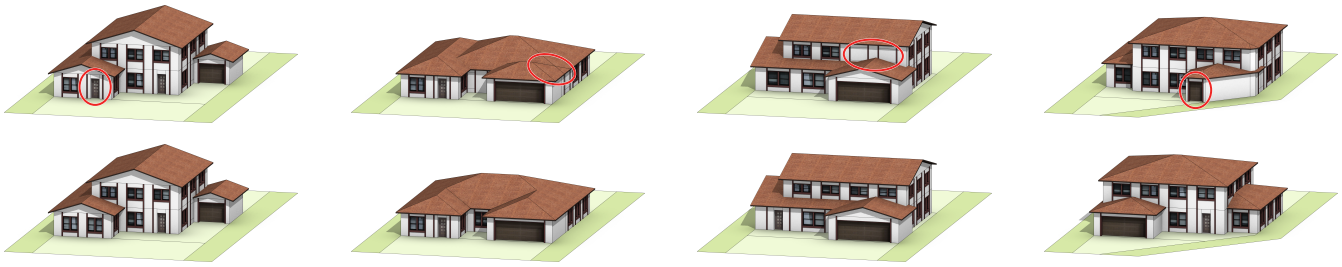
Advanced Procedural Modeling of Architecture

Michael Schwarz Pascal Müller

Esri R&D Center Zurich



(a) Multi-shape coordination (b) Boolean operations (c) Advanced context sensitivity (d) Best of several alternatives

Figure 1: *Our novel grammar language CGA++ enables many advanced procedural modeling scenarios not possible with previous solutions (top; bottom: ours), as exemplified with a grammar for residential suburban buildings comprising a main house, a wing, and a garage, and allowing different configurations of these. (a) With CGA++, modeling decisions can be coordinated across multiple shapes, e.g., to guarantee that overall exactly one door is created. (b) CGA++ enables operations involving multiple shapes, such as Boolean operations. Hence, masses can be merged to avoid overlapping geometries, allowing, e.g., one roof covering the whole building. (c) Generic contextual information can be obtained and acted on in CGA++, whereas previous solutions at best support a narrow set of context sensitivity. While they only allow canceling windows partially occluded, CGA++ additionally enables consistently adjusting all top floor windows. (d) Traditionally, only one alternative can be pursued during one specific derivation. CGA++, however, makes it possible to investigate multiple ones and choose the best of them. On a corner lot, the building grammar may fail if it executes only one option stochastically, and the selected one causes the garage to end up on an irregular footprint. CGA++ allows all options to be explored, robustly evading such failure cases.*

Abstract

We present the novel grammar language CGA++ for the procedural modeling of architecture. While existing grammar-based approaches can produce stunning results, they are limited in what modeling scenarios can be realized. In particular, many context-sensitive tasks are precluded, not least because within the rules specifying how one shape is refined, the necessary knowledge about other shapes is not available. Transcending such limitations, CGA++ significantly raises the expressiveness and offers a generic and integrated solution for many advanced procedural modeling problems. Pivotally, CGA++ grants first-class citizenship to shapes, enabling, within a grammar, directly accessing shapes and shape trees, operations on multiple shapes, rewriting shape (sub)trees, and spawning new trees (e.g., to explore multiple alternatives). The new linguistic device of events allows coordination across multiple shapes, featuring powerful dynamic grouping and synchronization. Various examples illustrate CGA++, demonstrating solutions to previously infeasible modeling challenges.

CR Categories: F.4.2 [Mathematical Logic and Formal Languages]: Grammars and Other Rewriting Systems; I.3.5 [Computer Graphics]: Computational Geometry and Object Modeling

Keywords: procedural modeling, grammar language

1 Introduction

Procedural modeling techniques are successfully employed in many domains, including urban planning, computer games, and movie production, for creating numerous instances of similar but varied objects with high detail. In the important case of buildings, typically a grammar-based approach is pursued, where a set of rules describe how one shape is refined into a set of new ones. Starting from an initial shape (e.g., a lot), these rules are iteratively applied, hierarchically evolving the structure of the model and incrementally adding details.

While impressive results of high visual complexity can be created with such grammars, several advanced modeling scenarios are not feasible with current grammar languages and systems such as CGA shape [Müller et al. 2006]. In particular, they exhibit the following fundamental limitations (Fig. 1 illustrates):

L1. Coordinating refinement decisions across multiple shapes is not directly supported and easily becomes impractical. Any decision that affects multiple shapes has to be made no later than at the point where these shapes' lineages diverge (and thus before these shapes exist at all) and then explicitly be passed on in rules. Notably, this implies that if this decision is influenced by properties of these (not-yet-existing) shapes or any other information only established later in the derivation process, these have to be inferred manually.

L2. Operations involving multiple shapes are normally not possible. For example, neither Boolean operations, such as forming the intersection of two shapes, nor assembling a shape from multiple shapes can be expressed.

L3. Contextual information is generally not available, precluding taking information from other shapes into account, as would be required for modeling objectives such as alignment. Merely for a few selected problems, ad-hoc solutions exist, such as limited occlusion queries [Müller et al. 2006].

L4. There is no support for exploring multiple alternatives (e.g., to choose the best one) or for performing auxiliary constructions (e.g., to derive parameter values). In particular, spawning a derivation from within a derivation and querying or embedding the result is not possible.

In this paper, we present a novel grammar language aimed at the procedural modeling of architecture that addresses and overcomes all of these limitations. In its design, particular care was taken to offer an integrated and generic solution instead of an external or ad-hoc one. Our language, called CGA++, is based on CGA shape [Müller et al. 2006] and evolves it carefully and in a natural way, raising its expressiveness and capabilities without deviating from or even abandoning its successful overall approach.

CGA++ introduces two new main language features. First, shapes become directly exposed in the grammar as first-class citizens; this entails a multitude of desirable consequences: Individual shapes can be uniquely identified as well as passed around and stored as values. Particularly, operations can take shapes as arguments, enabling Boolean operations on multiple shapes (addressing L2). It further becomes possible to access, traverse, and query the shape tree induced by the derivation process, which facilitates obtaining generic contextual information (L3). Importantly, also new temporary shape trees can be created on the fly, e.g., by spawning a new derivation or by invoking a function on an existing tree. This means that alternatives can be pursued and auxiliary shapes be utilized right from within a grammar (L4).

Second, a dynamic grouping mechanism and synchronization facility is provided with the linguistic device of events. It enables coordination across a group of shapes, such as exchanging information or establishing a consistent decision on how to proceed individually (L1). Moreover, events can be used to influence the order of the derivation process; in particular, this allows ensuring the availability of all shapes required when performing a contextual query (L3).

With CGA++, we are hence the first to provide a generic and integrated grammar-based solution for modeling of architecture that does not suffer from the limitations listed above. Our underlying contributions include introducing full first-class citizenship for shapes (Sec. 3) and events as a linguistic means for complex multi-shape coordination (Sec. 4). In addition, we present several further language elements that primarily aim at facilitating the ease of use and offering a concise syntax (Sec. 5). As we demonstrate by examples (Sec. 6), together these new features are extremely powerful and make a wide range of new applications possible. Among others, they enable

- operations using multiple shapes as input, such as Boolean operations, and decompose–refine–recompose workflows,
- creating auxiliary and sub-constructions as well as adopting and reasoning about their results,
- (generic) coordination across multiple shapes, and
- establishing and querying generic contexts, such as spatial adjacency of shapes.

2 Background

Various formal grammar systems have been proposed over time for procedural content creation. L-systems [Prusinkiewicz and Lindenmayer 1990] are parallel string rewriting systems, which are used for plant generation by interpreting the strings as Logo-style turtle commands. Multiple forms of support for context sensitivity have been explored for them (cf. Sec. 7.5). Shape grammars [Stiny and Gips 1972; Stiny 1980; Stiny 2006] are a powerful tool to explore, analyze, and understand spatial designs. They have been used in many domains, including architecture. A shape grammar consists of rules that replace one (sub)shape by another one, where geometric shape matching is performed. By contrast, set grammars [Stiny 1982] treat shapes as symbols instead of as geometric objects; they have been used in the form of split grammars for modeling façades [Wonka et al. 2003]. Drawing from this body of work, Müller et al. [2006] introduced CGA shape, a language for shape grammars[1] for the procedural modeling of architectural buildings. Numerous extensions and variants exist, addressing, for example, visual editing [Lipp et al. 2008], additional classes of shapes [Krecklau et al. 2010], interconnected structures [Krecklau and Kobbelt 2011], GPU-based execution [Steinberger et al. 2014], and specific application domains [Whiting et al. 2009; Schwarz and Wonka 2014] (cf. Secs. 7.4 and 7.5 for a detailed treatment). We adopted the latest incarnation of CGA shape [Esri 2014; Esri 2015] as basis for CGA++, including its syntax.[2]

In the following, we provide a brief review of CGA shape. It operates on shapes, treating them as (symbolic) objects. A shape comprises an oriented bounding box called scope and geometry (along with material information) inside this scope. Rules of the form α `-->` β determine how a shape is refined, where α is a symbol and β is a sequence of *actions* (symbols and operations), defining the successors of α. Classically, a shape is assigned a symbol, and if a rule whose left-hand side α matches this symbol is applied, the shape is substituted by the rule's right-hand side β. Not unlike the turtle command interpretation of modules in L-systems [Prusinkiewicz and Lindenmayer 1990], the actions in β are executed sequentially, where a symbol instantiates the shape in its current state and assigns it this symbol, while an operation modifies or subdivides the shape. Starting with an initial shape, this derivation process continues until no more rules can be applied; the remaining shapes define the final model. The hierarchical refinement performed during the derivation induces a tree of shapes, referred to as *shape tree*. Note that we can think of α as the rule's name and the appearance of such a name in β as an invocation of the according rule. Further, we refer to β as the rule's body.

