Available online at www.sciencedirect.com
ScienceDirect
Procedia Computer Science 00 (2017) 1–31
Procedia Computer Science

Tiling Algebra for Constraint-based Layout Editing

Clemens Zeidler[a], Gerald Weber[a], Alex Gavryushkin[b], Christof Lutteroth[c,a]

[a] *University of Auckland,*
38 Princes Street, Auckland 1010, New Zealand
[b] *ETH Zürich*
Mattenstrasse 26, 4058 Basel, Switzerland
[c] *University of Bath*
Claverton Down, Bath BA2 7AY, UK

Abstract

The constraint-based layout model is a very powerful model to describe a wide range of graphical user interface (GUI) layouts, based on linear constraints. However, the advantages of the constraint-based layout model come at a price: layout designers have to ensure layouts are sound, i.e., they are solvable and items in the layout do not overlap each other. Keeping a layout sound is non-trivial since editing one constraint may have undesirable effects on other constraints.

In this article, we propose a new formalism for constraint-based layouts which we call a tiling algebra. Editing operations on layouts are specified algebraically, which guarantees that these operations keep a layout sound. We propose to model tiling operations with two operators that are isomorphic cancellative semigroup operators with involution if seen as binary operators. While these semigroup operators alone already cover an interesting subset of layouts, called fragments, we show that there are more involved layouts, such as the pinwheel layout, which cannot be modelled with these operators alone. For this reason we introduce a third operator which is isomorphic to a Boolean conjunction.

Our approach helps to describe constraint-based layouts correctly and to make layout editing robust. We apply the proposed algebra to two real constraint-based systems, which illustrate how it can be used to support sound layout creation and modification.

Keywords: constraint-based layout model, algebra, GUI layout, layout editing, sound layouts

1. Introduction

Most graphical user interfaces (GUIs) use layout models to position GUI items, e.g., to place widgets in a window. Using layout models simplifies the layout specification and makes GUIs resizable, i.e., the GUI can adapt to different window and screen sizes. A promising layout model is the constraint-based layout model, which specifies a layout using linear constraints [1, 2]. Figure 1 shows an example of a constraint-based layout where widgets are constrained to certain *tabstops*, i.e., horizontal and vertical lines.

The constraint-based layout model can describe a wide range of layouts and is more powerful than most other layout models such as the grid-bag layout model [3]. For example, the linear layout model simply arranges layout items in a row or a column while the grid-bag layout model arranges layout items in a fixed grid; both layout models can be described using a constraint-based layout model. Figure 2 shows an example of a constraint-based layout which has a resize behavior that cannot be achieved with most other layout models. This makes the constraint-based layout model a versatile tool to design good GUI layouts [4].

Email addresses: clemens.zeidler@auckland.ac.nz (Clemens Zeidler), g.weber@auckland.ac.nz (Gerald Weber), alex@gavruskin.com (Alex Gavryushkin), c.lutteroth@bath.ac.uk (Christof Lutteroth)

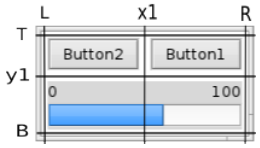

Figure 1. Widgets are aligned to tabstops. Here we have four outer tabstops L, T, R, B and two inner tabstops x1 and y1.

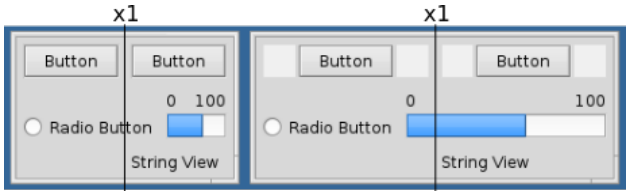

Figure 2. Example constraint-based layout. The behavior of the middle row is independent of the top and bottom rows when resizing. The tabstop $x1$ that separates the two buttons is connected to the left side of the string view. This is impossible to achieve with a gridbag layout.

The set of constraints that describes a complete GUI layout is called a *layout specification*. One problem of the constraint-based layout model is that it can be cumbersome and difficult to create and manage complex layout specifications. Dependencies between constraints may not be obvious to the designer, and it can be difficult to tell from a specification how the specified GUI layout will look and behave. Editing a specification can be difficult, e.g., replacing a variable in a constraint may have undesired effects in another constraint using the same variable.

Another problem is that of soundness: specifications may have certain properties that make them undesirable in GUI layouts, i.e., they may be "unsound". For example, a layout specification may contain conflicting constraints, i.e., the constraint system is not solvable, which means that the GUI cannot be rendered. Furthermore, layout items in a GUI should not overlap each other to ensure that all items are completely visible and accessible at all layout sizes. Note that we need to differentiate between *concrete layouts*, as they are rendered on a screen, and more general *layout specifications*, which are typically size-independent and can be rendered at many different sizes, leading to many possible concrete layouts. For example, the layout on the left in Figure 3 appears non-overlapping, but if the layout size is reduced as shown on the right, the check boxes overlap the buttons due to a missing constraint. We call a layout specification *sound* if it is solvable and cannot have overlapping layout items [4].

