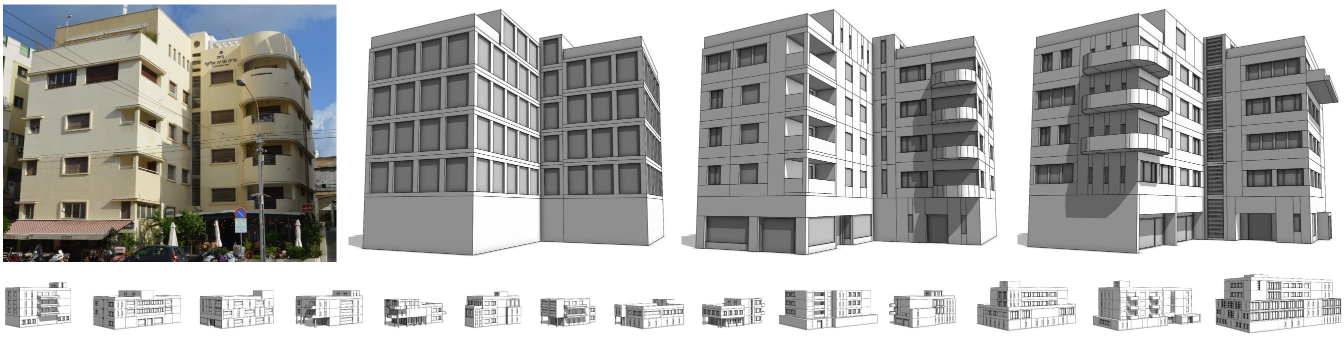

Figure 1: *We propose a system which gives the user full artistic control over rule-based procedural models. With our system, an artist created building designs showcasing architectural styles in Tel Aviv based on reference imagery (left). The artist authored one basic rule set (middle left) where local edits enabled him to quickly reconstruct an existing building (middle right) or design a new alternative (right). By using local edits, the artist was able to create a large amount of detailed building designs in a matter of hours rather than days (bottom).*

Abstract

Procedural modeling is used across many industries for rapid 3D content creation. However, professional procedural tools often lack artistic control, requiring manual edits on baked results, diminishing the advantages of a procedural modeling pipeline. Previous approaches to enable local artistic control require special annotations of the procedural system and manual exploration of potential edit locations. Therefore, we propose a novel approach to discover meaningful and non-redundant good edit locations (GELs). We introduce a bottom-up algorithm for finding GELs directly from the attributes in procedural models, without special annotations. To make attribute edits at GELs persistent, we analyze their local spatial context and construct a meta-locator to uniquely specify their structure. Meta-locators are calculated independently per attribute, making them robust against changes in the procedural system. Functions on meta-locators enable intuitive and robust multi-selections. Finally, we introduce an algorithm to transfer meta-locators to a different procedural model. We show that our approach greatly simplifies the exploration of the local edit space, and we demonstrate its usefulness in a user study and multiple real-world examples.

CCS Concepts
• ***Computing methodologies*** $\rightarrow$ ***Mesh geometry models;***

1. Introduction

Procedural modeling is a popular 3D content creation method used across many industries ranging from film and game production to architecture and planning. Professional 3D tools such as Houdini, Maya, Rhino, CityEngine or Speedtree provide frameworks to define a procedural model as well as its control user interface (UI). The latter allows the artist to interactively modify certain attributes of the models. In practice, not enough artistic control can be provided by these UIs. Therefore, an artist typically bakes the procedural model into a mesh to apply additional local edits. As a consequence, the advantages of a procedural modeling pipeline are lost.

Local editing of a procedural model is a challenge that has not yet been solved in a generic way. Some procedural frameworks try to store such edit operations in the scene graph, resulting in difficult manual modeling processes and edits which are not robust against changes in the procedural model (see Figure 2(a)). Other frameworks allow the author of the procedural system to explicitly define local edit interfaces (e.g. [LWW08] and [JPCS18] where the author needs to tag rules and is required to setup facade grids including dimensions per row and column, resulting in tedious rule authoring processes and limited application areas (see Figure 2(c)). As a consequence, authors do not use local edits and instead create procedural modeling systems with countless attributes in order to have more artistic control, resulting in large unmaintainable code bases and unintuitive UIs (see Figure 2(b)).

In this paper, we propose a novel approach for local editing in rule-based procedural modeling systems that does not require any annotations, rule changes, or pre-processing. It utilizes the fact that every author of procedural modeling systems uses attributes, similar to every programmer using variables instead of magic numbers in his code. Our hypothesis is that attribute usage in a procedural model implicitly contains the information necessary to do local edits which are robust against changes, are transferable onto other models, and allow for full artistic control. Figure 1 shows how an artist efficiently designed a set of buildings with our local editing system. Our approach makes the following main contributions:

- We present the novel concept of *Good Edit Locations* (GELs) which describes, *independently* per attribute, the optimal set of possible edit locations in the derivation tree of a procedural model. They allow the artist to intuitively modify attributes locally without knowledge of the rule set or the derivation tree hierarchy. Without the need for any pre-processing or manual annotation, the set is automatically defined by an efficient bottom-up algorithm which can be executed in real-time.
- To persist the artist's local edits, we introduce *meta-locators*, robust descriptors to identify a GEL or a set of GELs in a derivation tree *independently* for each attribute. Meta-locators use multiple functions to analyze the local context of a GEL. For example, a meta-locator could describe an local edit on the first window of the top floor on the facade facing south (but note again that no explicit definition of facade, floor, or window is required). Further, combination functions are introduced on top of meta-locators, which allow, for example, to replace a set of meta-locators with one meta-locator using wildcards. This enables a robust and intuitive user experience for designing multi-selection.
- The final requirement of an artist is to conveniently transfer the local edits from one model to other models, e.g. with a copy-paste user experience. Therefore we describe a novel method that does not require user assistance even in the case where the derivation trees have different topologies.

2. Previous Work

Procedural modeling is a popular model to generate complex objects or larger environments, for example plants [PL90], urban street networks [PM01], roads [GPMG10], facades [WWSR03], buildings [MWH*06], parcels [VKW*12], rollercoasters [KK11], and terrains [GGG*13]. In this paper, we are mainly concerned with methods to control procedural modeling. One idea is to define external attributes that can be queried by the procedural model [PJM94, PM01].

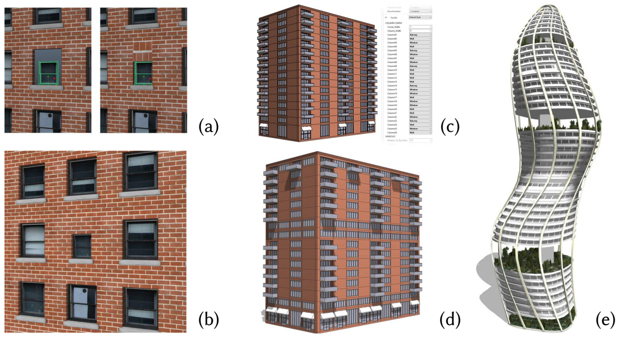

Figure 2: *Local editing of procedural models is still a challenge. (a) In modeling tools like Maya, the editing of the the scene graph of the procedural model can result in holes or texturing problems. (b) in our approach the window dimension can be locally edited without problems. (c) In procedural tools like CityEngine, authors have to define countless attributes to control for example every window ornament, leading to unintuitive UIs and complex procedural systems. (d) in our approach, the artist has full artistic control over the facade appearance. (e) Our local edits can be applied to arbitrary shapes or hierarchies. Unlike [LWW08], our local edits allow unnested grids (e.g. caused by ornaments) and do not require special annotations.*

A significantly more difficult coupling of user input and procedural modeling are techniques that aim to find a procedural model with certain properties, sometimes referred to as inverse procedural modeling. Several techniques propose combinations of probabilistic sampling or optimization techniques [TLL*11, VGDA*12, RMGH15, SW14] to find parameters of a procedural model with a fixed structure. A second approach is to optimize for the structure of a grammar given some input shapes, e.g. [TYK*12, MVG13, WYD*14]. One particular form of inverse procedural modeling is to control a procedural model by sketching [NGDA*16, HKYM17]. All these techniques are a first step to bringing existing models into a more editable form by converting them to procedural descriptions. Therefore they can be used as input to our approach, which uses forward procedural modeling.

Another concept is to enable the user to change a model while it is derived, to interleave procedural and interactive editing. This has been proposed for plant modeling [HBDS17], street modeling [CEW*08], and ecosystem modeling [EVC*15].

Another related research question is how to edit the (rules of) procedural models directly. In recent years, graph-based procedural modeling systems became popular, e.g. [Pat12, SEBC15, BBP13]. Another idea is the generation of procedural sub-models that can be interacted with as separate components [LHP11, KK12]. Our work also uses a user interface and procedural handles [KWM15, Hav05] to interact with a procedural model so we build on some of this recent work. However, the actual design of the user interface is not a focus of our paper.