One set of operations allows modifying the current shape, such as translating (`t`), rotating (`r`), or resizing (`s`) its scope, extruding the faces of its geometry (`extrude`) or building roofs on them (e.g., `roofGable`), and loading an asset as its new geometry (`i`). Another set of operations offers different ways of subdividing the shape, where actions can be specified for the resulting parts. For instance, the operation `split(`*axis*`)` splits the current shape along one axis of its scope, where the split pattern is given as a list of *size*`:`*actions* pairs. Syntactically, such lists are delimited by braces and use `|` as separator.

Each parameter provided for operations and rules (a symbol may have parameters) can be an arbitrary expression, which may involve functions. In addition to built-in ones, such as `sin`, it is also possible to define custom functions as part of the grammar. Both expressions and rule bodies may further utilize constructs for a conditional or stochastic selection among multiple options. Syntactic details are offered in the supplemental material, where we provide a concise reference for CGA++ that covers all language elements encountered in the examples presented throughout this paper.

3 Shapes as first-class citizens

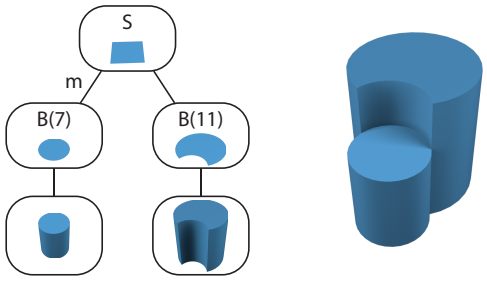
One main key to overcoming the limitations outlined in Sec. 1 is making both shapes and the notion of a shape tree explicitly available within a grammar. With the derivation process defining the shape tree, each shape occurring during the derivation corresponds

[1] Despite the name, the rule matching actually uses symbols.

[2] Generally, any existing CGA shape grammar constitutes a valid CGA++ grammar (barring some minor syntactical nuances).

```
S    --> i("circle") m=B(7) t(3,0,0) s(10,0,10) minus(m) B(11)
B(h) --> extrude(h)
```

(a) Grammar



(b) Shape tree

(c) Result

Figure 2: *Example of a grammar (a), showcasing labels and operations using other shapes (these new language features are highlighted in blue). The according derivation process defines a shape tree (b), whose leaf nodes determine the final result (c).*

to a node in this tree and hence further identifies the subtree rooted in that node. That is, the terms shape, (shape) node, and (sub)tree essentially offer different views of the same entity. Adopting this unifying trinity interpretation, we expose these entities directly in our language. Importantly, the support is not limited to passive use in expressions but also allows new values to be created.

3.1 Using existing shapes

CGA++ provides multiple options for referencing an existing shape. In addition to querying the shape tree, shapes within a rule body can be conveniently accessed by name. Once a shape has been identified, it can participate freely in expressions; in particular, it may be used as argument to functions and operations. As a simple instructive example, the grammar in Fig. 2 demonstrates how the Boolean operation `minus`, which modifies the current shape by subtracting a given shape from it, can be employed to avoid overlapping building footprints.

By preceding an action in a rule body with a label (*label* `=` *action*), such as `m` in the example, the result of this action is assigned this label and can be accessed directly by using the label (it basically becomes a local variable name). Note that it is hence also possible to refer to an intermediate result that would generally not be included in the shape tree, such as the outcome of shape-modifying operations like `t` or `i`. If labeled, such a result gets actually integrated into the shape tree to allow a meaningful interpretation as node and subtree; however, care must be taken that it is excluded from the final procedurally generated model. Details on how this can be realized are provided in the supplemental material. Furthermore, the keyword `this` offers direct access to the current shape.

With shapes being usable as values, operations become possible that accept further shapes as arguments, enabling Boolean operations, such as `minus`, among others. Analogously, functions can be provided that query properties of one or more shapes, such as examining the spatial relationship of two shapes. For instance, `overlaps(`$shape_1$`,` $shape_2$`)` determines whether two shapes overlap.

Querying the shape tree CGA++ offers functions for arbitrarily navigating the shape tree (cf. Fig. 3). Given a node, both its parent and a list of its children can be queried. Related convenience functions exist that return all leaf nodes for the according subtree or a list of all nodes enumerated according to a specified traversal order.

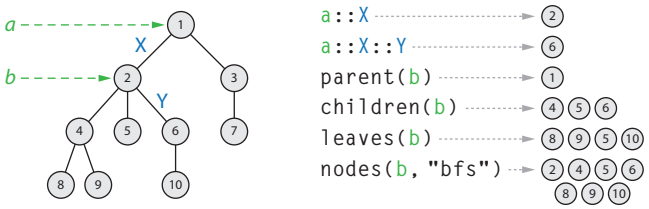


Figure 3: *Example illustrating some of the offered tree navigation options.* `a` *and* `b` *are values representing the shape nodes 1 and 2, respectively, whereas* `X` *and* `Y` *are labels.*

The absence of a node (e.g., the parent of the shape tree's root) is expressed with the null node literal, denoted by `null` in grammars. To reference a labeled child, the (left-associative) access operator `::` can be used, which takes a node on its left side and a label on its right side.

When consulting the shape tree, it is often easiest to locate shapes based on some criterion, not least because this allows abstracting from the exact structure of the tree. One powerful device in this regard are attributes. In CGA++, each shape can have an arbitrary number of attributes, which can be both set and retrieved. An attribute is identified by a name, and its value can be of any type supported by the language, including a shape. The attribute system is also used by several operations, which automatically set some predefined attributes. Both the presence of an attribute and its value can be utilized to identify shapes, and according convenience functions are provided.

3.2 Creating new shapes

The real power of introducing shapes as first-class citizens into grammars unfolds only when going beyond referencing and querying existing shapes, and allowing creating new shapes.

Constructing new shape trees Spawning a new derivation process from within the current one, the construct `<` *actions* `>(`*base*`)` creates a new (sub) shape tree. Employing the shape identified by *base* as initial shape, the specified actions are executed and the resulting tree is returned. If the *base* argument is omitted, the current shape is used.

Such a new shape tree can be used in expressions just like any other shape, enabling executing numerous advanced modeling tasks. Potential applications include deriving a (temporary) variant of a shape (e.g., enlarging it) for use with a spatial query function such as `overlaps`. As another example, multiple such shape trees can be created to cover a range of alternatives (e.g., different façade decompositions) and determine which one to choose.

Crucially, it is possible to incorporate a given shape tree into the current derivation. The operation `include(`*tree*`)` embeds the given tree as a sibling of the current shape, whereas the operation `adopt(`*shape*`)` modifies the current shape such that it matches the specified one.

Functions A new shape tree is also produced by several built-in functions, deriving it from one or more input (sub)trees. One set of these functions is primarily concerned with modifying shapes. For instance, `transformScope(`*source*`,` *target*`)` returns a copy of *source* where the scopes of all nodes are subjected to the transformation that makes the root node's scope identical to the scope of *target*, thus essentially fitting the source into the target. For many operations, according functions are provided that take the shape to work on as argument and return the result as a new shape; whenever reasonable, they further extend an operation's effect to a whole

tree. As an example, `t(`*tree*`,` Δx`,` Δy`,` Δz`)` translates the scopes of all nodes of the specified tree by the given offset. Corresponding functions also exist for subdivision operations, each returning a list featuring all resulting parts.

Another set of functions focuses on manipulating the structure of trees. In addition to elementary functions for creating a new tree with given children and for adding, removing, or replacing subtrees of a given tree, high-level tree rewriting functions are offered. These allow pruning subtrees as well as refining leaf nodes by invoking a provided rule for them.

Resumable shapes Refining a tree later on is further facilitated by *resumable shapes*. These serve as indicator nodes for where the derivation should be continued in later stages. Such a node is created with the rule-body action `?`*name*`(`arg_0`,...)`, recording the current shape and the provided arguments; it is basically an ordinary shape node with special attributes. The corresponding function `continue(`*tree*`,` $name_0$`=`$rule_0$`,...)` invokes the rule $rule_i$ for all resumable shapes in the given tree whose name matches $name_i$, using the arguments associated with the respective resumable shape, and returns an accordingly refined tree.