Creating software applications for editing constraint-based layouts in an easy-to-use and sound way is hard. The problem of sound constraint-based layout editing has never been formalized and understood theoretically. As a result, many such applications are ad hoc, without theoretical underpinning, and unsound layout specifications can be created easily. In this article we introduce a tiling algebra that addresses the aforementioned problems of constraint-based layouts: First, the tiling algebra provides a formal but intuitive notation for layout specifications, i.e., a notation that reflects the arrangement of layout items, while being powerful enough to describe all meaningful layout specifications. The algebra is based on two sibling operators, horizontal and vertical tiling. Simple cases of these operators turn out to form two cancellative semigroups with involution, and they are naturally isomorphic to each other. For specific complex layouts, however, a third non-cancellative, idempotent, commutative semigroup operator is needed.

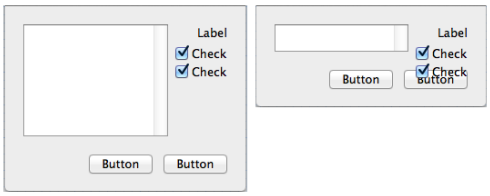

Figure 3. The constraint-based layout on the left appears sound. Yet, when decreasing its size, as shown on the right, an overlap between widgets occurs due to a missing constraint.

Interestingly, the two tiling operators do not naturally follow the standard distributivity law, but a similar law we call tiling distributivity. All *non-overlapping orthogonal tilings* can be described, i.e., all layout specifications that are based on non-overlapping rectangles in a rectangular grid. Tiles can also be empty, e.g., to describe empty space within a layout. Secondly, the tiling algebra ensures that a layout specification remains sound while it is created or modified, by formally specifying sound edit operations. This helps to make layout editing applications robust so that layouts are kept sound during editing.

We illustrate the benefits of the algebra in two case studies. The first system is called Stack & Tile [5] and is an extension of a classic window manager, where the user can combine multiple windows into groups. In Stack & Tile, the user can stack windows on top of one another and tile windows beside each other. By applying the proposed algebraic concepts, valid window group combinations can be created using just two layout edit operations. The second system is a constraint-based GUI builder, the Auckland Layout Editor (ALE) [4], which allows GUI designers to specify and edit complex constraint-based GUI layouts. ALE makes use of the proposed algebra to offer full-fledged layout editing with only a few operations and keep layouts sound at all times. We show how the edit operations of Stack & Tile and ALE can be mapped to operations of the tiling algebra, and how this helps to ensure soundness. Other systems that could benefit from the presented algebra include editors for document layout, web layout, graphics design, CAD and user interface customization.

In summary, this article makes the following contributions:

- A tiling algebra which provides a formal notation for constraint-based layouts that visually reflects the arrangement of layout items.
- A set of algebraic edit operations to create and edit layout specifications.
- Proof that these edit operations leave a sound layout in a sound state.
- Two case studies that illustrate how the algebra helps in the development of layout editing applications.

Section 2 discusses related work on GUI description languages, other algebraic approaches, and constraint-based layout. Sections 3 and 4 give a short description of Stack & Tile and ALE to motivate our contributions. Section 5 introduces the tiling algebra and Section 6 describes its semantics. Section 7 defines the algebraic edit operations. Section 8 discusses how our approach can be applied to real-world problems. Section 9 gives concrete examples of how the tiling algebra is used in Stack & Tile and ALE. Section 10 concludes the article.

2. Related Work

2.1. Description Languages and Other Algebraic Approaches for GUIs

There are many algebraic languages for software specification [6, 7]. The algebraic language presented in this article is a domain-specific description language for user interfaces. A user interface description language provides an abstract definition of a GUI, and depending on the problem, it can have different abstraction layers [8, 9, 10, 11, 12]. Abstraction makes it possible to describe GUIs for multiple platforms. In contrast to previous user interface description languages, our algebra considers also the soundness of layout editing operations.

Algebraic approaches have been proposed before in the user interface domain. Matrix algebra [13] has been used in order to describe push button interfaces. The authors demonstrated how the algebra can be successfully applied to a simple real-world application such as a calculator. Algebraic graph grammars [14] were proposed as a formalism for specifying editors for graph-like visual languages. In the domain of geo-information systems map algebra has been used to describe how different types of maps can be combined to form a new map [15]. Moreover, algebraic approaches have been used to describe the content of visual data. Feature algebra [16] has been used to formalize visual database queries. Image algebra [17] has been proposed as a way to manipulate and transform symbolic images, e.g., for spatial reasoning.