Our work is mainly concerned with persistent edits in procedural modeling, which are preserved if a model needs to be regenerated

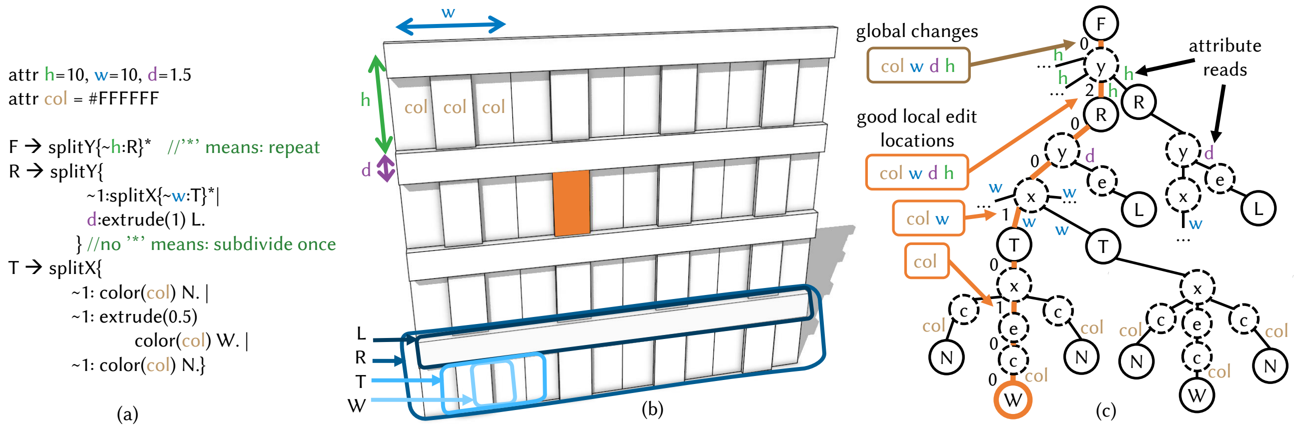

Figure 3: *(a) These rules generate a facade F containing row R, ledge L, tile T, and window W, as seen in (b). (c) An abbreviated derivation tree with an orange selection W is shown. The numbers along the orange path are the treekey. We find GELs* independently *for the attributes.*

under different starting conditions. Lipp et al. [LWW08] introduced semantic locators to persistently store and apply local edits. Jesus et al. [JPCS18] extended this idea using a query and example based interface for more complex selections.

In contrast to our meta-locators, semantic locators require annotations in the grammar, and have dependencies between attributes. Also, choosing meaningful semantic locators requires manual trial-and-error exploration in contrast to our automatic detection of GELs. The transfer algorithm of Lipp et al. can only handle single selections, and needs manual decisions on when to transform.

A key advantage of editing procedural models is that they enable high-level semantic edits. A related research area in shape modeling is to analyze geometric models so that similar higher-level editing operations become possible [GSMCO09,LCOZ*11,BWSK12].

Our problem statement can also be seen as a special case of shape matching. For example, Tevs et al. [THW*14] propose a framework for matching shapes via geometric symmetries and regularities and Alhashem et al. [AXZ*15] propose a discrete shape matching algorithm for two input shapes. In contrast to these other problem statements, we are able to leverage structural information provided by the rule-based procedural model to get more robust results.

Rule-based modeling Background Müller et al. [MWH*06] introduced *CGA*, a procedural system targeted at building generation. In CGA, starting at a shape, consisting of a label, attributes and geometry, the system searches for a rule with a matching label, and applies the rule. Rules have a label and one or more operations, which can use attributes. They output refined shapes.

For example, in Figure 3(a), the rule with label `F` has the operation `splitY` which reads the attribute h and splits the input shape into multiple shapes with label `R` along the y direction. When no matching rule label is found for a shape label, it is a terminal shape. The union of the geometry of all terminal shapes will yield the final model (`N`, `W` and `L` in Figure 3). The process of getting from the start shape to the terminal shapes is called *derivation*.

During derivation we build a *derivation tree*. In order to support nested operations, we store both *shape nodes* (e.g. `F` or `R` in Figure 3(c), shown as a solid circle) and *operation nodes* (e.g. `splitY`, which is abbreviated as `y`, shown as a dashed circle) in the derivation tree. When an operation reads or writes attributes while generating a shape, we mark the generated shape with those attributes. For example, in Figure 3 the attribute h is read while executing the operation `y`, thus we mark the shape `R` with h.

A node can be uniquely identified in a derivation tree: First the path from the root to the node is determined. Then, for each node on this path, its index in the ordered list of siblings is extracted, and added to a list. We call this list the *treekey* of a node, Lipp et al. [LWW08] named it exact instance locator. In Figure 3(c) such a path is highlighted in orange, and the treekey is $(0,2,0,0,1,0,1,0,0)$.

Local Edits Without loss of generality, we specify a local edit as writing the value v to attribute a at a specific treekey tk during derivation. Such a local edit is defined by the tuple $l = (tk, a, v)$, and our implementation is capable of applying local edits during derivation of the procedural model, thus the derivation process includes local edits in addition to the initial shape and rules.

3. Finding Good Edit Locations (GELs)

We would like to set out to make three fundamental observations about local edits that are important to define GELs. In our work, a GEL is specified per attribute, so that each attribute has a different set of GELs. First, there are many locations where applying an edit has no effect. For example, in Figure 3(c), editing the attribute h at treekey $(0,2,0)$ will be without effect, because h is not read below. These locations in the tree are unsuitable as GELs. Second, there are many redundant locations where applying an edit leads to the same modification, for example, in Figure 3, editing d at treekey $(0,2,0,1)$ has the same effect as editing it at treekey $(0,2)$. For these edit locations, it makes sense to select the edit location highest up in the tree as representative. Third, there are many combinations where writing an attribute in multiple locations has the same effect as writing it in one location. For example in Figure 3, setting the color *col* at multiple treekeys $\{(0,2,0,0,0),(0,2,0,0,1),(0,2,0,0,2),(0,2,0,0,3)\}$ is the same as setting it at $(0,2)$. For specifying local edits, it is better to store as few locations as possible, so we prefer to store a single edit instead of a set of edits whenever possible.

To describe *GELs*, we use the concept of *read coverage*. We define the read coverage of a specific attribute a at a tree node n as the number of times the attribute is read in the subtree below n. If there is a write access to attribute a in the subtree below n, all reads of attribute a below the write location are not counted. It is necessary to ignore all reads below writes, because they will not be affected by any edit above the write.

The concept of read coverage can now be used to define a GEL for attribute a, based on the observations above. GELs for attribute a are all locations in the tree where the read coverage increases.

Algorithm 1 implements this by computing the read coverage bottom-up, independently for each leaf and attribute. This independence has the advantage that the edit granularity can be different for different attributes. For example in Figure 3(c), the attribute d can only be edited uniquely at treekey $(0,2)$, but the color *col* has multiple good locations.

Algorithm 1 findGoodEditLocations($leaf, a$)

$node = leaf$
$numLeaves = 1; coverage = 0$
if a in $node.attrs_{read}$ **then** $coverage$++
$locations = \emptyset$
while $node.parent \neq null$ **do**
    $l_{new} = 0; c_{new} = 0$
    calcCoverage($node.parent, a, l_{new}, c_{new}$)
    if $c_{new} > coverage$ and $l_{new} > numLeaves$ **then**
        add $node$ to $locations$
    $node = node.parent$
    $numLeaves = l_{new}; coverage = c_{new}$
return $locations$

Algorithm 2 calcCoverage($node, a, numLeaves, coverage$)

if a in $node.attrs_{read}$ **then** $coverage$++
if a in $node.attrs_{written}$ **then return**
if $node.children == \emptyset$ **then** $leaves$++
for $child$ in $node.children$ **do**
    calcCoverage($child, a, numLeaves, coverage$)

Another example of GELs is shown in Figure 3. When one window W is selected, the GELs are at treekeys $(0,2,0,0,1,0,1)$ for *col*, at $(0,2,0,0,1)$ for *col* and w and at $(0,2)$ for *col*, w, d, and h. Note that the previously mentioned redundant locations are automatically ignored. Results of edits at GELs are shown in Figure 4.