Fig. 4 shows an example where two partial shape trees are constructed and the one crystallizing as the better option is then completed by refining the resumable shapes of the according tree. The `Parcel` rule explores two alternative development schemes by first partially constructing the two corresponding shape trees; they are assigned to variables `a` and `b` for subsequent use. The involved rules `DesignA` and `DesignB` ultimately generate `Footprint` shapes and extrude them, yielding building masses; their further refinement is deferred by emitting a resumable shape `?Mass` for each. Employing the function `V`, which sums the volumes of the leaf shapes of a given tree, the `Parcel` rule then determines the tree with the larger total mass volume (`a` in the shown example case) and continues its refinement at its resumable shapes using the according rule (`Mass1` or `Mass2`, respectively); finally, the resulting shape tree is embedded using `include`.

4 Multi-shape coordination with events

The powerful ability to access other shapes during the derivation (e.g., to establish contextual information) necessitates that these other shapes already exist. To ensure their existence, often certain control of the derivation order is needed. In simple cases, a signaling mechanism could be used, where a signal is fired after creating the required shape, and the rule wanting to access that shape first waits for this signal. More generally, situations can arise where only once a whole set of shapes has been created, a decision on how to proceed evolving each of them can be made, utilizing information from all these shapes. As a simple example, when union-ing multiple shapes, the largest one may be selected and then be modified by a Boolean union operation with all the remaining shapes, whereas these other shapes are discarded, replacing each with the empty shape (`NIL`). Note that such scenarios require (a) a means for identifying the set of involved shapes and (b) a facility to coordinate the individual refinement decisions.

In CGA++, we address all these demands by introducing *events*. They serve as synchronization points, offer influence on the order in which the derivation process refines shapes, and allow multiple independent branches of the derivation to exchange information and coordinate how to proceed. Mainly, two elements are involved in an event: the special `event(`*name*`)` operation and an event handler. The operation raises the specified event within a rule, which suspends the current branch of the derivation and identifies it as participant in the event. Once all active branches have raised some event, the set of participants is completely known, all of them effectively having reached a common synchronization point, and the event handler is consulted. It can access the current shapes of all the participating branches and makes a coordinated decision on how to continue in each branch; this is conveyed by yielding one rule for each participant. Subsequently, each branch is resumed, first executing the respective rule determined by the handler directly in-place; this basically corresponds to replacing the `event` operation with the rule's actions. Fig. 5 illustrates.

```
Parcel --> with(a = < DesignA >, b = < DesignB >) {
  case { V(a) > V(b): include(continue(a, Mass = %Mass1))
       | else:        include(continue(b, Mass = %Mass2)) } }

DesignA --> split("x") { 15: Footprint | ~20: NIL }*
DesignB --> setback(15) { all = Footprint }
Footprint --> extrude(rand(10, 30)) ?Mass

Mass1 --> ...
Mass2 --> ...

func V(tree) = sum(map(1 : leaves(tree), volume(1)))
```

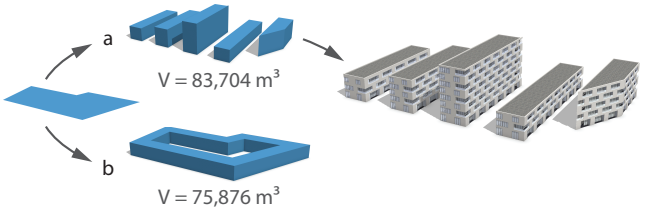


Figure 4: *Example demonstrating the construction of new trees and the use of resumable shapes for multi-stage refinement. It considers two different designs, chooses the one resulting in the larger total mass volume, and continues its derivation to develop the masses into detailed buildings.*

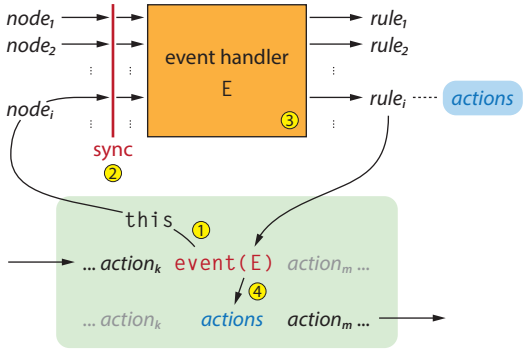


Figure 5: *An event is raised with the* `event` *operation (1) and synchronizes multiple branches of the derivation process (2). Taking the current shapes of all participating branches as input, the event's handler (3) returns a rule for each, specifying how to proceed. This rule's actions are then executed directly in-place (4).*

The event handler is specified as part of an event's definition; this happens directly within a grammar using the following construct:

`event` *name*`(`$param_0$`,...) {` *priority* `} =` *handler*

By providing parameters, a whole family of events can be defined, where each concrete parameter assignment constitutes a separate event (i.e., `e(0)` and `e(1)` are two different events). Among others, such parametric event definitions facilitate iterations with an arbitrary number of steps. Furthermore, each event is assigned a priority number, which may depend on the parameter values and is 0 if not specified, to guide the order in which multiple concurrent events are handled.

4.1 Event handling

An event handler ultimately determines one rule for each participant in the event, whose actions then get executed. The simplest example of such a handler is just `pass`, which yields an empty rule (with no actions) for all participants. It is employed if merely the synchronization aspect of an event is needed, which ensures that any shape that exists in a participating derivation branch before the respective `event` operation is present and thus accessible in all these branches right after the `event` operation. As another, more advanced example, the handler `foreach {` *actions* `}` yields a rule whose body is made up by the specified actions for each participant.

In general, an event handler can be an arbitrary expression that evaluates to a list of rules (one rule for each participant); it may hence effectively realize an arbitrarily complex, coordinated decision procedure for how to proceed with all the participants. These are provided as a list of their respective current shapes via the special variable `$nodes` (such implicitly available variables are always `$`-prefixed). Actually, `pass` and `foreach` are just two examples of convenience functions offering a concise notation for common application cases. Both take a list of shapes as argument, which in this context defaults to `$nodes` if omitted, and return a list of rules. In case of `foreach`, this argument is even iterable (cf. also Sec. 5), which means that *actions* is evaluated for each shape separately and can access both this respective shape and its list index via special variables.

Illustrative example Fig. 6 shows an example where at first a parcel is subdivided stochastically by recursively splitting off footprints. Our goal is to build an office tower on the largest of them and apartments on the others. To identify the largest one, we employ the event `IdentifyLargest` in the `Footprint` rule; raising it initially suspends the further derivation for the respective footprint shape. Once all footprints have been generated and entered the event, its event handler is evaluated. This handler takes the list of all footprint shapes (`$nodes`) to derive a list of their areas (`A`), determines the index of the largest one, and then, using the `foreach` function, determines a rule setting the Boolean attribute `isLargest` appropriately for each shape. Subsequently, the `Footprint` rule is continued, executing the determined rule in-place, thus setting the attribute. Its value is queried later in the `Mass` rule to choose the correct refinement strategy. Note that the goal in this simple example constitutes one of the typical tasks that cannot be solved with traditional grammar-based systems. These would require determining the largest footprint beforehand, which is rarely practical because of the unknown geometric form of the parcel and the presence of stochastic elements.

Rule values The flexible use of rules surfacing in the exposition so far is enabled by granting them first-class citizenship in CGA++, allowing them as values in expressions as well as their on-the-fly instantiation. Any rule defined in a grammar can be referenced by prefixing its name with `%` (e.g., `%A`). If the rule has parameters, it is also possible to obtain a reference to it that comes with an associated argument list (e.g., `%A(20,14)`); the returned rule value captures the argument values and is parameterless. Furthermore, anonymous rules are supported: the construct `%(`$param_0$`,...)<` *body* `>` returns a new rule, with the values of all variables from outside the rule that are referenced by the actions in *body* being captured. That is, the value of such a variable at the time that the rule is created by the construct is stored as part of the rule value. Such capturing of input values is extremely powerful, allowing individualized rules and transferring information. In particular, it is fundamental for the support of general event handlers and hence used extensively (among others, by the `foreach` function, which returns a list of anonymous rules `%<` *actions* `>`). A given

```
Parcel --> split("x") { rand(8, 16): Footprint | ~1: Parcel }
Footprint --> event(IdentifyLargest) extrude(area()/6) Mass
Mass --> case { get("isLargest"): Offices | else: Apartments }
Offices --> ...
Apartments --> ...

event IdentifyLargest =
  with(A = map(n:$nodes, area(n)), largest = index(A, max(A)),
    foreach($nodes) { set("isLargest", $index == largest) } )
```