The extents of layout items in a GUI can be regarded as spatial intervals. Allen [18] introduced a formal language to describe relations of intervals. In this interval language the relative position of two intervals can be specified, e.g., if an interval is equal to, is before, meets or overlaps the second interval. While the original language considered only

temporal intervals, the language has been augmented to describe spatial multi-dimensional intervals algebraically [19, 20]. Specifically, it has been applied to describe the block layouts of elements in a document [21].

The algebra presented here is comparable to the multi-dimensional interval algebra described in [21] in a sense that it describes block layouts. Because we address the specification of resizable GUIs, only relations between layout items that hold for all layout sizes are described. We consider only adjacent intervals and most importantly, we consider how the soundness of a layout can be ensured when using algebraic operations. Moreover, we allow specifying horizontal and vertical relations of multiple layout items in a single algebra term. This greatly helps to reason about soundness.

2.2. Constraint-based Layout

Formal description languages for GUI layouts are often based on constraints [22, 23, 24]. Constraints are used to specify the desired characteristics of a layout and can be combined using logical operators. Particularly linear constraints have been used for GUI layout, as they are sufficiently expressive for common GUIs and fairly easy to solve [1, 2, 25]. Constraints can be used to place layout items at certain absolute coordinates or relative to other layout items. In our algebra, layout items are only constrained relative to each other, which makes it possible to define fairly simple algebraic edit operations. We consider linear constraints of the form

$$\sum_i a_i \cdot x_i = b$$

with the variables x_i and $a_i, b \in \mathbb{R}$. Inequalities are defined analogously. There are two common types of linear constraints: *hard* constraints, which have to be satisfied, and *soft* constraints, which may be violated if necessary.

The constraint-based layout model, as used for Stack & Tile and ALE, is used as follows. GUI layouts consist of layout items, which have properties and are arranged relative to other layout items. Generally, layout items can be any rectangular objects, but without loss of generality, we call them just widgets in the following. Each widget has a *minimum*, a *preferred* and a *maximum* size, also called *intrinsic* sizes. A widget should assume its preferred size if there are no other constraints on it, similar to the behavior of a sponge. When inserting a widget into a layout, some constraints are automatically derived from the intrinsic sizes of a widget: a hard inequality is created for the minimum size, a soft equality for the preferred and a soft inequality for the maximum size. These constraints are added in both the horizontal and vertical direction.

Variables in a constraint are called *tabstops* and represent horizontal or vertical grid lines. Other frequently used names for the same concept are aligners, guides, snap lines or anchor lines. The edges of each widget are always connected to two horizontal (top and bottom) and two vertical (left and right) tabstops, which form a rectangular *area*. Connecting widgets to the same tabstop aligns them. Figure 1 shows a simple layout and its tabstops. Each GUI container, such as a panel or a window, defines tabstops for its four borders.

If each widget is connected to at least one horizontal and one vertical tabstop, the layout is sufficiently specified and can be computed by solving the constraint system. The intrinsic size constraints determine the size and the tabstops determine the position of each widget. A constraint solver controls how soft constraints are handled, which determines the final visual appearance of the layout. In the constraint-based systems discussed later, the quadratic active set method is used [26]. In contrast to a linear approach, this leads to unique layouts with certain aesthetic properties [27]. An overview of different solving techniques can be found in other works [1, 28, 29].

A sufficient minimum size of the widgets can be guaranteed by a sufficiently large minimum size of the outermost GUI container, e.g., the window. In this case, a layout always stays solvable as only a soft inequality constraint is used for the maximum size. If a maximum size constraint is violated, then more space is allocated than a given widget is able to use. By default, the widget is then aligned in the center of its allocated space, as illustrated on the right side of Figure 2.

3. The Stack & Tile Window Manager

In Stack & Tile, users can stack and tile windows together to create window groups according to their needs. Groups behave like single windows, and can be used together with other windows as usual. Stack & Tile combines the advantages of stacked and tiled windows with the freedom of overlapping windows (see Figure 4).

Stack & Tile can help users to manage their windows more effectively [5]. Within an active Stack & Tile group, tiled windows are always visible and not occluded by other windows. This makes it easier to exchange data between them. A window stack can be used to group windows together if their content does not need to be visible at the same time. By grouping the windows used for a certain task, this can result in a cleaner desktop and facilitate switching between tasks that involve multiple windows.

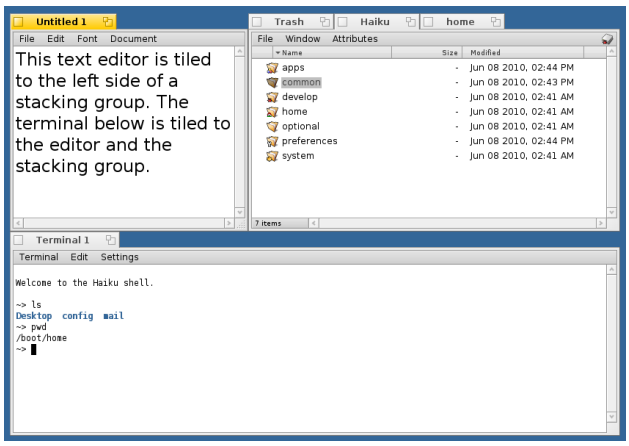

Figure 4. Example of a Stack & Tile group. On the right side, three windows are stacked into a stacking group. Tiled to this group are on the left side a text editor and at the bottom a terminal.