User Interface for Edits at GELs (note: changed subsection to paragraph)

Our user interface enables artists to specify edits at GELs. To select one GEL, the user clicks on a leaf shape. For each attribute, we

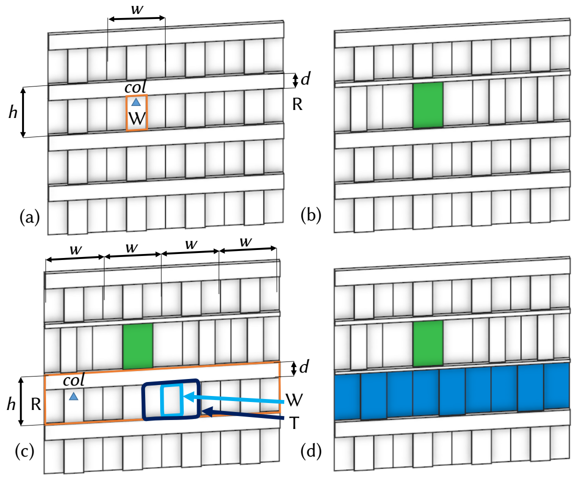

Figure 4: *(a) For the selected shape, GELs are determined for each attribute, and handles are displayed. (b) All handles were modified. Note the different granularity, e.g. changing the color col only affects the window while changing h affects the floor. (c) Selecting one window, then stepping up two GELs increases the scope of the handles w and col to the row R. Changing them results in (d).*

ascend towards the root in the derivation tree until we find the first GEL. This can be a different location for each attribute. We display procedural handles [KWM15] at those locations to enable interactive modifications of attributes. Dragging a handle will create or update a local edit.

For example in Figure 4(a), which is based on the rules in Figure 3, we display the following handles after clicking the orange leaf: for height h a handle is shown for the whole row R, while for the color *col* a handle (blue triangle) is shown at the window W. We highlight the shape corresponding to the deepest edit location in the derivation tree, which is the window W.

Increasing the scope Using a hotkey the user can step up to the next higher GELs on the path towards the root. This can be done until arriving at the root node, thus degenerating into a global edit. For example, in Figure 4(c) one window W was selected, then step up was clicked twice: First, the color handle *col* moves up to a tile T. Second, both the handle for *col* and w move to the row R. On the next step up, all handles would affect the whole facade. Note that it is not possible to select a whole column this way, because there is no GEL representing a column. A way to achieve this is using group selections using wildcards. These selections will be described at the end of the next section.

4. Meta-locators

One GEL can be uniquely addressed using a treekey. However, treekeys are not robust regarding changes in the derivation tree. Changes in the derivation trees can occur in a number of situations, for example when a global attribute, the initial shape, or the rules change, as shown in Figure 6. In order to preserve local edits in such situations, we need a more general way to describe the GEL.

The main idea is to describe the context of a GEL using multiple local context functions. Then we construct a *meta-locator*, based on the results from those functions to identify a GEL. To describe a set of GELs we introduce combination functions for meta-locators. As the GELs are created bottom-up per attribute, the meta-locator is also a bottom-up description with adaptive granularity.

4.1. Local Context Functions

Given a GEL n and attribute a, we define the local context $L(n,a)$ as the list of all siblings (including n) which are a GEL for attribute a. The filter using a is done to avoid influence from other attributes, for example adding an ornamented ledge with a new attributes should not influence the floor numbers.

A local context function $r = c(s,a)$ calculates a unique rational number $r \in \mathbb{Q}$ for a subset of siblings $s \in LS(n,a)$ where the domain $LS(n,a)$ is a subset of $L(n,a)$. c can be undefined for some elements in $L(n,a)$. The simple *local index* context function: $c_{idx}(n,a)$ operates on $LS = L$ and returns the order position that n has in L.

We define multiple *direction context* functions. They analyze the bounding box centers of nodes in L. For the c_x context function we project all siblings along the x direction of the parent bounding box. If all projected positions are unique within a certain threshold, we order the positions along x and return the order index of n. Otherwise c_x is undefined. Analogously c_y and c_z define indices with regard to the y and z directions.

The *component index context* function c_{comp} analyzes if a component split, as introduced in CGA [MWH*06] was performed on the parent shape. A component split separates a shape into multiple components with reduced dimensions, for example a box into six faces. First we look at the dimensions of the bounding boxes of all siblings in L. If all of them have a zero dimension where the parent is non-zero, we assume it is a component split, and the local index is returned. Otherwise the function is undefined.

For component splits, the *component orientation context* functions analyze the angle of the component normals with respect to global directions. The function c_{south} operates on the domain LS_{south} including all siblings facing south within some threshold. c_{south} orders the elements in LS_{south} along the vector orthogonal to south and up, which is the east vector. When n is included in LS_{south}, c_{south} returns the order index of n, otherwise it is undefined. Analogously $c_{north}, c_{east}, c_{west}$ return indices for other directions.

For all functions, we also define the composite functions *percentage* and *reverse*: Given the maximum value r_{max} for a function c with a given domain LS, we can define a function c_p returning percentages for c: $c_p(n) = c(n) * 100/r_{max}$, and the reverse function $c_{reverse} = r_{max} - c(n)$.

4.2. Constructing Meta-Locators

A meta-locator m identifies a GEL n for a given attribute a. It is a list of pairs $(c_j, c_j(p,a))$, containing a context function c_j, and the result $c_j(p,a)$ for nodes p which are GELs of a along the path from the root to n. We also add pairs $(c_j, *)$ for the path from n to the deepest child, where the wildcard $*$ indicates that all children are chosen. While the children are not necessary to locate n in this specific derivation tree, they help when transforming m to a different derivation tree where parent and child relations might be reversed.

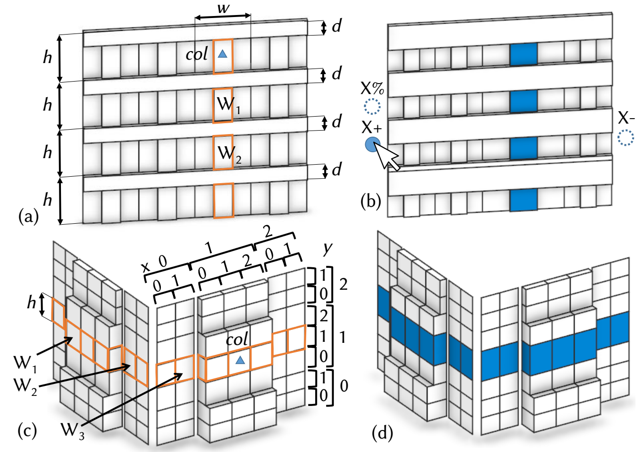

Figure 5: *(a) Selecting two windows W automatically expands to the column. (b) Handle changes affect the column. Also showing example of function override UI. (c) Selecting three tiles expands to a complex selection, which is adjusted using one handle (d).*

To construct m, we walk from the root towards n, and at every node p that is a GEL of a we look at the defined local context functions. Note that multiple context functions c can be defined for a given p. However, as the results of all defined functions need to be unique, choosing one function is enough to uniquely locate a GEL in a given derivation tree. The chosen function is added to the meta-locator together with the evaluated value.

Which function to choose depends on the desired behavior when the derivation tree changes. For example, the composite function $c_{reverse}$ makes sure edits stay relative to the last sibling. We provide both an automatic heuristic to choose a function c_j, and a user-interface to allow overriding of the choice c_j to a different function c_k that is defined for p.

By default we first set the component index function as the choice c_j, if this is undefined the direction and then orientation, and finally the local index function. When there are multiple directions (or orientations), we choose the one where the projected positions have the biggest (or angles have the smallest) variation. The user interface to override the choice c_j to c_k, for example to choose a composite function, is shown in Section 4.5. Overrides of c_j to c_k are always applied to all instances of c_j in the meta-locator (but they can vary between different local edits), as long as c_k is defined in the corresponding node. Override decision are stored alongside the meta-locator in order to apply it again on derivation tree changes.

Example The first GEL of the attribute c along the path to the shape T with treekey $(0,2,0,0,1)$ in Figure 3 is at treekey $(0,2)$.

We evaluate the context functions here. As there is a y split, the direction function c_y return a defined value, and will be chosen by our heuristic. The row R is the third row when ordering the siblings along the parents y direction. Therefore c_y evaluates to 2, one less because our counting is zero-based. The pair $(c_y, 2)$ will be added to the meta-locator m. Then, the context functions for the second GEL at treekey $(0,2,0,0,1)$ are evaluated. As there is a x split, the context function c_x will be chosen. Adding the pair of c_x and the order 1 creates the final meta-locator $((c_y,2),(c_x,1))$. Now add pairs for the path to the deepest child to the list, replacing all results with the wildcard $*$. The meta-locator is: $((c_y,2),(c_x,1),(c_x,*))$.

4.3. Combining Meta-Locators

To describe sets of GELs, we define the range $f_{range}(c, r_{min}, r_{max})$, even f_e, odd f_o, and wildcard $f_*(c)$ functions, which can be used instead of a specific result $c(p,a)$ in a meta-locator. In the meta-locator pairs we store them using the short-hand notation $(c, [r_{min}, r_{max}])$, (c, e), (c, o), and $(c, *)$.

Such meta-locators are constructed by combining two meta-locators m_1 and m_2. First the meta locator with the longer or equal list m_l and the one with the shorter list m_s out of m_1 and m_2 is selected. Then, we find pairs in m_s, where the corresponding pair in m_l uses the same function c, but has different results r_s and r_l. For the range function, such pairs are replaced with $(c, [min(r_s, r_l), max(r_s, r_l)])$, for the wildcard function with $(c, *)$. If r_s is different to r_l, but both are even or odd, (c, e) or (c, o) are used.