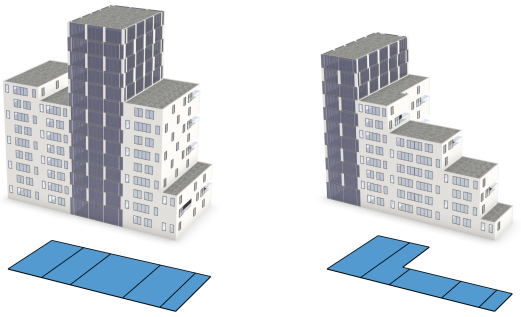


Figure 6: *Example showing a simple common use case for events. The event* `IdentifyLargest` *is used to identify the largest footprint resulting from randomly subdividing a parcel; the corresponding mass is then developed into an office building, whereas apartments are erected on all the other footprints.*

rule can be invoked or executed in-place within a rule body via the operations `invoke` and `apply`, respectively. Note that the latter one enables rules to be utilized as a kind of macro for sequences of actions that are used at multiple occasions in a grammar; for example:

```
FlattenY --> s('1, 0, '1) set("scope.ty", 0)
InitialShape --> apply(%FlattenY) ...
```

Hierarchical handling There are situations where, once the set of event participants has been established, a division of these into (disjoint) groups emerges and further coordination is only necessary (and most conveniently done) locally within each individual group. For instance, such groups may be induced by spatial relationships between shapes, such as (transitive) adjacency or overlap. Handling such cases is facilitated by dedicated partitioning functions. These offer various ways of partitioning a list of shapes according to some criterion. For each resulting group, a given event handler expression can then be evaluated separately just for this group. Finally, the partitioning is reversed by merging the lists of rules yielded by the handlers for all groups accordingly, thus producing a single list of rules for all event participants. As a concrete example (cf. Fig. 7), consider the task of resolving overlaps among shapes before continuing their further development. Once a shape has been placed, an event can be raised, whose handler breaks the set of all participating shapes into subsets of connected shapes and pursues an overlap resolution strategy for each subset:

```
event ResolveOverlaps =
  partitionByPred($nodes, overlaps($a, $b),
                  subtractLargerOnes($groupNodes))
```
(S1)

`partitionByPred` partitions the list of shapes (`$nodes`) such that any two elements (`$a`, `$b`) for which the predicate (`overlaps`) evaluates to true are put into the same group. For each resulting group of transitively overlapping shapes (`$groupNodes`), the handler `subtractLargerOnes` (defined in the supplemental material) is evaluated; it yields rules that subtract from each shape all overlapping shapes with a larger area.

Coordination support Coordinating the future of multiple shapes may involve abandoning some of them (e.g., because they

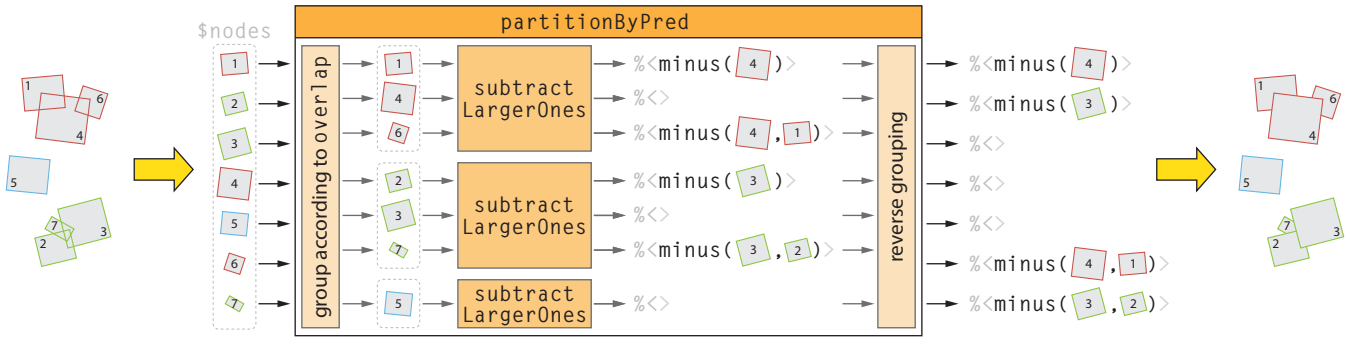


Figure 7: *Illustration of the event handler in snippet S1 on a simple example of overlapping shapes. First, a dynamic grouping of all participating shapes is performed, identifying subsets of transitively overlapping shapes. For each group, a handler implementing the overlap resolution strategy (`subtractLargerOnes`) is consulted. It returns the according actions as rule values. These are finally collected into a joint list for all groups, whose order matches the one of the event's input list of shapes.*

were union-ed into another shape); this is simplified by two new operations: `stop` aborts the execution of the current rule body, causing all remaining actions to be ignored. `kill` additionally removes all shapes resulting from the current rule body from the final result (by adding an empty shape as successor to each according leaf node). In the case that coordination requires further participant-specific values, attributes of the respective current shape can be employed: before raising the event, the necessary input parameters are simply set as attribute(s); analogously, attribute-setting operations can appear in the rule returned by the handler to communicate specific output quantities (as in the grammar in Fig. 6).

4.2 Advanced scheduling

The capabilities and the expressiveness of events are enhanced with event groups and the ability to wait for events; these also influence the order in which events are handled and actions across different branches of the derivation process are executed.

Event groups To accommodate the frequent scenario where different subtrees of the evolving shape tree should be coordinated analogously but independently from each other, the concept of *event groups* exists. They provide an easy means to confine an event to a certain subtree, while allowing multiple such subtree-specific instances of the event at a time. For example, when using events to coordinate across the floors of a building, event groups allow that a separate, local instance of an event is raised for each building, in which only the floors of this single building participate (instead of the floors of all buildings). Concretely, a group node identifies a subtree and is created with the operation `group(`*name*`)`, which further causes all shapes created by succeeding actions to become descendants of the group node. By specifying the group name as an additional argument in the `event` operation, the event becomes local to the subtree rooted in the closest group node ancestor with a matching name (otherwise, the shape tree's root is taken). Each resulting *event instance*, defined by an event and a subtree, is then handled separately. An illustrating example is provided in Fig. 8.

Note that event instances explicitly cover a special case of hierarchical handling, where the subset each participant belongs to is directly specified at the time of raising the event (via the respective group node). Event groups effectively provide a scoping mechanism, and this interpretation is also reflected in which order event instances are handled.

Scheduling Before detailing how events interact with the derivation process, let us first clarify the notion of derivation branches. During the derivation process, the execution of actions within a rule body can introduce new shapes. Their respective refinement constitutes a new branch of the derivation; these can be carried out independently from each other as well as any further existing branch, including the one that previously introduced the shape to be refined. A symbol, for instance, instantiates the current shape, and because any subsequent refinement of this new shape (specified by the rule corresponding to the symbol) has no effect on the state of the current shape, this refinement can be performed independently from the actions following the symbol (determining the further refinement of the current shape).

Within each branch, actions are executed sequentially until an event is raised, causing the branch to be suspended. Once this happened to all active branches, the highest-priority event instance pending is determined. The according event handler is evaluated, and the branches participating in the event instance are resumed, initially injecting the actions of the respective rule returned by the handler. Note that any further pending event instance is only handled once all branches are suspended again (and thus these branches' participation in the event is known). When determining the highest-priority event instance, dependencies among the roots of the instances' subtrees are regarded: we exclude all instances whose root has any other instance's root as a descendant from the selection. This is motivated by considering each instance's subtree as defining a scope and causes any nested scope to be handled before its enclosing one.

Signaling Events can be used to signal the availability of a certain set of shapes. The `wait` operation allows a branch of the derivation to wait for such a signal; it suspends the branch until a specified event got handled within a certain subtree. To this end, we keep a list of pending and processed events for the root of each subtree encountered in an event instance. In case all derivation branches are waiting for events, we report a warning and resume the one waiting for the highest-priority event instance.

5 CGA++ grammar language

Exposing shapes as first-class citizens and introducing events as a flexible and powerful means of coordination are cornerstones in enhancing the capabilities of existing grammar languages such as CGA shape and enable successfully addressing the limitations outlined in the Introduction. These novel language elements of CGA++ are complemented by several further new language features, which offer simpler, clearer, and more concise forms of notation and enhance the expressiveness. Some of them are also instrumental ingredients for fully realizing the concepts of shape values and events (e.g., lists).

Supported objects In addition to Booleans, numbers, and strings, CGA++ supports lists and tuples. Lists are sequences of an arbitrary number of values of identical type, whereas tuples represent fixed-size sequences of values of possibly different types. Furthermore, shapes and rules but also functions are available as first-class citizens and hence can be used as values in expressions. Both built-in and user-defined functions can be referenced by their respective name. Moreover, CGA++ supports anonymous functions: the construct `[`*param*$_0$`,...](`*body*`)` yields a new function value, where any variable from outside the function referenced in the expression *body* has its value captured as part of the function value (thus forming a closure).