Internally linear constraints are used to describe a Stack & Tile group, employing the tabstop and area system of the Auckland Layout Model (ALM) [2]. Areas are simply the tiles where windows can be placed, and tabstops are their borders. For example, two stacked windows are sharing the same tabstops and are therefore always getting the same size. Window operations modify the constraint specification that describes all Stack & Tile groups, and the window manager solves the specification and re-renders the windows.

Currently, there is a fully working Stack & Tile implementation available, which has been integrated into the default Haiku OS[1] user interface. It is well known and appreciated in the Haiku OS community.

3.1. Stack & Tile Operations

Stack & Tile offers two simple operations that can be used to connect windows: First, stacking, which makes use of the tab-like appearance of the Haiku OS window title bars; secondly, tiling of windows, which means that windows are arranged beside each other. A Stack & Tile operation can be triggered by holding down the Stack & Tile key, which is by default the Windows key, and dragging a window near to another window (see Figure 5). Releasing the Stack & Tile key or dropping the window finally executes a Stack & Tile operation. The dragged window is called the *candidate window*, and the window that it is dragged to is called the *parent window*. In this manner, *Stack & Tile groups* can be created.

Stacking: Briefly, a stacking operation is triggered by moving the title tab of the candidate window onto the title tab of a parent window while holding the Stack & Tile key. To be more precise, the candidate window has to be dragged by the title tab so that the upper edge of the candidate window tab is on the parent window tab, and the x-position of the mouse cursor is in the x-range of the parent title tab. When a valid stacking candidate-parent pair is found, the window title tabs of both windows are highlighted (left of Figure 5).

After stacking windows on top of one another, the stacked windows have the same position and size. The title tabs are automatically arranged beside each other so that the stacked windows are accessible over a tab interface at the top of the stack. The result is comparable with a tab bar, e.g. in a tabbed web browser, with a similar functionality for reordering tabs.

[1] `www.haiku-os.org`

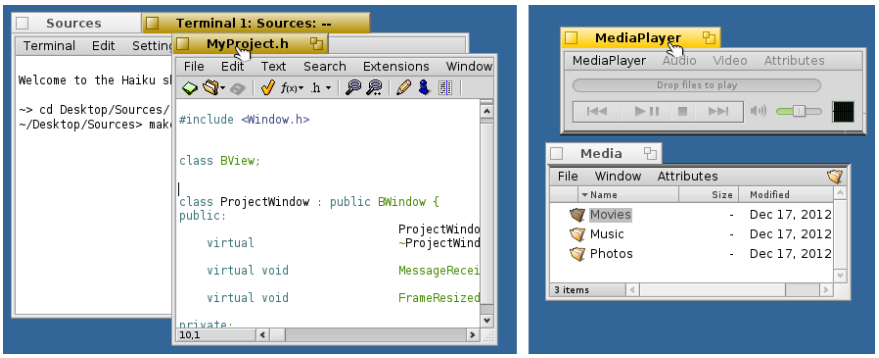

Figure 5. Moving a window while holding the Stack & Tile key initiates stacking or tiling. Window tabs and borders are highlighted in yellow and gray, respectively. Left: The editor window is going to be stacked on top of the Terminal. Right: MediaPlayer is going to be tiled to the Media folder.

Tiling: If the Stack & Tile key is pressed, dragging and dropping a candidate window border close to one or more parent window borders triggers the tiling operation (right of Figure 5). Tiled windows always share a border position. For example, when tiling two windows horizontally, the right border of the left window always has the same position as the left border of the right window. Furthermore, top and bottom of both windows are aligned to each other. Windows can be tiled at any arbitrary free rectangular region of a group and can span several windows. Figure 4 shows an example of a tiled window group.

Removing from a Group: A window can be removed from a Stack & Tile group by holding down the Stack & Tile key and dragging the window away from the group. After removing the window from the group, the window behaves just like an ordinary window in the desktop metaphor. In case the removed window was the only tiling connection between other windows in the group, the group is split into several independent groups. For example, if five windows are tiled in a row and the middle window is removed, the group is split into two: one group comprising the first two and one group comprising the last two windows.

3.2. Traditional Window Management Operations

When interacting with a Stack & Tile group the semantics of the traditional window management operations change slightly. Stack & Tile applies window management operations to multiple windows, which has already been considered helpful in Elastic Windows [30].