Note that to actually apply a meta-locator m describing a set during derivation, it must be unrolled to a set M of individual meta-locators first. For every pair in m containing $f_{range}(c, r_{min}, r_{max})$ we find the set of results $R = c(s,a) | s \in LS(p)$. Then, for each result $r \in R | r_{min} < r < r_{max}$ a copy of m is created, where we replace the pair with (c, r). For the wildcard function f_* this is similar, however all elements $r \in R$ are taken.

Example The meta-locator for shape W_1 in Figure 5(a) is $m_1 = ((c_y,2),(c_x,2),(c_x,1))$, and for W_2 it is $m_2 = ((c_y,1),\ (c_x,2),\ c_x,1))$. The first pair has matching functions c_y but mismatching results $2 \neq 1$. Therefore we construct meta-locators $((c_y,*),(c_x,2),(c_x,1))$ which can be unrolled to $((c_y,Y_1),(c_x,2),(c_x,1)) \forall Y_1 \in (0,1,2,3)$. This represents a whole column.

Figure 5(c) is a more complex example with two x and y splits. The shapes W_i have meta-locators

$((c_{comp},0),\ (c_x,1),\ (c_x,0),\ (c_y,1),\ (c_y,1))$ for W_1,

$((c_{comp},0),\ (c_x,2),\ (c_x,0),\ (c_y,1),\ (c_y,1))$ for W_2 and

$((c_{comp},1),\ (c_x,0),\ (c_x,1),\ (c_y,1),\ (c_y,1))$ for W_3.

Combining them selects a floor across all facades:

$((c_{comp},*),\ (c_x,*),\ (c_x,*),\ (c_y,1),\ (c_y,1))$.

Mapping When calculating the meta-locator for all GELs we obtain a mapping: $tk \leftrightarrow m$, i.e. a meta-locator m can be mapped to a treekey tk and vice-versa, for all GELs in a given derivation tree. Therefore we can write the local edit tuple as $l = (m,a,v)$.

4.4. Write Local Edits Back Into Rules

Storing local edits as tuples $l = (m,a,v)$ separately from the rules has the advantage that they can be transferred to different derivation trees and rules, as shown in chapter 5. As a disadvantage, every derivation tree change requires running the transfer algorithm, even when the rules stay the same. In this case it can be beneficial to write the local edits back into the rules. This way the transfer algorithm is no longer required on derivation tree changes.

Automatic Semantic Tags The idea is to add semantic tags [LWW08] based on the meta-locator m to the rules. Note that the functions $c_{idx}(p,a)$ in m can *not* be evaluated during a top down derivation, because we do not know if an attribute will actually be used later in the derivation, therefore we do not know which siblings are GELs. However, $c_{idx}(p,a)$ can be approximated by assuming all siblings are GELs. For example in CGA, the `split.index` attribute can be used to approximate the direction context functions c_x, c_y and c_z. This can result in a different local edit position if for example ornaments are added, which is a general limitation of semantic tags as introduced by Lipp et al. [LWW08].

For every context function c_j in m, except the last one, an attribute assignment using `set(cj, aj)` is added to the rule at the position where the coverage increase happens. `aj` is an approximation of $c_j(p,a)$. For example, in Figure 3 the coverage for attribute `col` increases three times, so we add assignments for the first two times. This results in the following rules: `R → set(cy, split.index)) splitY(...)` and `T → set(cx, split.index) splitX(...)`. Then, for the last context function, an assignment of v to a is added after a case statement which tests against ai. For example, `extrude(0.5) color(col) W` is changed to `extrude(0.5) (case(cy==2 && cx==1) set(col, v)) color(col)) W`. When wildcards occur as function results, the corresponding case statement is omitted.

4.5. User Interface for Sets of GELs

To design meta-locators representing a set of GELs, the user first selects two or more leaf shapes. Then, we find GELs for those leaf shapes, and calculate their respective meta-locators. Depending on an optional modifier key the user presses, the meta-locators are either combined using ranges or wild-cards.

For example, in Figure 5(c) the user clicks on three shapes. The meta-locators are combined, and the unrolled result is highlighted in blue, as shown in (d). Note that step-up is a special case of wildcards: Clicking on two windows on the same floor in Figure 5(a) and combining them with wild-cards would select the same GEL as clicking on one window, then stepping up once. We prefer using wild-card selections, because they make the split order (floors or columns) transparent to the user.

If multiple context functions match for a specific node p, we show an icon representing the automatically chosen function next to the first node $p_1 \in LS(p)$. In Figure 5(b), c_x was chosen by the heuristic, but the composite functions percent and reverse are also defined. If the user drags the circle below c_x, dashed icons for the other defined functions are shown next to their respective first node $p_1 \in LS(p)$. Dropping the circle over a icon defines an override.

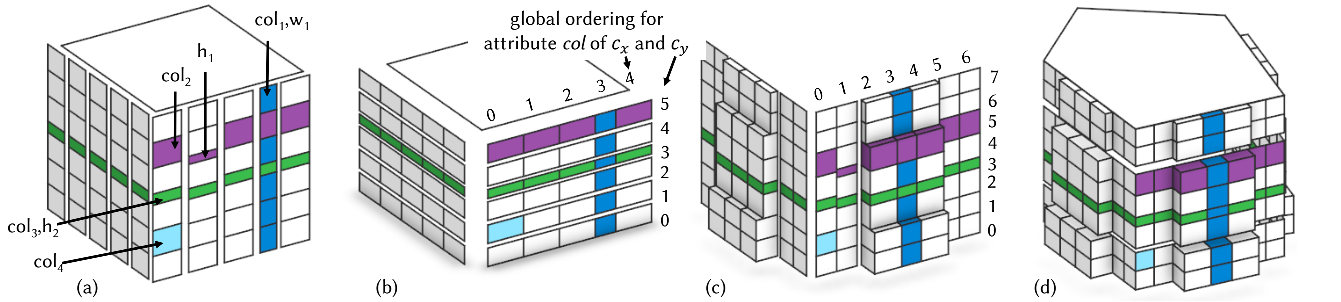

Figure 6: *Local edits of the color col_i, width w_i and height h_i are applied (a). Those edits are transferred to a house having a different rule with switched y/x split order and lower height (b), one with nested splits (c), and one with nested splits and a different initial shape (d).*

5. Transfer of local edits between derivation trees

The problem of edit transfer can be described as follows. We are given a set of local edits for a *source* derivation tree and we would like to find a corresponding set of local edits for a *target* derivation tree. The core of this problem is matching tree locations from the source tree to the target tree. To tackle this problem we propose to enlist the meta-locators described in the previous section. We first describe the matching for meta-locators describing a single location and then extend to meta-locators describing a set of locations, i.e. meta-locators including wildcards and ranges. Given a local edit $l_s = (m_s, a, v)$ in the source tree at a location specified by meta-locator m_s, we want to find a matching tree location in the target tree specified by meta-locator m_t. Essentially we have to find a mapping $m_s \to m_t$, for each local edit.

As a first step, we calculate all meta-locators m_t of all GELs in the target derivation tree using local context functions and choosing one based on the heuristic introduced in 4.2.

Then, for each local edit, we try to find an m_t that *matches* m_s. If m_t contains overrides of selection functions (which can be different for each local edit), we also apply those overrides on m_t. When a match is found, we execute the local edit in the target derivation tree. The challenges are to decide when meta-locators are matching, how to handle different derivation tree structures, and how to prioritize matches when multiple matches are found.

Exact and unordered matches Two meta-locators m_s and m_t match *exactly* when all list entries are the same and have the same order. This is an ordered match with priority 1. In case the meta-locators have the same pairs, but their order is different, we consider this an *unordered* match with priority 2. The motivation behind this match is that this typically indicates that the source and target model are very similar, but that the two derivation trees create different hierarchical structures for these models. An example for an unordered match occurs when transferring edit col_4 from Figure 6(a) to (b). The first derivation tree splits the facades first into columns and then into floors and the second derivation tree splits the model first into floors and then into columns. The meta-locator for edit col_4 is $((c_{comp}, 1), (c_x, 0)), (c_y, 1))$ in building (a) and is matched to meta-locator $((c_{comp}, 1), (c_y, 1)), (c_x, 0))$ in (b).

Handling different structure When the structure of the derivation tree has major differences, it is very likely that there are no exact or unordered matches. To detect such differences we count how often each context function c occurs both for m_s and m_t.

Whenever a count for c is different between m_s and m_t, there is a structural difference. For example, when considering a transfer from Figure 6(a) to (c), we can observe that the facade in (c) has more entries for the c_x and c_y function. The edit col_4 in Figure 6(a) has the meta-locator $m_s = ((c_{comp}, 1), (c_x, 0), (c_y, 1))$. The count for c_{comp}, c_x, and c_y is one each. In the building 6(c) the meta-locator $m_t = ((c_{comp}, 1), (c_x, 0), (c_x, 0), (c_y, 0), (c_y, 1))$ also has a count for c_{comp} of one, but two for each c_x and c_y. Therefore the counts for c_x and c_y mismatch between m_s and m_t.