Expression arguments One example of a new language feature that aims at ease of expression and providing a succinct syntax is *expression arguments*. They are supported by many built-in functions processing multiple items (such as the elements of a list) and allow specifying expressions that are to be evaluated by a function (e.g., for each item) directly as arguments to this function. Within such an expression, evaluation-specific values (such as the current item) are exposed via implicit variables. For instance, in snippet S1, `overlaps($a, $b)` constitutes an expression argument that is evaluated by `partitionByPred` for each pair of shapes, where a pair's elements are accessible via `$a` and `$b`. Note that in case of the convenience function `foreach` (cf. Sec. 4.1), the specified *actions* also act as an expression argument.

Iterable arguments If a function takes a list as one argument and evaluates an expression argument for each element of this list, then a semantically meaningful name for accessing the respective element within the expression argument can be specified as part of the list argument (*name* `:` *list*). (If omitted, the element is available via an implicit variable.) Moreover, the element's index is exposed as an implicit variable.

Chain operator When iteratively applying a sequence of functions to a value, the chain operator `->` can be useful for improving legibility; it applies its left-hand side as first argument to its right-hand side. For instance,

```
this -> t(2, 0, 2) -> s(2, 9, 6) -> r(30, 0, 0)
```

is equivalent to but easier to parse (for many humans) than

```
r(s(t(this, 2, 0, 2), 2, 9, 6), 30, 0, 0).
```

Implicit argument In some contexts, there exists a natural, well-defined value for which an occurring function is generally applied. Supporting notational conciseness, we hence often allow simply omitting this value. For example, if a built-in function appears within a rule body and accepts a shape as first argument, the current shape (`this`) is automatically used if this argument is omitted. Similarly, the list of participating shapes (`$nodes`) is used implicitly as first argument in an event's handler expression.

Auxiliary variables In numerous situations, like when dealing with complex expressions or using the result of an expression multiple times, clarity and ease of formulation can benefit from introducing variables for the values of some involved expressions. To this end, the new construct

```
with(var1 = value1, ..., expression)
with(var1 = value1, ...) { actions }
```

is provided, which allows assigning values to variables and subsequently using them in an expression or within the arguments of actions, respectively (as done in the grammars in Figs. 4 and 6).

Enhanced operations Several operations offered by CGA shape [Esri 2015] have been enhanced in CGA++ to capitalize on its new features. For instance, some subdivision operations such as `comp` (component split) expect a list of *selector*`:`*actions* pairs, each identifying a specific set of parts and defining according actions for their further refinement. If an `=` separator is used instead of `:`, the parts are not refined individually but merged and processed as one single shape. In CGA shape, values for *selector* are restricted to some predefined selectors (such as `left`, `top`, `side`, or `all`), thus limiting the selection possibilities. By contrast, CGA++ allows an arbitrary predicate expression for *selector*, which can reference the respective part to check for a match (exposed as shape value) and related quantities, such as its index, via implicit variables. Previous selectors such as `left` and `top` have been turned into (locally available) built-in functions that take a shape as argument and classify it according to some properties, such as the direction of its surface normal. Once again aiming for notational conciseness, the part's shape generally does not have to be explicitly provided as argument for such selector functions. Concretely, if *selector* evaluates to a function, this function is applied implicitly, using the part's shape as argument if one is expected; this policy extends to the operands of logical operators (it is also often pursued for expression arguments). Consequently, complex selections such as `(left || top) && $index < 10` become possible and without compromising the succinctness.

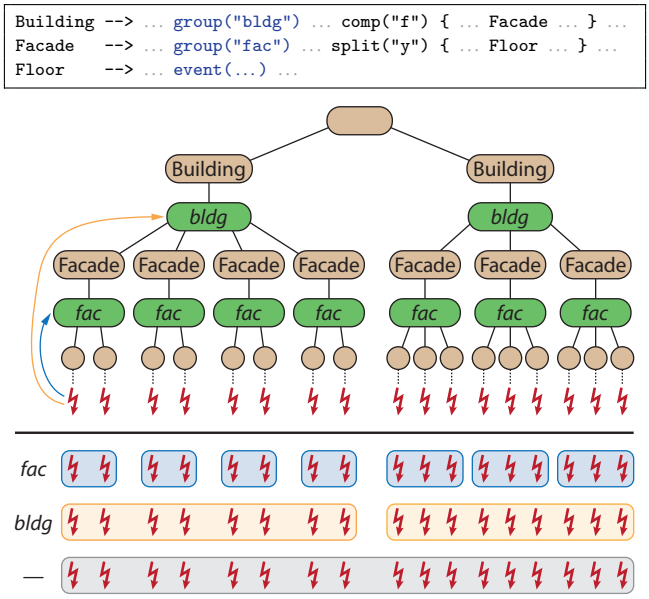


Figure 8: *Example demonstrating how event groups can be used to restrict an event instance to a subtree. Top: Excerpt from a grammar, in which groups are opened and events are raised. Middle: Resulting (simplified) shape tree, showing group nodes in green. Bottom: Depending on the group name specified when raising the event, different event instances result.* `event(E, "fac")` *causes only derivation branches refining floors of the same façade to participate in an event instance,* `event(E, "bldg")` *restricts an event instance to a single building, and* `event(E)` *includes all branches in the event.*

6 Applications

Demonstrating the new modeling capabilities offered by CGA++, we present three concrete examples in the following. They cover different application domains and showcase modeling problems whose solution eludes previous grammar-based procedural systems.

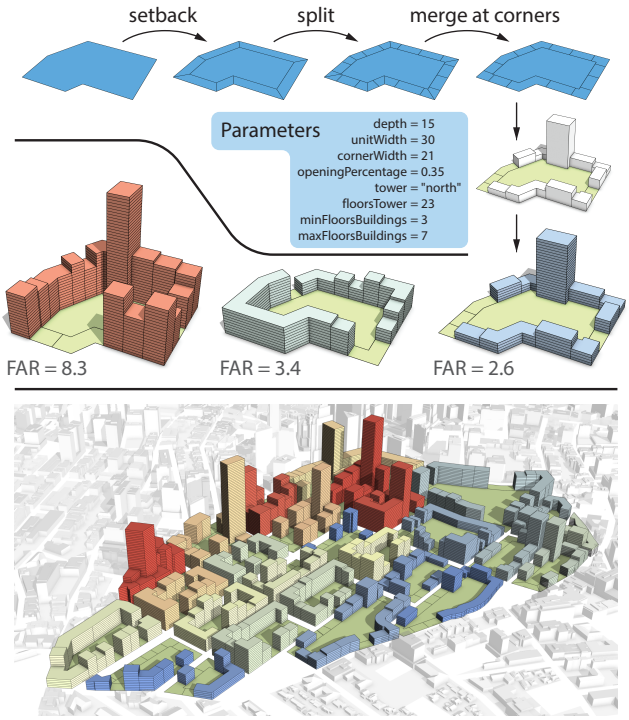


Figure 9: *Perimeter block design. Top: Pursued modeling strategy and involved design parameters. Middle: Example results for different parameter choices. Bottom: Example of a real-world urban planning result.*

6.1 Urban planning

Our first example originates from interaction with urban planners and incorporates many of their day-to-day requirements. It shows that previously impossible parametric urban design rules can easily be encoded with CGA++, demonstrating that our language is well suited for urban planning applications.

One typical task for urban planners is designing the so-called perimeter block, a block parcel with buildings on its boundary. Fig. 9 shows the involved parameters that urban planners work with and presents example results obtained with the following grammar:

```
Parcel --> set("parcelArea", area())
  setback(depth) { all: Footprints | remainder: GreenSpace }
Footprints -->
  split("x") { ~cornerWidth: event(MergeTouching) Footprint
             | { ~unitWidth: event(SelectTower)   Footprint }*
             | ~cornerWidth: event(MergeTouching) Footprint }
Footprint -->
  case { get("isTower"): s('1, '1, '1.5) ... Tower
       | else: event(SetToGreenSpace) ... Building }
...
Floor --> event(CalcFAR)
  set("material.color.rgb", ramp(get("FAR")/10)) ...
```

As illustrated in the top of Fig. 9, the **`setback`** and **`split`** operations first refine the parcel. The event **`MergeTouching`** is used to perform a Boolean union operation on the corner footprints:

```
event MergeTouching =
  partitionByPred($nodes, touches($a, $b), U($groupNodes))

func U(shapes) = foreach(shapes) {
  case { $index == 0: union(shapes) | else: kill } }
```

First, (transitively) touching footprints are grouped together; the shapes in each group are then union-ed into the first shape of the group. Note that with **`select`** and **`forall`** alternatives for **`foreach`** exist that would yield an even more compact syntax in this case.

The event **`SelectTower`** is analogous to **`IdentifyLargest`** in Fig. 6 but uses a different selection criterion. Instead of determining the footprint with the largest area, it flags the one positioned most in the direction given by the design parameter **`tower`** (e.g., **`"north"`**).