Activating one window in a Stack & Tile group raises all windows in the Stack & Tile group. The window that triggered the group operation gets the input focus.

Moving a window by a certain offset also moves all other group windows by the same offset. This means windows in a Stack & Tile group keep their relative positions to each other.

Resizing one window in a Stack & Tile group leaves all windows in the group aligned to each other. For example, windows that are tiled to a resized window are moved or resized accordingly. This is done by temporarily setting a high priority for the size constraint of the resized window. In this way, the resized window gets its new size and the other windows adapt according to the solution of the constraint system.

Hiding or **showing** a window also hides or shows all other windows in the group. Thus, all windows in a Stack & Tile group are either hidden or shown.

4. The Auckland Layout Editor (ALE)

Similar to other GUI builders, ALE's user interface consists of a component palette and an editing canvas (Figure 6). New widgets can be dragged from the palette into the editing canvas, where the designer can change the layout using a rich set of edit operations, as described in the following.

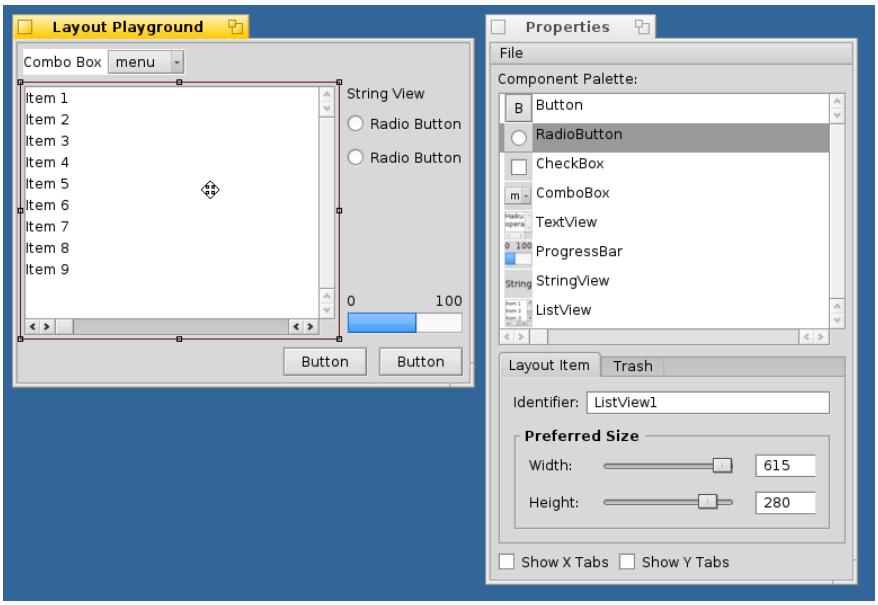

Figure 6. Screenshot of the ALE GUI builder. New widgets can be inserted into the editing canvas (left) via drag and drop from the component palette (right).

4.1. ALE Layout Edit Operations

Ideally, a GUI builder should make the creation of a layout easy to learn, simple to achieve, and quick to perform. ALE's edit operations for constraint-based layouts automatically keep widgets aligned to each other as far as possible, by connecting them to existing tabstops. This makes it easier to rearrange widgets while keeping the layout consistent. Furthermore, the edit operations leave a widget connected to at least one horizontal and one vertical tabstop. This ensures that the position of a widget can always be determined. The provided edit operations are moving, swapping, resizing, inserting, and removing of widgets. These operations enable the user to create complex layouts without additional constraint authoring. Moreover, an algorithm automatically manages all necessary constraints to keep a layout non-overlapping [4].

As in most GUI builders, a GUI can be created and edited by drag and drop operations. Edit operations are started by dragging an already placed widget or the border of a widget. Then, ALE checks at each mouse position if an operation can be applied and gives corresponding visual feedback (Figure 7). For this, the operation is applied tentatively in the background. If an operation is appropriate, the user can commit the operation by "dropping" the dragged item and the result becomes visible.

Moving a Widget While dragging a widget, its shape is visualized as a dotted rectangle. Two cases need to be considered when moving a widget from one position to another. First, a widget can be inserted into an empty area (Figure 7). To identify the correct empty area, we attempt to snap the dragged rectangle to existing tabstops. If the widget can only be snapped on one side in a certain direction, horizontally or vertically, a new tabstop is inserted at the opposite side. If a widget cannot be snapped to any tabstop in a certain direction, then two other suitable tabstops have to be found. These are the respective tabstops of the largest empty rectangular area at the cursor position.

The combination of snapping widgets to existing tabstops and placing them into the largest empty area at the cursor makes the move operation quite versatile. For example, it is possible to place a small item accurately into the corner of an empty area by snapping it on two borders. Dropping a widget roughly in the middle of an empty area will fill the area with the widget. Furthermore, a widget can, e.g., be placed in the left half of an empty area by moving the item's left border close to the area's left border midway between top and bottom. ALE visualizes the location where a widget will be placed when "dropped" with a green rectangle (Figure 7).