For every function with mismatching count, we compute the global order index using the algorithm introduced by Lipp et al. [LWW08]. Their algorithm computes a global ordering index using a post-order traversal of the derivation tree. For example, the global order of the edit col_4 with meta-locator m_s in Figure 6(a) is 0 for c_{comp}, 1 for c_y and 0 for c_x. The global orders match for meta-locator m_t in Figure 6(c). This algorithm is performed on meta-locators, and therefore the result is independent for each attribute.

When global order matches are found, we construct a new meta-locator m_{sG} by first copying it from m_s. Now, for every function c with mismatching count, where a global order match was found, we replace all pairs referencing c in m_s with the ones from m_t. In the previous example this results in $m_{sG} = ((c_{comp}, 1), (c_x, 0), (c_x, 0), (c_y, 0), (c_y, 1))$, which is equal to m_t. Finally, we test for an unordered match between m_{sG} and m_t. If it passes, we call it a global order match and set priority 3.

Not matchable edits There are multiple cases for which a meta-locator cannot find a match. This can happen if the meta-locator describes a location that does not exist in the target derivation tree (e.g. the fourth floor in a two-storey building) or it describes an attribute that cannot be changed in another structure. For example, the edit h_1 in Figure6(a) modifies the height h of one tile. This edit is not possible in building (b), because it is first split into rows along y with heights h and then along x into columns. Therefore h can only be edited per row. This also means that editing h per tile is *not* a GEL in Figure 6(b), because it has no effect, therefore it will not be found at all in Algorithm 1.

5.1. Handling sets of edit locations

Now, we extend to matching local edits at a set of locations, described by meta-locators that include wildcards or ranges, to another set of locations.

For ranges, we simply create two meta-locators, one for the start and one for the end of the range. Then we transform those individually. For wildcards we consider two cases, both times using the globally matched meta-locator m_{sG} as source:

First, a meta-locator can include wildcards exclusively in pairs at the end of the list, but not in the middle. These meta-locators are typically generated by the step up operation described in Section 3, but they can also be generated by simple *multi-selections* as described in Section 4.5. While these meta-locators describe a set of attribute usages in different parts of the derivation tree, the edit can be accomplished by changing an attribute at a single tree location higher up in the tree. Figure 6 shows two such examples, the edits col_1 and w_1. The meta-locator for col_1 is $((c_{comp},0),\ (c_x,3),\ (c_y,*))$.

Since our meta-locators already include trailing wildcards for children, this case can be handled with the previously described algorithms without further changes.

Second, a meta-locator can include wildcards in pairs somewhere in the middle of the list. These meta-locators are typically generated using *multi-selection* as described in Section 4.5. For example, the edits col_2, col_3, and h_2 in Figure 6 were created using multi-selection. The meta-locator of edit col_2 is $((c_{comp},0),\ (c_x,*),\ (c_y,5))$. This meta-locator is *not* representable by one GEL. To find this type of meta-locators, we start from meta-locators in the target derivation tree without wildcards and meta-locators that have wildcards only in trailing pairs and enumerate all possible wildcard placements in the middle. For example, given a meta-locator with context function results $(0,1,2,*,*)$ we enumerate $(0,*,2,*,*)$, $(*,1,2,*,*)$ and $(*,*,2,*,*)$. The detailed enumeration algorithm is shown in Algorithm 4. If an ordered or unordered match is found with a permutation, it is a multi-selection match with priority 4.

For example, the edit col_2 in Figure 6(a) with meta-locator $((c_{comp},0),\ (c_x,*),\ (c_y,5))$ has an unordered match to meta-locator $((c_{comp},0),\ (c_y,5),\ (c_x,*))$ in Figure 6(b). Note that in (b) the x/y splits are swapped, thus a meta-locator with a wildcard in the middle can match a meta-locator with a wildcard at the end. The opposite happens for edit col_1 with locator $((c_{comp},0),\ (c_x,3),\ (c_y,*))$. It is matched to locator $((c_{comp},0),\ (c_y,*),\ (c_x,3))$.

Handling multiple edits It is possible for edits to be in conflict with each other. A simple conflict happens when one edit overwrites an attribute change effected by a previous edit. A more complex conflict happens when meta-locators using wildcards are used. Then a set of derivation tree locations described by one meta-locator can partially intersect the derivation tree locations described by another meta-locator. In general, we do not use explicit conflict handling, but let subsequent edits simply overwrite the values of previous edits. For example, in Figure 6(a) the edit col_1 looses against edits col_2 and col_3 because it is higher up in the derivation tree, while in building (b) it always wins.

Prioritizing multiple matches Matches are sorted by their assigned priority. In case of multiple matches having the same priority, the matches are next sorted by the amount of entries in the meta-locator list in descending order. This prefers meta-locators with more matching pairs. Then sorting proceeds using treekey length in ascending order, preferring edits higher up in the tree. Finally, the first match is chosen.

Algorithm 3 findMatch(m_s, $locations$)

$matches = \emptyset$
for m_t in $locations$ **do**
  if EXACTMATCH(m_s, m_t) **then**
    add m_t to $matches$ with priority 1
  else if UNORDEREDMATCH(m_s, m_t) **then**
    add m_t to $matches$ with priority 2
  else
    m_{sG} = GLOBALTRANSFORM(m_s, m_t)
    if UNORDEREDMATCH(m_{sG}, m_t) **then**
      add m_t to $matches$ with priority 3
    else if MULTISELECTIONMATCH(m_{sG}, m_t) **then**
      add m_t to $matches$ with priority 4
sort $matches$ by priority and |entries| in m_t and associated treekey
return first element in $matches$

Algorithm The matching is an iterative process. For each local edit we call Algorithm 3. The matching subfunctions are shown in Algorithm 4. For unmatched edits, we try again after all others were matched and executed. We might get more matches this way, for example when one edit increases the building height, and another one changes one floor which only appears because of the increased height. We iterate as long as we find new matches. It is possible that some meta-locators remain unmatched. This happens e.g. when a local edit addresses floor 6 but the building just has 4 floors.

Triggering edit transfers Local edits are stored as tuples $\{m,a,v\}$ with an initial shape. Whenever the derivation tree of the initial shape changes, the local edits need to be transformed to the new derivation tree using Algorithm 3. This is done automatically, e.g. when a rule changes, an initial shape changes, or when edits are added using the copy-paste functionality (described in Section 5.2).

5.2. User Interface for Transfers

We provide a user interface to *copy-paste* edits from one shape with a source derivation tree to another shape with a target derivation tree. It allows for copying of some edits between model parts, or copying all edits at once.

The user first needs to specify a location in the source tree using the method described in Section 4.5, resulting in meta-locator $m_{selCopy}$. On clicking on copy in the context menu, all edits are filtered with $m_{selCopy}$ and stored in the copy buffer. To filter edits, we take all edits whose meta-locators match $m_{selCopy}$ and all edits whose meta-locators match a specific wildcard replacement of

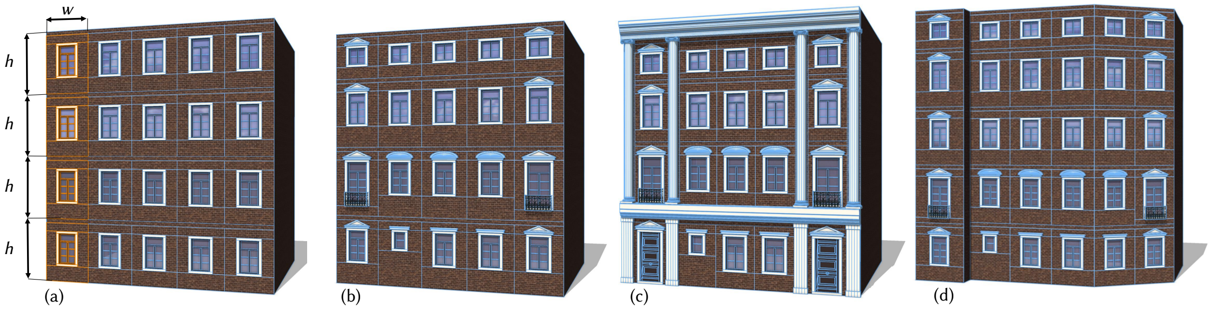

Figure 7: *Editing and automatic transfer of edits across topology changes. (a) Procedural facade encoded row-first which one column of tiles selected. (b) The same facade with some local edits applied. These edits are stable against topology changes as shown in (c) where a ground floor and pillars were enabled. (d) The edits from (b) are transferred to a building with more floors and a more complex footprint.*