With the event **`SetToGreenSpace`**, the common rule authoring challenge of doing something only on a specific fraction or number of shapes is tackled. In traditional systems such as CGA shape, the author may only pursue a stochastic approach, i.e., invoke the according rule with a probability matching the envisaged fraction, and hope that chance yields the desired result. By contrast, CGA++ allows selecting a deterministic number of shapes:

```
event SetToGreenSpace = with(
  k = rint(size($nodes) * openingPercentage),
  r = map($nodes, case { $index < k: %< GreenSpace stop >
                       | else:       %< > } ),
  shuffle(r))
```

After deriving the number **`k`** of footprints that should be turned into green space, a list of rules is compiled, comprising exactly **`k`** times the rule for green space development and the empty rule **`%<>`** for the remaining footprints, and then randomly permuted. As a result, when resuming the **`Footprint`** rule, the **`GreenSpace`** rule is invoked for **`k`** random footprints; the accompanying **`stop`** operation causes the rest of the **`Footprint`** rule to be skipped in those cases.

Finally, **`CalcFAR`** showcases a further typical application of events: analyzing the geometry as a whole after generation.

```
event CalcFAR = with(gfa = sum(map(n : $nodes, areaTop(n))),
  foreach($nodes) { set("FAR", gfa / get("parcelArea")) } )
```

The gross floor area (i.e., the total area on all floors) is computed, and the value of the resulting floor area ratio (FAR), which is an important density indicator in urban planning, is set as an attribute. Its value can then be used later on, e.g., to colorize the model.

6.2 Buildings

The encoding of procedural building designs often poses several challenges. In this example, we present potential solutions to a few commonly encountered ones, enabled by CGA++'s new features.

Inspired by plans for ecological skyscrapers in Asian mega cities, offering plenty of green space for sustainable living, our example design consists of (at most) two tower blocks, featuring shifted floors and terraces hosting roof gardens, that are linked by a series of skybridges, as shown in Fig. 10. The main design parameters are the gross floor area (**`GFA`**) and the maximum building height allowed (**`maxH`**); these are typically given by the investors and the authorities. One approach to meet these requirements is demonstrated by the following grammar:

```
Parcel -->
  split("x") { ~1: BldArea | ~1: GreenSpace | ~1: BldArea }
BldArea --> event(DistributeGFA)
  extrude(get("nFloors") * floorH) Tower

event DistributeGFA = with(
  A = map(s : $nodes, area(s)),
  n1 = numFloors(A[0], GFA),
  n2 = numFloors(A[1], GFA - A[0] * n1),
  map(n : list(n1, n2), %< set("nFloors", n) >))

func numFloors(shapeArea, targetFloorArea) =
  min(ceil(targetFloorArea / shapeArea),
      floor(maxH / floorH))
```

Given a parcel, at first two buildable areas (**`BldArea`**) for the towers are created. Subsequently, the GFA needs to be distributed among

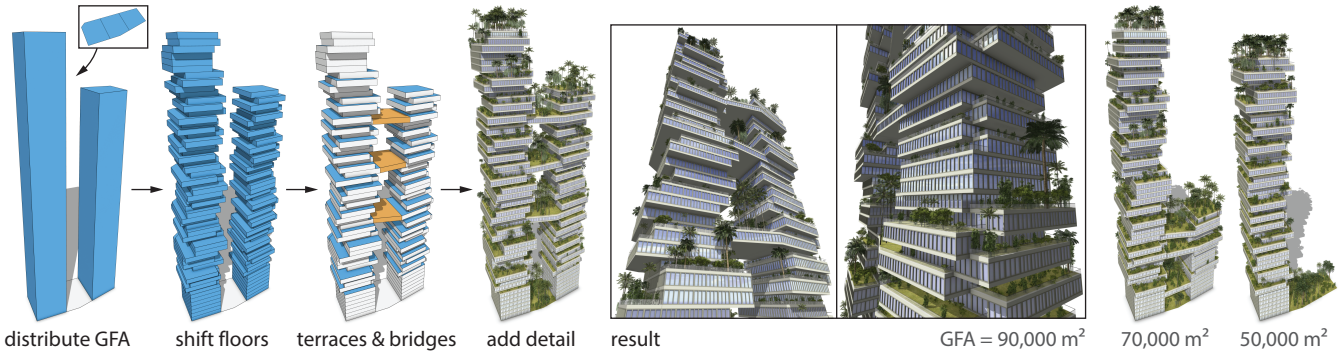


Figure 10: *Building design with (at most) two interconnected towers and shifted floors that provides a given GFA (gross floor area). Left: Major steps of the modeling approach. Middle: Close-up views of the example result. Right: Results for different GFA values.*

them. To this end, we want to assign as much as possible of the floor area to the first tower without exceeding the maximum building height and make the second tower only as high as needed to provide the remaining floor area (we use `maxH` = 230 m and a floor height `floorH` = 4 m for all results shown).

This is one specific instance of the frequent problem that multiple building parts have to jointly fulfil an overall requirement. As the sizes of buildable areas are only known after these have been created, each area does not know about the size of the others without some form of communication. While an according device is missing in traditional grammar-based systems, making them severely struggle with such tasks, CGA++ offers events as a convenient solution. Concretely, we employ the event `DistributeGFA` to access the areas of all `BldArea` shapes and then compute the number of floors for each tower, conveying it via the attribute `nFloors`.

The resulting building mass for each tower is then refined by splitting it into floors and stochastically shifting them horizontally:

```
Tower --> group("tower")
  split("y") { floorH: set("floorIdx", get("split.index")) ...
                       t(randShift(), 0, randShift()) Floor }*
Floor -->
  event(CreateTerraces, "tower") event(CreateBridges)
  comp("f") { bottom: FloorPlane | side: Facade }

func randShift = prob { 0.35: 4 | 0.35: -4 | else: 0 }
```

To create the terraces, we have to determine those parts of the floors' top surfaces that are not occluded by the bottom surface of the respective next floor, and we again employ an event for this. As this terrace extraction is purely local to each tower, we define an event group "`tower`" in the `Tower` rule that encompasses all floors of one tower. Hence, a distinct event instance is raised for each tower in the `Floor` rule, allowing us to handle each individually:

```
event CreateTerraces = with(
  f = [s](get(s, "floorIdx")),
  byFloor = append(sort($nodes, f($a) < f($b)), null),
  above = map(n : $nodes, byFloor[index(byFloor, n) + 1]),
  foreach($nodes) {
    comp("f") { top: minus(above[$index]) Terrace } } )
```

First, the floors are sorted in ascending order by their index (which was assigned when splitting the tower mass). For convenience, we employ a local auxiliary function `f`, which is defined using CGA++'s anonymous function facility. Subsequently, for each floor the floor above can easily be determined by consulting the sorted list. This floor above is subtracted from the top surface of the floor to yield the terrace area. Note that this modeling task is an example of the common problem of determining the exact residual of a surface after the stochastic placement of elements on top of it.

The skybridges connecting some of the floors of the two towers are also created via an event:

```
event CreateBridges =
  partitionByNumber(n : $nodes, get(n, "floorIdx"),
                    createBridge($groupNodes))

func createBridge(floors) = case {
  size(floors) != 2 || p(0.9): pass(floors)
| get(floors[0], "scope.tx") < get(floors[1], "scope.tx"):
        list(%< BridgeTo(floors[1]) >, %<>)
| else: list(%<>, %< BridgeTo(floors[0]) >) }

BridgeTo(target) --> with(
  targetFace = comp(target, "f") { left } -> first -> first)
{ comp("f") { right: connectTo(targetFace) Bridge } }
```

First, the floors of both towers are grouped by their floor index via `partitionByNumber`, which functions analogously to `partitionByPred` but establishes groups by identical number values instead of by a binary predicate. As we aim at connecting only a few randomly selected floors, we do nothing with a probability of 90% or if the floor exists in only one tower. In the remaining cases, we select the floor of the left tower and connect its right face to the left face of the other floor from the right tower via the `connectTo` operation, which creates a connecting tube. Note that for determining the target face from the right tower, the function equivalent of the component split operation is used.

Finally, the façades, terraces, and skybridges are refined to add procedural detail, including placing vegetation models.

6.3 Façades

Alignment plays a central role in many real-world applications, but complex cases can often not be realized with existing grammar-based systems, especially if random elements are involved. CGA++ introduces the means to successfully deal with such problems, which we demonstrate with an example of generating random front façades.

Let us assume the following design objectives: (a) all floors should feature randomly placed elements (e.g., windows); (b) all upper floors should look identical; (c) the first floor should have one door that is center-aligned with an element of the upper floors or the empty space between two such elements; (d) the other elements in the first floor should align on both their left and their right with the left or right of an upper-floor element; (e) elements (and the façade boundaries) should be separated by empty wall space whose width lies within a certain interval.