Second, a widget can be placed between an existing tabstop and a widget adjacent to that tabstop. This happens when a widget is dropped close to the existing tabstop and on the adjacent existing widget (Figure 8). In the following, only the insertion at a vertical tabstop is described; the horizontal case works analogously. When dropping the dragged

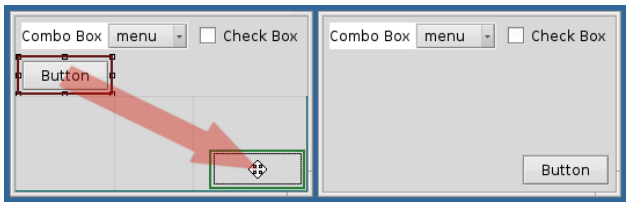

Figure 7. Move operation: Dragging the button to the bottom-right of the empty area. The area where the button will be inserted when dropped is highlighted in green.

widget, it is inserted so that its top and/or bottom are connected to those of the existing widget using the same rules as in the first case: if the top or bottom of the dragged rectangle is close to the top or bottom of the existing widget, then they are snapped together. If only one side is snapped, either top or bottom, a new tabstop is created at the opposite side. If the dragged rectangle is not close enough to the top or bottom of the existing widget, then the dragged widget is connected to both the top and bottom of the existing widget. In Figure 8, the string view is snapped to the bottom of the right button. The left and right tabstops of the moved widget are then set so that the widget is placed between the vertical tabstop and the existing widget.

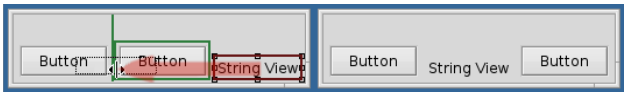

Figure 8. Move operation: Moving a string view between two buttons.

Swapping Two Widgets The swap operation simply swaps the position of a widget with the position of another widget. Here it is sufficient to connect the moved widget to the tabstops of the other widget and vice versa.

Resizing a Widget Dragging one of the borders or corners resizes a widget and allows the user to connect the dragged borders to different tabstops. During resizing, all relevant tabstops are visualized as light blue lines to aid alignment. A resize operation is aborted if the enlarged widget would overlap other widgets.

There are two cases to consider for resizing. First, a widget can be resized to an existing tabstop, by dragging and snapping it to said tabstop (Figure 9). Second, an item can be resized to match its preferred size when a dragged border is released without snapping it to a tabstop. In this case, a new tabstop is inserted for the dragged border. This can also be interpreted as detaching a widget from a tabstop; the so "freed" widget gets its preferred size. Finally, the resize behavior of widgets can be controlled manually via the preferred size settings in a properties window (see right side of Figure 6).

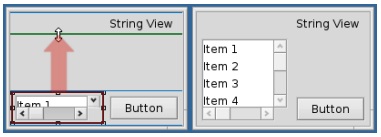

Figure 9. Resizing the top of the list widget to the bottom tabstop of the string view.

Inserting and Removing Widgets

Inserting a new widget is practically identical to the move operation. A widget can be removed from the layout by dragging it outside of the layout and dropping it.

4.1.1. Filling Gaps

A move, resize, or remove operation can detach adjacent widgets, as one or more items may loose their connection to a tabstop. In this case, a gap may appear in the layout (Figure 10) and the position of some widgets may become undefined in the constraint system; the items "float".

ALE avoids such situations by checking for unconnected widgets after move, resize, and remove operations. If the layout contains parts that are unconnected, all groups of "floating" widgets are moved one after another into the direction of the removed or resized widget. If a group was connected to the right side of the removed or resized item, the group is moved to the left, and similarly for other directions. When the foremost widget of the floating group hits the border of another item, it is connected with the corresponding tabstop of the other item (Figure 10). During this procedure, the moved group may get connected to another floating group. All floating groups are moved until they are connected directly or indirectly to at least one horizontal and one vertical layout border.

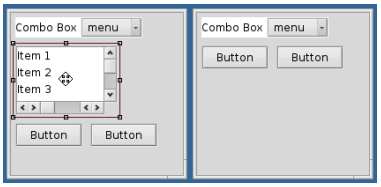

Figure 10. Filling gaps: Upon removal of the list widget (left), the two buttons both "float" vertically. Moving the group of both buttons to the top fills the gap. The top of the buttons is now connected to the bottom tabstop of the combo box (right).

5. Tiling Algebra

Stack & Tile, as well as ALE, have some similarities when it comes to layout editing. Both systems use rectangular areas that are constrained to be adjacent to each other and need to keep layouts in a sound state. That is, all these rectangular areas are not permitted to overlap. Furthermore, both systems use a constraint-based model and all edit operations need to leave the constraint system in a valid and solvable state.