Algorithm 4 Matching subfunctions, with input meta-locators $m_s = ((c_{s0}, r_{s0}), \ldots, (c_{sk-1}, r_{sk-1}))$ and $m_t = ((c_{t0}, r_{t0}), \ldots, (c_{tl-1}, r_{tl-1}))$ where c is a context function and r is a result of $c \in \mathbb{Q}$, $k = |m_s|$ and $l = |m_t|$

function EXACTMATCH(m_s, m_t)
  return true when $k = l$ and $c_{sm} = c_{tm}$
   and $r_{sm} = r_{tm}$ for all $m \in [0, k[$
function UNORDEREDMATCH(m_s, m_t)
  return true when $k = l$ and $c_{sm} = c_{tn}$
   and $r_{sm} = r_{tn}$ with $m = [0, 1, ..., k-1]$
   where n is any permutation of m.
function GLOBALTRANSFORM(m_s, m_t)
  $m_{sG} = m_s$
  for all unique functions c in m_s **do**
   calculate global order of c for m_s and m_t
   if global order matches **then**
    replace pairs using c in m_{sG} with the pairs from m_t
  return m_{sG}
function MULTISELECTIONMATCH(m_s, m_t)
  i = index of first wildcard in m_t - 1
  for $j = 1$ to $2^i - 1$ **do** ▷ Enumerate multi-selections
   $m_{wc} = m_t$
   for $k = 0$ to $i - 1$ **do**
    if bit k is set in j **then**
     set result of pair k in m_{wc} to $*$
   if UNORDEREDMATCH(m_s, m_{wc}) **then**
    return true
  return false

$m_{selCopy}$. Next, a target selection is defined, yielding meta-locator $m_{selPaste}$. On paste, we perform the following operation for each meta-locator m corresponding to an edit in the copy buffer. We change m to make it relative to $m_{selPaste}$ by replacing wildcard indices of the pairs in $m_{selPaste}$ with the index of the corresponding pairs in m. Finally, Algorithm 3 is used again to transform $m_{selPaste}$.

The example in Figure 8 has three local edits of colors. The green color edit col_{1S} is at location $((c_{comp}, 1), (c_x, 3), (c_y, 2))$. The selected row in (a) has meta-locator $((c_{comp}, 1), (c_x, *), (c_y, 2))$. All three edits are matched by the selection filter. In (b), the target selection with $m_{selPaste} = ((c_{comp}, 0), (c_x, *), (c_y, 1))$ is shown. In (c), the three edits were pasted, i.e. made target-relative, transformed and applied to the derivation tree. The final meta-locator for col_{1T} is $((c_{comp}, 0), (c_x, 3), (c_y, 1))$.

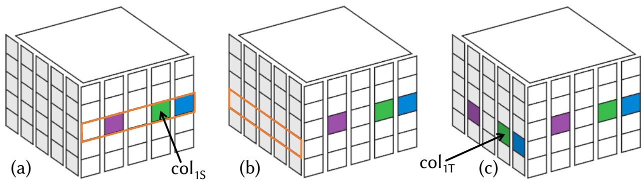

Figure 8: *Copy-paste of local edits from one floor onto another one: (a) A floor with local edits is selected and copied. (b) A target floor is selected. (c) The local edits are pasted relative to the target.*

6. Results and Discussion

Creating Variations of an Architectural Style Our methods have been used extensively in a project that was exhibited for 6 months at the Architecture Biennale in Venice, Italy. Many buildings were modeled in a geotypical look for multiple 3D cities, including an idealized version of Tel Aviv, Israel, in 1935, according to designs by city planner Patrick Geddes.

The artists encoded the basic style components in a procedural rule and then modeled the buildings using local edits (Figure 1). The edits included choosing the main facade layout and floor pattern, selecting balcony types, placing windows, etc.

This approach allowed them to explore and design the styles much faster than using either a fully procedural or fully manual approach. Using our hybrid approach keeps both rule-writing and manual actions to a minimum. The designers needed about one day for encoding the default style in rules, 5 minutes for manually creating the 3D mass model and 15 minutes for placing edits to create the reconstruction or a design variation. Manually modeling such a building would have taken them about 4 hours for each variation.

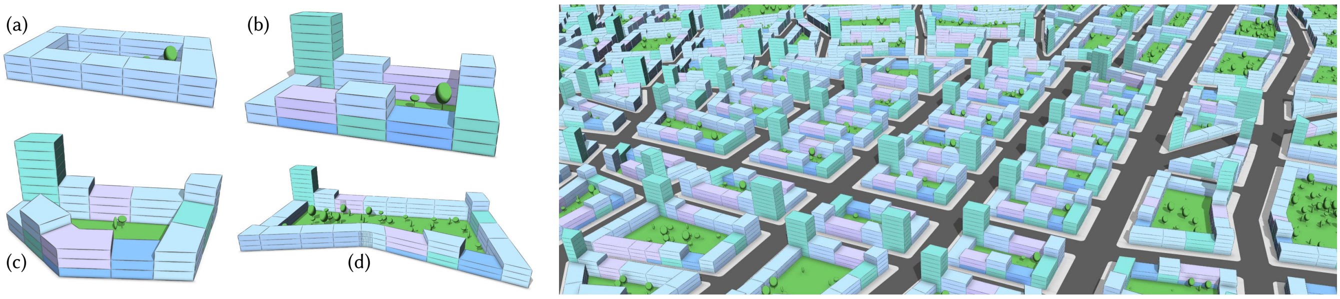

Figure 9: *Transfer of urban planning design. Left: (a) Rule which divides a block into units. (b) Local edits are used to design one block. (c), (d) The design is copy-pasted onto two different blocks, with different form and number of edges. Right: transfer to a whole city.*

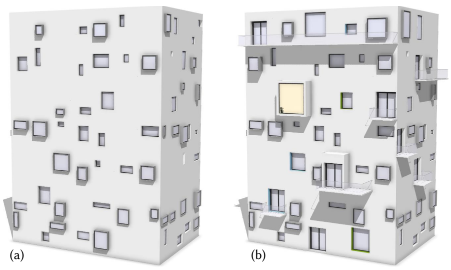

Figure 10: *Editing a Stochastic Building Facade. Left: The initial result of a stochastic rule. Right: The final model after being edited by an artist. Positions, sizes and appearance of some windows are corrected and doors and balconies are inserted.*

Refining Facades This example shows how the tools described in this paper make local edits robust against changes in the rule base. Figure 7(a) shows a procedural facade, encoded row-first. When clicking on a shape, we analyze the attribute usage and show handles for meaningful edit locations. The user can quickly select a column of tiles (orange highlight) using a second click. By dragging handles, local edits can be applied to create an interesting facade which would have required complicated if-case-statements in the procedural rules (b). These edits are stable against topology changes as shown in Figure 7(c) where a ground floor and pillars were enabled, resulting in a completely different topology with nested grids. Figure 7(d) shows the edits from (b) after being transferred to a building with more floors and a complex footprint, spanning multiple facades. Note how the edits stay in place even if the facade spans multiple faces and has more floors. This is possible using both the component orientation context functions and the reverse composite functions.

Editing a Stochastic Building Facade Procedural modeling is great to generate stochastic variations. However, artists often desire some local control. Local edits allow for this, as shown in Figure 10. Here, the rules generate a modern building with windows of random size and at random positions. The windows of the initial model on the left are edited to obtain the final result on the right. Windows are moved around, their size and appearance is changed, balconies are added, and doors are inserted. For example, one edit enables balconies on the whole top floor of the front facade. Defining and storing the edits separate from the rule is much easier and flexible than editing the rules and keeps the rules clean.

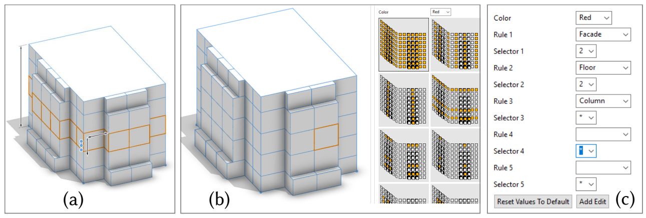

Figure 11: *We compared three local edit interfaces in our user study. (a) Ours: Selections automatically expand, and clicking on the handle applies the edit. (b) Example-based: On selection, multiple examples are shown. Clicking on a example applies the edit [JPCS18]. (c) Query-Based: Structured queries define the selection [JPCS18]. We provide a user interface for query construction.*

Transfer of Urban Planning Design The example in Figure 9 demonstrates how our method can be used to transfer local edits from one initial shape to other initial shapes. In this urban planning use case, a procedural rule is used to divide a block into units. Then, the block is manually designed with local edits: the heights are changed and usages (color-coded) are assigned to whole units or specific floors. An urban planning rule needs to be respected which allows at most one tower per block. Finally, in order to rapidly design the whole city, the edits are copied to all other blocks, most having a different number of edges than the original block.