The following grammar, whose high-level approach is illustrated in Fig. 11, offers one potential solution for implementing this design with CGA++ (see the supplemental material for omitted details):

```
FrontFacade --> with(
  emptyFs  = split(this, "y") { ~2.5 | { ~2.3 }* },
  topF     = refine(last(emptyFs), %FillUpperFloor),
  middleFs = map(f : sublist(emptyFs, 1, size(emptyFs) - 2),
                 transformScope(topF, f)),
  firstF   = createFirstFloor(first(emptyFs), topF))
{ include(firstF) include(middleFs) include(topF) }
```

We first split the façade into multiple floors with the `split` function, which yields a list of resulting shapes. The one corresponding to the top floor is then refined with the rule `FillUpperFloor`. Afterward, this resulting shape (sub)tree is copied to the middle floors via `transformScope`, which adjusts the scope to match the respective middle floor's one. Finally, the first floor is created with `createFirstFloor`, before the trees representing the floors are embedded via `include`.

Top floor The elements that may be placed in an upper floor are encoded as a list, which provides for each element an according rule and its permissible minimum and maximum width:

```
const upperFloorElems = list(
  tuple(%WindowCell(%SingleWindow), 0.6, 1.0),
  tuple(%WindowCell(%DoubleWindow), 1.0, 1.4))
```

The rule `FillUpperFloor` first determines the available space and then derives those elements from the list fitting into this space:

```
FillUpperFloor --> with(
  availSpace = get("scope.sx") - 2 * minWallW,
  availElems = filter(e : upperFloorElems, e[1] <= availSpace))
{ case { size(availElems) > 0: with(
    elem = randomElem(availElems),
    w = rand(elem[1], min(elem[2], availSpace)),
    wallW = minWallW
          + rand(0, min(maxWallW - minWallW, availSpace - w)))
    { split("x") { wallW: Wall
                 |  w: set("floorCell", true) invoke(elem[0])
                 | ~1: FillUpperFloor } }
  | else: Wall } }
```

If none fits, a wall piece is created; otherwise, we stochastically choose an element and widths, satisfying the design's constraints, for this element and the wall to the left of it. The wall and the element (marked as a `floorCell`) are then split off, and the remaining part of the floor is filled recursively.

First floor The construction of the first floor showcases context-sensitive splits with non-trivial alignments:

```
func createFirstFloor(f, ref) = with(
  cells = getCells(ref),
  candidatePlacements = legalDoorPlacements(f, cells),
  placement = randomElem(candidatePlacements),
  doorW = rand(minDoorW, placement[1]),
  refine(f, %< split("x") {
      placement[0] - doorW/2: FillFirstFloor(cells)
    | doorW:                  DoorCell
    | ~1:                     FillFirstFloor(cells) } >))
```

At first, the elements from a reference floor (`ref`) marked as `floorCell` are determined by `getCells`, which traverses that floor's shape tree. From these, all places (encoded by the center position) where a door can be put satisfying all constraints (such as the center alignment), along with the according maximum door widths are derived by `legalDoorPlacements`. One of these candidate placements and a corresponding door width are then stochastically selected, and an according split is performed. The remaining parts left and right to the door are refined by the rule `FillFirstFloor`, which randomly places elements in a greedy manner such that vertical alignment across floors as required by objective (d) is achieved.

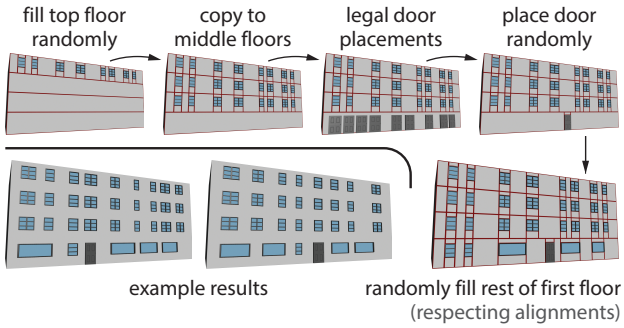


Figure 11: *Façade design with random elements and alignments. Top: General solution approach (split lines are shown in red). Bottom: Example results for different random seeds.*

7 Discussion

In the following, we discuss design choices made, briefly explore potential alternative solutions, and relate our solution to prior work. Furthermore, we identify limitations and open challenges.

7.1 Implementation

We implemented a prototype of CGA++ that supports all features presented in this paper and offers a large subset of the operations available in the latest dialect of CGA shape [Esri 2015] (including all those outlined in the reference in the supplemental material). Our application takes a grammar as input and compiles it to an intermediate form, performing limited static type checking as well as a few optimizations (e.g., constant forwarding). Given an initial shape (typically a polygon representing a lot), the compiled grammar is then executed, returning the leaf shapes constituting the final model.

7.2 New language features

Two important aspects of any language are (a) what can be expressed, realized, and achieved with it, and (b) how easy this is. Both first-class citizenship of shapes and the concept of events primarily target the first aspect, without neglecting the second one. They significantly advance the capabilities of previous grammar languages and offer a seemingly simple but extremely powerful and (as we believe) elegant solution for addressing many existing fundamental limitations of what can be modeled with such languages. By contrast, many other new language elements are mainly motivated by the important quest for ease of expression. Therefore, some of them are not necessary from a purely technical point of view, even including the convenient first-class citizenship of rules and functions (see the supplemental material for details).

One guiding objective when devising CGA++ was to offer a consistent, concise, readable, and meaningful syntax for the new features without compromising generality and expressiveness or breaking any established language concepts. For instance, convenience constructs for event handling, such as `foreach` or the more versatile `select`, cover many common use cases and are designed to offer a compact syntax analogous to that of subdivision operations, where actions can be directly specified (without having to explicitly wrap them into an anonymous rule). By exposing these constructs as functions and not restricting event handlers to those preconceived constructs (and cases) but allowing arbitrary expressions yielding a list of rules instead, arbitrary custom solutions are possible when desired and can even readily employ the convenience functions as building blocks. Such a custom solution may also be encapsulated in a function and then used as building block itself.

In the design of CGA++, we partially drew inspiration from the plethora of existing programming languages, in particular functional languages (such as Scheme), recent functional extensions to mainstream languages (such as in C++11), and scripting languages with a focus on (and notoriety for) conciseness (such as Perl).

7.3 Solution approaches

In general, various approaches are conceivable to cope with advanced modeling scenarios and strategies that are precluded in current grammar languages due to the limitations described in the Introduction. With CGA++, we are the first to offer an integrated solution within a grammar language. Carefully evolving an established language and introducing novel language elements, CGA++ hence allows mastering such modeling challenges without having to leave the familiar realm of a grammar language or even abandon the advantages of grammars, such as their notational effectiveness.

Alternatively, one may switch to a (suitable) more general (non-grammar) language to perform procedural modeling; potential candidates include scripting languages in modeling software or languages for generative modeling, such as GML [Havemann 2005]. However, as detailed in the supplemental material, such a solution entails some practical challenges and inconveniences. At least in some cases, one further option can be resorting to external tools, such as to a standard modeling software for performing Boolean operations. Among others, these are often hard to integrate in the modeling workflow, easily causing some disruption, and may require significant manual effort, harming scalability, though.

7.4 Related variants and extensions of CGA shape

A significant body of prior work deals with grammar-based procedural modeling, and CGA++ subsumes and extends multiple capabilities and features of them. Krecklau et al. [2010] suggest a more object-oriented variant of CGA shape, called G^2. It associates each shape with a certain class, where a separate set of operations is provided for each class. A simplified form of CGA shape is captured by one class, while trilinear freeform deformation cages are newly offered by another class. Generalizations such as this or support for convex polyhedral scopes [Thaller et al. 2013] may be orthogonal to CGA++'s focus but remain applicable to CGA++ as well. Furthermore, G^2 offers several enhancements that simplify grammar writing, including shape-local Boolean flags and the ability to pass rule names as arguments. The latter (which is actually necessitated by G^2's syntax, where subdivision operations expect part-refining rules as arguments) constitutes a first step toward the comprehensive first-class treatment of rules in CGA++, whereas flags are subsumed by CGA++'s support for arbitrary user-defined attributes.

To allow for local modifications in their visual editor, Lipp et al. [2008] introduce semantic tags and instance locators as means to identify subsets of nodes. Such tags effectively attach a name to the root node of a subdivision operation's result, while a locator identifies a particular node in the shape tree via a sequence of child indices. Both concepts are external to and not exposed in the edited grammar itself. By contrast, CGA++ can easily realize semantic tags via attributes and represent locators; in particular, these can be fully used and acted on in the grammar.

Patow [2012] proposes a system based on a graph-based interpretation of a grammar's set of rules. In follow-up work [Barroso et al. 2013], complex graph-rewriting is further explored, which could be adapted to CGA++. Finally, several language extensions exist that primarily target specific applications such as creating interconnected structures [Krecklau and Kobbelt 2011] or assisting lighting design [Schwarz and Wonka 2014]. In principle, according domain-specific operations could also be added to CGA++.