In this section, a new algebraic approach is presented, which is able to describe layouts of rectangular areas that are connected with each other. Every area is enclosed by four tabstops, i.e., the left, top, right and bottom tabstops the area is connected to. For GUI layouts, the content of an area is usually a widget such as a button or a text view, while for Stack & Tile the content is a window. We provide definitions and axioms that describe the syntax of the proposed algebra.

The algebra is based on two sets, one for tabstops and one for areas. Areas and tabstops can be combined with tiling operators to form more complex layouts which we call fragments. We define the set of all layout specifications layoutSpecs as a superset of the set of fragments. A layoutSpec can be seen as a conjunction of fragments.

Definition 1 (Tabstop)**.** A *tabstop* is an element of either the set of all vertical tabstops $Tabstop_x$ or the set of all horizontal tabstops $Tabstop_y$.

Definition 2 (Area)**.** An *area* is an element of the set $Area$.

We now introduce the tiling operators that are characteristic of our algebraic approach. There are two tiling operators: the slash $/$ operator expresses the relation between vertically adjacent areas ("on top of"), and the pipe $|$ operator expresses the relation between horizontally adjacent areas ("left of"). We introduce the tiling algebra in the style of a *term algebra* [31]. $Area$ is the set of ground terms of the algebra, and $/$ and $|$ are the basic operations. The set $Fragment \supseteq Area$, which is inductively defined over the operations, contains terms of the algebra.

Definition 3 (Tiling Operators)**.** The horizontal and vertical tiling operators $/$ and $|$ map two fragments and a tabstop to a new fragment.

Type:

$$|: \quad Fragment \times Fragment \times Tabstop_x \rightarrow Fragment$$
$$/: \quad Fragment \times Fragment \times Tabstop_y \rightarrow Fragment$$

Notation:

$$|(A, B, x) = A \underset{x}{|} B$$
$$/(A, B, y) = A \underset{y}{/} B$$

Note that $/$ and $|$ can be written as binary operators as a shorthand: the tabstop in the third parameter can be omitted if it does not already occur in the fragments of the first two parameters and if it does not need to be referred to in other tiling operations. For example, the expression A/B means that an area A is on top of an area B; $C|D$ means that C is on the left of D.

Definition 4 (Fragments)**.** The set $Fragment$ is defined inductively as follows: The atoms are the *zero fragment* 0 and all elements from the set $Area$, i.e., $Fragment \supseteq Area \cup \{0\}$. If $A, B \in Fragment$ and $x \in Tabstop_x$ then $A\underset{x}{|}B \in Fragment$; and if $C, D \in Fragment$ and $y \in Tabstop_y$ then $C\underset{y}{/}D \in Fragment$. There are no elements in $Fragment$ other than those defined above.

A fragment represents the intuitive concept of a hierarchical layout topology, i.e., abstract arrangements of areas which are combined using only $/$ and $|$. Intuitively, the fragment A/B means that all areas in A are above all areas in B. Similarly, $A|B$ means that all areas in A are on the left of all areas in B. A single fragment specifies many concrete layouts of the same topology but with different area positions and sizes. For example, the fragment $(A\underset{x}{|}B)\underset{y}{/}C$ describes a layout as shown in Figure 12. This assumes that the fragment describes a complete rectangular layout with the "outer" borders of the layout items implicitly connected to the layout borders (see Section 5.1 for details).

Non-hierarchical, "interlocked" layouts such as the pinwheel in Figure 13 require a description that consists of more than one fragment. Fragments can be combined into more general layout specifications, which are elements of the set *LayoutSpec*, using the asterisk operator $*$.

Definition 5. A layout specification, *layoutSpec* for short, is an element of the set *LayoutSpec* with $Fragment \subseteq LayoutSpec$.

Definition 6. The asterisk binary operator $*$ combines two layoutSpecs.

Type:

$$* : \quad LayoutSpec \times LayoutSpec \rightarrow LayoutSpec$$

Notation:

$$*(A, B) = A * B$$

The operator precedence for the $*$ operator is lower than for the $|$ and $/$ operators. However, the operators $|$ and $/$ have the same precedence.

We define the following axioms for the $*$ operator. These axioms mirror important axioms of the conjunction in a Boolean algebra. In our application scenario, there is no need for a dual operator, which would represent a Boolean disjunction.

Axiom 1 (Associativity)**.** The $*$ operator is associative:

$$(A * B) * C = A * (B * C).$$

Axiom 2 (Commutativity)**.** The $*$ operator is commutative:

$$A * B = B * A.$$

Axiom 3 (Idempotence)**.** The $*$ operator fulfills the axiom of idempotence:

$$A * A = A.$$

The $*$ operator defines an idempotent, commutative semigroup that is not cancellative. That it is not cancellative can be seen in the following example. Lets assume that $B = A * C$ and $B \neq C$. Using idempotence we can write

$$A * B = A * A * C = A * C$$

which means that from $A * B = A * C$ does not follow that $B = C$.