6.1. Discussion

User Study We compared our user interface described in Section 4.5 with the example-based and query-based approaches presented in [JPCS18]. Figure 11 depicts those interfaces. The example- and query-based approaches require semantic tags, therefore we employed our automatic semantic tagging method introduced in Section 4.4 for them.

Eight users where given the task to replicate the local color edits

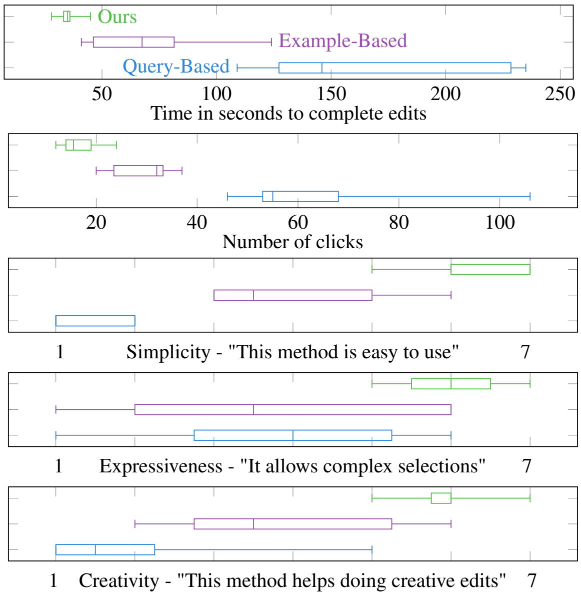

Figure 12: *Results of our user study (n=8). Qualitative results use a Likert scale (1 = strongly disagree, 7 = strongly agree).*

seen in Figure 6(c) using those three methods in randomized order, without knowing which method was ours. Four users had procedural modeling knowledge, the others a general computer graphics background. They were given a brief introduction and example for each method. Then the number of clicks and time to do the task was recorded, and are shown in Figure 12. It shows that our method required the least time and number of clicks.

The users also filled out a qualitative questionnaire. As shown in Figure 12, our method scores the highest. For simplicity, comments for our method were: "it feels natural" and "there is a direct connection". For the example-based method comments where "it helps to see what is possible", however also "disconnect makes it hard to see what will happen". For expressiveness, there were similar comments. However, for the example-based approach three users were concerned that it might not scale well for more complex buildings, due to combinatorial explosion. The query based approach is noticeable worse for simplicity and creativity. However for expressiveness results have a wide spread, also in the group with procedural modeling knowledge. Four people mentioned "it is only good for programmers", and two said "however it enables very complex selections".

Space of Possible Local Edits In Table 1 we show the total number of addressable edit locations for different methods. Lower numbers imply it is easier for a user to find non-redundant and meaningful edits. Additionally, we show the maximum number of locations when selecting a single leaf and stepping up to the root as described in Section 3. This directly correlates to how many clicks a user needs when stepping up to a specific selection.

With exact instance locators, as introduced by Lipp et al. [LWW08], every attribute can be changed at every derivation tree node using treekeys. Semantic locators [LWW08] use a heuristic based on scope aspect ratios to add semantic tags to split operations. Depending on the heuristic parameters, more or less splits are tagged, creating a range of addressable edits. We show both the worst case, where all splits are tagged, and the best case, where we manually added tags to only count split operations potentially creating more than one shape reading any attribute.

Table 1 shows a maximum of 19 potential edits for our method, when selecting a leaf in the model shown in Figure 7(c). Semantic locators have 28 to 98 potential edits. This is because semantic locators from Lipp et al. allow edits of the floor height and wall color on the window level, even when windows do not read those attributes. In general, it is preferable to have as few addressable locations as possible, provided all edits with *unique effects* are still possible. Two edits are distinguishable if the resulting leaf shapes vary in at least one attribute and they are redundant if they result in leaf shapes with identical attributes.

Our method consistently has the smallest number of addressable locations. We can even improve upon the manually tagged semantic edits, because we analyze attribute usage independently, therefore preventing cases where an attribute edit will not be read. Our method only presents edits with unique effect, while semantic tags also allow edits with either no effect or the same effect as other edits.

Comparison of Transfers In order to compare our transfer algorithm to the methods introduced by Lipp et al. [LWW08], we use copy and paste of multiple edits from one initial shape to another, and manually count how many transfers succeed. We assume that users want the global order of affected leaf shapes to be the same. Therefore a transfer is successful if it affects at least one attribute read, and affects the terminal shapes in the same global order (e.g. an edit affecting the second window must not shift to the third).

The transfer from Figure 7(b)→(c) fails for all previous methods, only ours passes, as shown in Table 2. This is because the added pillars create offsets in the global order. Our method handles this because order is calculated independently for attributes. The previous semantic approach works for Figure 6(a)→(b), but fails for all but two cases in Figure 6(a)→(c), because it does not support wildcard transformations combined with hierarchy changes. The transform from Figure 9(b)→(c) works for all methods. An interesting case is transfer 10(b)→(a) where exact locators have the best result, implying that exact locators work well when global attributes and rules remain the same. Three edits fail to transfer using our method. This is because the split detection based on bounding boxes fails due to the irregularity in the facade. Extracting splits from the grammar might improve this.

Note that the transfer method in Lipp et al. [LWW08] requires a manual tags and user decision whether to perform an exact or global order match. We manually searched the best case for the old approach in Table 2. By contrast, our method does not require manual tagging and works automatically.

Implementation and Performance We implemented our algorithms as plugin in CityEngine [Pro17]. For complexity analysis, we define n as the number of derivation tree nodes, a the number of

Fig.	All Edit Locations			Leaf Selection		
	Exact	Semantic	Our	Ex.	Sem.	Our
1r.	13170	30-445	30	300	5-30	5
7(c)	23751	203-5467	130	329	28-98	19
9(b)	1018	94-290	65	118	12-16	11
10(b)	266608	12928-55456	3336	784	128-208	53

Table 1: *Number of edit locations for our examples in total and for a single leaf selection, comparing three methods: exact locators (Ex.), semantic locators from Lipp et al. with a range from best to worst case, and our method.*

Figure	Edits	Exact	Semantic	Ours
7(b)→(c)	8	0.0%	0.0%	100.0%
6(a)→(b)	7	0.0%	100.0%	100.0%
6(a)→(c)	7	0.0%	28.6%	100.0%
9(a)→(b)	14	100.0%	100.0%	100.0%
10(b)→(a)	48	100.0%	75.0%	93.8%

Table 2: *Success rate when transferring edits from a source to a target, both absolute and in percent. For each edit we checked manually if it affects the attributes in the correct leaf shapes.*

attributes, l the number of GELs in the derivation tree and e_w and e the number of local edits with and without wildcards; s is the maximum number of splits in a branch of the derivation tree and limits the length of the meta-locators. To calculate GELs for all nodes and attributes, Algorithm 1 has complexity $O(n \cdot a \cdot 2)$ as every node is visited once for coverage calculation, and once for comparing coverage, provided that the coverages and GELs are cached once calculated. We also calculate the global order of every GEL during this traversal. Algorithm 3 to find matching meta-locators has worst case complexity (if no match is found and thus all branches are executed) of $O(l \cdot s \cdot log(s) \cdot (e + 2^{s-1} \cdot e_w)$. Note that the complexity of Algorithm 3 does not dependent on the number of attributes a, because the edits are already tied to specific attributes.

Timing results when performing a copy/paste operation, which uses the transformation algorithm, are shown in Table 3 (Hardware: Intel i7-6700 3.4Ghz, Nvidia 980). For the examples in Figures 6 and 9 the transformations can be performed interactively at about 10 frames per second, while in Figure 10 it is about 0.5 fps.

In order to allow for interactive editing in those cases, we limit how often we transfer as follows: We only transform when rules or the initial shape change, or treekeys of edits are no longer found in the derivation tree. In most cases this results in a good approximation of transforming at every parameter change, while only incurring the derivation time cost.

Figure	Edits	Total	Derive	Good Edits	Transf.
7(c)	8	100.5	54.1	46.2	0.2
9(b)	13	103.9	73.9	17.7	12.2
10(b)	48	1211.4	408.7	668.9	133.8

Table 3: *We first copied the local edits, then removed them from the models, and finally measured the times in ms it took for pasting local edits again. Split times are shown for the derivation, finding GELs (Algorithm 1) and transfer (Algorithm 3).*

Limitations and Failure Cases GELs are found by analyzing the actual derivation tree. If one subtree collapses to just one attribute usage, for example by setting the number of floors to one, local edits assigned to floors are no longer matched, even if the rule contains a floor split. When the number of floors is increased afterwards, those edits are matched again.

There is no special handling of recursions, therefore recursions potentially add one GEL for every invocation. This can result in edit locations that are similar to a multi-selection and therefore redundant, and causes the 19 possible edits for the model in Figure 9, as shown in Table 1. The same happens for exact locators and semantic tags introduced by Lipp et al. [LWW08]. Detection of recursions could alleviate this problem.