7.5 Context sensitivity in grammar systems

Taking context into account is often essential in realizing complex and varied results. In grammar systems for procedural content creation, the degree and the form of according support vary. For instance, in the case of L-systems [Prusinkiewicz and Lindenmayer 1990], the directly adjacent textual context of a string element can be considered for rule selection. Moreover, extensions exist that allow querying the environment [Prusinkiewicz et al. 1994], communicating with the environment [Měch and Prusinkiewicz 1996], accessing user-defined curves [Prusinkiewicz et al. 2001], filling in suitable argument values [Parish and Müller 2001], and token-based message passing among guides hosting L-systems [Beneš et al. 2011]. All of them have in common that they rely significantly on elements external to the L-system.

Classical shape grammars [Stiny 2006] involve geometric matching of (sub)shapes, and it is hence possible to take the immediate spatial context into account when deciding whether a rule can be applied. An extension [Liew 2004] further offers a few special context predicates to this end. Such a spatial context roughly corresponds to the textual context in L-systems, but because the shape specified as a rule's head only needs to occur in the context and not form the full context to yield a match, a larger class of cases is supported. However, the considered context remains purely local, and hence decisions involving multiple spatially unrelated shapes, such as acting only on the largest one, are not possible. By contrast, our language allows for such abstract contextual conditions (e.g., by involving the shapes in question in an event).

Symbolic shape grammar languages, such as CGA shape [Müller et al. 2006], basically approach support for context sensitivity by offering dedicated language elements for a few common problems (the supplemental material covers some of them in depth). The most prominent example is considering the occlusion of the current shape by some other shapes, for which query functions [Müller et al. 2006] and related operations, such as removing all occluded parts [Schwarz and Wonka 2014], were devised. Further language extensions introduce support for a limited form of alignment via consistent splits, where special snap shapes can be emitted and then taken into account by subsequent subdivision operations [Müller et al. 2006], as well as facilities for collecting (rectangular faces of) shapes, identifying pairs of them, and connecting each pair's two elements [Krecklau and Kobbelt 2011]. A related extension suggests rules that apply not to one shape but to the collection of all shapes with a given symbol [Thaller et al. 2013] (without considering the problem of when in the derivation process such a rule can and should be applied, though).

By contrast, with shapes' first-class citizenship and events, CGA++ aims for a generic solution and extends the possibilities for context-sensitive refinement significantly. In particular, making the full shape tree accessible within a grammar enables the grammar to compile a wide spectrum of contextual information (such as finding spatially close shapes). This is assisted by events, which provide a device for influencing the derivation order such that the availability of shapes required for forming a context can be ensured. Moreover, an event can be an effective means itself for establishing a complex context, where participation in the event conveniently identifies related shapes; it further allows for coordination across all the involved shapes when acting on the context. Note that the functionality of existing ad-hoc solutions (such as the above examples) can often be replicated and augmented using just CGA++'s general language features.

Additionally, for several applications, solutions external to a shape grammar language exist. One example is embedding a grammar into a feedback loop to optimize the grammar's parameters such

that the resulting building satisfies certain structural soundness criteria [Whiting et al. 2009]. Interestingly, with CGA++, such (self-sensitive) iterative (re)creation and analysis of results become possible in the grammar itself (via spawning new derivation processes and operating on shape trees). Note, however, that depending on the complexity of the computations involved in the analysis, using a grammar language for this step may not be appropriate.

7.6 Guidance of derivation order

Not least to ensure that when performing a contextual query certain shapes are already available and hence considered, exercising some form of influence on the derivation order is often necessary. Compared to concepts such as priorities [Müller et al. 2006], evaluation phases [Steinberger et al. 2014], and construction stages [Schwarz and Wonka 2014], which all merely operate globally (see the supplemental material for a critical review), CGA++ offers a more flexible and convenient solution with events, which serve as general synchronization points. In particular, local and hierarchical dependencies among derivation branches may be expressed directly and easily using event groups.

7.7 Limitations and open challenges

While CGA++ overcomes important limitations of existing grammar languages and systems, there are still many modeling tasks that cannot be handled completely, and some of those that can may be unnecessarily complex to express. For example, the provided subdivision operations, even though slightly enhanced with respect to existing systems, are insufficient to cope with situations where the split should take actual features of the shape to subdivide into account and may not even be definable by a single cut plane. Especially given that the Boolean operations enabled by CGA++ can easily introduce complex forms with holes, devising appropriate, more powerful subdivision operations is an important challenge to be addressed by future work. Another avenue is investigating how the new expressiveness can be exposed in advanced user interfaces, such as visual grammar editors [Lipp et al. 2008].

8 Conclusion

We have presented CGA++, a novel shape grammar language for the procedural modeling of architecture that increases expressiveness significantly compared to the state of the art. Cornerstones of our language are making shapes full first-class citizens and offering coordination across multiple shapes via events. They are complemented by a rich set of built-in functions and syntactic devices that facilitate a concise and comprehensible notation. As we have demonstrated with several examples, CGA++ overcomes various limitations inherent to current systems and enables many new applications and solutions to previously elusive modeling tasks.

Acknowledgments

The second author was supported in part by the EU FP7 projects INDICATE (608775) and SAFECITI (607626).

References

Barroso, S., Besuievsky, G., and Patow, G. 2013. Visual copy & paste for procedurally modeled buildings by ruleset rewriting. *Computers & Graphics 37*, 4, 238–246.

Beneš, B., Šťava, O., Měch, R., and Miller, G. 2011. Guided procedural modeling. *Computer Graphics Forum 30*, 2, 325–334.

Esri, 2014. Esri CityEngine 2014.1.

Esri, 2015. CGA shape grammar reference. http://cehelp.esri.com/help/topic/com.procedural.cityengine.help/html/cgareference/cgaindex.html.

Havemann, S. 2005. *Generative Mesh Modeling*. PhD thesis, TU Braunschweig.

Krecklau, L., and Kobbelt, L. 2011. Procedural modeling of interconnected structures. *Computer Graphics Forum 30*, 2, 335–344.

Krecklau, L., Pavic, D., and Kobbelt, L. 2010. Generalized use of non-terminal symbols for procedural modeling. *Computer Graphics Forum 29*, 8, 2291–2303.

Liew, H. 2004. *SGML: A Meta-Language for Shape Grammars*. PhD thesis, Massachusetts Institute of Technology.

Lipp, M., Wonka, P., and Wimmer, M. 2008. Interactive visual editing of grammars for procedural architecture. *ACM Transactions on Graphics 27*, 3, 102:1–102:10.

Měch, R., and Prusinkiewicz, P. 1996. Visual models of plants interacting with their environment. In *Proceedings of SIGGRAPH 96*, 397–410.

Müller, P., Wonka, P., Haegler, S., Ulmer, A., and Gool, L. V. 2006. Procedural modeling of buildings. *ACM Transactions on Graphics 25*, 3, 614–623.

Parish, Y. I. H., and Müller, P. 2001. Procedural modeling of cities. In *Proceedings of SIGGRAPH 2001*, 301–308.

Patow, G. 2012. User-friendly graph editing for procedural modeling of buildings. *IEEE Computer Graphics and Applications 32*, 2, 66–75.

Prusinkiewicz, P., and Lindenmayer, A. 1990. *The Algorithmic Beauty of Plants*. Springer-Verlag, New York.

Prusinkiewicz, P., James, M., and Měch, R. 1994. Synthetic topiary. In *Proceedings of SIGGRAPH 94*, 351–358.

Prusinkiewicz, P., Mündermann, L., Karwowski, R., and Lane, B. 2001. The use of positional information in the modeling of plants. In *Proceedings of SIGGRAPH 2001*, 289–300.

Schwarz, M., and Wonka, P. 2014. Procedural design of exterior lighting for buildings with complex constraints. *ACM Transactions on Graphics 33*, 5, 166:1–166:16.

Steinberger, M., Kenzel, M., Kainz, B., Müller, J., Wonka, P., and Schmalstieg, D. 2014. Parallel generation of architecture on the GPU. *Computer Graphics Forum 33*, 2, 73–82.

Stiny, G., and Gips, J. 1972. Shape grammars and the generative specification of painting and sculpture. In *Information Processing 71*, 1460–1465.

Stiny, G. 1980. Introduction to shape and shape grammars. *Environment and Planning B 7*, 3, 343–351.

Stiny, G. 1982. Spatial relations and grammars. *Environment and Planning B 9*, 1, 113–114.

Stiny, G. 2006. *Shape: Talking about Seeing and Doing*. MIT Press.

Thaller, W., Krispel, U., Zmugg, R., Havemann, S., and Fellner, D. W. 2013. Shape grammars on convex polyhedra. *Computers & Graphics 37*, 6, 707–717.

Whiting, E., Ochsendorf, J., and Durand, F. 2009. Procedural modeling of structurally-sound masonry buildings. *ACM Transactions on Graphics 28*, 5, 112:1–112:9.

Wonka, P., Wimmer, M., Sillion, F. X., and Ribarsky, W. 2003. Instant architecture. *ACM Transactions on Graphics 22*, 3, 669–677.