Choosing one context function in a meta-locator entry can be insufficient to uniquely identify a GEL. For example, choosing either c_x or c_z for the block subdivision in Figure 9 is not enough to identify blocks uniquely. A workaround is to add intermediate splits. Generally solving this would require multiple functions per entry.

When the structure mismatches for one context function, our transformation algorithm falls back to global ordering. This means that all hierarchical information for this function is ignored. This could be improved by using a ordering based on other context functions instead, or using extended context queries [SM15].

The transformation algorithm requires attribute names to match between derivation trees, and assumes that equality in names implies similar semantics of attributes. This can either result in lost edits, or edits placed at wrong positions.

Future Work To improve the attribute name mismatch limitations, it would be interesting to automatically detect similar attributes based on their effect on derivation tree properties using machine learning techniques. This would require a large database of procedural models. We intend to tackle this problem once we have collected enough training data.

6.2. Conclusions

We presented a novel approach for the local editing of procedural models, requiring no technical knowledge of the underlying rule system by artists. No cumbersome manual tagging, rewriting, or pre-processing of rules is necessary. Therefore, non-technical artists can use the system intuitively. This is achieved by leveraging the attributes of procedural systems. We analyze attributes to find GELs, which greatly simplifies the discovery of meaningful and non-redundant local edits.

To persist attribute edits at GELs we introduced meta-locators, defined upon local context functions. Combination functions on meta-locators enable an intuitive and robust multi-selection workflow. Meta-locators are evaluated independently per attribute and are thus more robust to derivation tree changes compared to previous work. Using a novel transfer algorithm, local edits are transferred to other procedural models without user assistance. We implemented our techniques as plug-in to CityEngine and demonstrated their usefulness in a user study and multiple real-world test cases.

Acknowledgements We thank Matthias Buehler, Cyrill Oberhaensli and Yulia Leonova for assessing the methods presented in this paper. Special thanks go to Yulia for providing the images for Figure 7 (Context: Her work in the 'Ideal Spaces Working Group' for the Architecture Biennale 2018 exhibit in Venice: 'Artificial Natures'). We thank Stefan Lienhard for reviewing and providing suggestions for improving this paper. This work was supported in part by the KAUST Office of Sponsored Research (OSR) under Award URF/1/3426-01-01.

References

[AXZ*15] ALHASHIM I., XU K., ZHUANG Y., CAO J., SIMARI P., ZHANG H.: Deformation-driven topology-varying 3d shape correspondence. *ACM ToG* (2015). 3

[BBP13] BARROSO S., BESUIEVSKY G., PATOW G.: Visual copy & paste for procedurally modeled buildings by ruleset rewriting. *Computers & Graphics* (2013). 2

[BWSK12] BOKELOH M., WAND M., SEIDEL H.-P., KOLTUN V.: An algebraic model for parameterized shape editing. *ACM ToG* (2012). 3

[CEW*08] CHEN G., ESCH G., WONKA P., MÜLLER P., ZHANG E.: Interactive procedural street modeling. *ACM ToG* (2008). 2

[EVC*15] EMILIEN A., VIMONT U., CANI M.-P., POULIN P., BENES B.: Worldbrush: Interactive example-based synthesis of procedural virtual worlds. *ACM ToG* (2015). 2

[GGG*13] GÉNEVAUX J.-D., GALIN E., GUÉRIN E., PEYTAVIE A., BENEŠ B.: Terrain generation using procedural models based on hydrology. *ACM ToG* (2013). 2

[GPMG10] GALIN E., PEYTAVIE A., MARECHAL N., GUÉRIN E.: Procedural generation of roads. *Computer Graphics Forum* (2010). 2

[GSMCO09] GAL R., SORKINE O., MITRA N. J., COHEN-OR D.: iwires: An analyze-and-edit approach to shape manipulation. *ACM ToG (Siggraph)* (2009). 3

[Hav05] HAVEMANN S.: Generative mesh modeling. *PhD thesis* (2005). TU Braunschweig. 2

[HBDS17] HÄDRICH T., BEDRICH B., DEUSSEN O., SÖREN P.: Interactive modeling and authoring of climbing plants. In *Computer Graphics Forum* (2017), Wiley Online Library. 2

[HKYM17] HUANG H., KALOGERAKIS E., YUMER E., MECH R.: Shape synthesis from sketches via procedural models and convolutional networks. *IEEE Trans. on Visualization and Computer Graphics* (2017). 2

[JPCS18] JESUS D., PATOW G., COELHO A., SOUSA A. A.: Generalized selections for direct control in procedural buildings. *Computers & Graphics 72* (2018), 106 – 121. 2, 3, 10

[KK11] KRECKLAU L., KOBBELT L.: Procedural modeling of interconnected structures. *Computer Graphics Forum* (2011). 2

[KK12] KRECKLAU L., KOBBELT L.: Interactive modeling by procedural high-level primitives. *Computers & Graphics* (2012). Shape Modeling International (SMI) Conference 2012. 2

[KWM15] KELLY T., WONKA P., MUELLER P.: Interactive dimensioning of parametric models. *Computer Graphics Forum* (2015). 2, 4

[LCOZ*11] LIN J., COHEN-OR D., ZHANG H., LIANG C., SHARF A., DEUSSEN O., CHEN B.: Structure-preserving retargeting of irregular 3D architecture. *ACM ToG* (2011). 3

[LHP11] LEBLANC L., HOULE J., POULIN P.: Component-based modeling of complete buildings. In *Proc. of Graphics Interface* (2011). 2

[LWW08] LIPP M., WONKA P., WIMMER M.: Interactive visual editing of grammars for procedural architecture. *ACM ToG 27* (2008). 2, 3, 6, 7, 11, 12

[MVG13] MARTINOVIĆ A., VAN GOOL L.: Bayesian grammar learning for inverse procedural modeling. In *Proc. of CVPR* (2013). 2

[MWH*06] MÜLLER P., WONKA P., HAEGLER S., ULMER A., VAN GOOL L.: Procedural modeling of buildings. *ACM ToG* (2006). 2, 3, 5

[NGDA*16] NISHIDA G., GARCIA-DORADO I., ALIAGA D. G., BENES B., BOUSSEAU A.: Interactive sketching of urban procedural models. *ACM ToG* (2016). 2

[Pat12] PATOW G.: User-friendly graph editing for procedural modeling of buildings. *IEEE Computer Graphics and Applications* (2012). 2

[PJM94] PRUSINKIEWICZ P., JAMES M., MĚCH R.: Synthetic topiary. In *ACM SIGGRAPH* (1994). 2

[PL90] PRUSINKIEWICZ P., LINDENMAYER A.: *The Algorithmic Beauty of Plants*. 1990. 2

[PM01] PARISH Y. I. H., MÜLLER P.: Procedural modeling of cities. In *Proc. of SIGGRAPH 2001* (2001). 2

[Pro17] PROCEDURAL INC. / ESRI INC.: Cityengine, 2017. 11

[RMGH15] RITCHIE D., MILDENHALL B., GOODMAN N. D., HANRAHAN P.: Controlling procedural modeling programs with stochastically-ordered sequential Monte Carlo. *ACM ToG* (2015). 2

[SEBC15] SILVA P. B., EISEMANN E., BIDARRA R., COELHO A.: Procedural content graphs for urban modeling. *International Journal of Computer Games Technology 2015* (2015). 2

[SM15] SCHWARZ M., MÜLLER P.: Advanced procedural modeling of architecture. *ACM ToG* (2015). 12

[SW14] SCHWARZ M., WONKA P.: Procedural design of exterior lighting for buildings with complex constraints. *ACM ToG* (2014). 2

[THW*14] TEVS A., HUANG Q., WAND M., SEIDEL H.-P., GUIBAS L.: Relating shapes via geometric symmetries and regularities. *ACM ToG* (2014). 3

[TLL*11] TALTON J. O., LOU Y., LESSER S., DUKE J., MĚCH R., KOLTUN V.: Metropolis procedural modeling. *ACM ToG* (2011). 2

[TYK*12] TALTON J. O., YANG L., KUMAR R., LIM M., GOODMAN N. D., MĚCH R.: Learning design patterns with Bayesian grammar induction. In *Proc. of UIST '12* (2012). 2

[VGDA*12] VANEGAS C. A., GARCIA-DORADO I., ALIAGA D. G., BENES B., WADDELL P.: Inverse design of urban procedural models. *ACM ToG* (2012). 2

[VKW*12] VANEGAS C. A., KELLY T., WEBER B., HALATSCH J., ALIAGA D. G., MÜLLER P.: Procedural generation of parcels in urban modeling. *Computer Graphics Forum* (2012). 2

[WWSR03] WONKA P., WIMMER M., SILLION F. X., RIBARSKY W.: Instant architecture. *ACM ToG* (2003). 2

[WYD*14] WU F., YAN D.-M., DONG W., ZHANG X., WONKA P.: Inverse procedural modeling of facade layouts. *ACM ToG* (2014). 2