The axioms for the tiling operators are defined as follows:

Axiom 4 (Tiling Idempotence)**.**

$$A \underset{x}{|} B = A \underset{x}{|} B * A = A \underset{x}{|} B * B$$

$$A \underset{y}{/} B = A \underset{y}{/} B * A = A \underset{y}{/} B * B.$$

This means a layoutSpec that contains two tiled fragments still contains these two fragments.

Axiom 5 (Tiling Associativity)**.** The tiling operators are associative:

$$(A \underset{l}{|} B) \underset{m}{|} C = A \underset{l}{|} (B \underset{m}{|} C)$$

and

$$(A \underset{l}{/} B) \underset{m}{/} C = A \underset{l}{/} (B \underset{m}{/} C).$$

Axiom 6 (Concatenation)**.** A layoutSpec of the form $A \underset{l}{|} B * B \underset{m}{|} C$ or $A \underset{l}{/} B * B \underset{m}{/} C$ can be concatenated to a fragment:

$$A \underset{l}{|} B * B \underset{m}{|} C = A \underset{l}{|} B \underset{m}{|} C$$

and

$$A \underset{l}{/} B * B \underset{m}{/} C = A \underset{l}{/} B \underset{m}{/} C.$$

Axiom 7 (Parallelism)**.** Fragments which are tiled with the same tabstop can be rewritten as follows. For the horizontal case

$$A \underset{x}{|} B * C \underset{x}{|} D = A \underset{x}{|} D * C \underset{x}{|} B$$

and for the vertical case

$$A \underset{y}{/} B * C \underset{y}{/} D = A \underset{y}{/} D * C \underset{y}{/} B.$$

Amongst each other, the tiling operators $|$ and $/$ do not fulfill the common distributivity laws. However, the following axiom holds, which is reminiscent of distributivity.

Axiom 8 (Tiling Distributivity)**.**

$$(A \underset{x}{|} B) \underset{y}{/} C = A \underset{x}{|} B * A \underset{y}{/} C * B \underset{y}{/} C$$

$$A \underset{x}{|} (B \underset{y}{/} C) = A \underset{x}{|} B * A \underset{x}{|} C * B \underset{y}{/} C.$$

and

$$(A \underset{y}{/} B) \underset{x}{|} C = A \underset{y}{/} B * A \underset{x}{|} C * B \underset{x}{|} C$$

$$A \underset{y}{/} (B \underset{x}{|} C) = A \underset{y}{/} B * A \underset{y}{/} C * B \underset{x}{|} C.$$

Example 1. The grid layout in Figure 14 can be specified as a single fragment:

$$(A \underset{x}{|} B) \underset{y}{/} (C \underset{x}{|} D).$$

This can be transformed to

$$(A \underset{y}{/} C) \underset{x}{|} (B \underset{y}{/} D).$$

The steps for this transformation are listed below (commutativity and associativity is used implicitly):

$$(A \underset{x}{|} B) \underset{y}{/} (C \underset{x}{|} D) \tag{1}$$

$$\overset{\text{Tiling Distributivity (Axiom 8)}}{=} A \underset{x}{|} B * A \underset{y}{/} (C \underset{x}{|} D) * B \underset{y}{/} (C \underset{x}{|} D) \tag{2}$$

$$\overset{\text{Tiling Distributivity (Axiom 8)}}{=} A \underset{x}{|} B * A \underset{y}{/} C * A \underset{y}{/} D * C \underset{x}{|} D * B \underset{y}{/} C * B \underset{y}{/} D * C \underset{x}{|} D \tag{3}$$

$$\overset{\text{Idempotence (Axiom 3)}}{=} A \underset{y}{/} C * A \underset{y}{/} D * B \underset{y}{/} C * B \underset{y}{/} D * C \underset{x}{|} D * A \underset{x}{|} B \tag{4}$$

$$\overset{\text{Parallelism (Axiom 7)}}{=} A \underset{y}{/} C * A \underset{y}{/} C * B \underset{y}{/} D * B \underset{y}{/} D * C \underset{x}{|} D * A \underset{x}{|} B \tag{5}$$

$$\overset{\text{Idempotence (Axiom 3)}}{=} A \underset{y}{/} C * B \underset{y}{/} D * C \underset{x}{|} D * A \underset{x}{|} B * C \underset{x}{|} D * A \underset{x}{|} B \tag{6}$$

$$\overset{\text{Parallelism (Axiom 7)}}{=} A \underset{y}{/} C * B \underset{y}{/} D * C \underset{x}{|} D * A \underset{x}{|} B * C \underset{x}{|} B * A \underset{x}{|} D \tag{7}$$

$$\overset{\text{Tiling Distributivity (Axiom 8)}}{=} A \underset{y}{/} C * C \underset{x}{|} (B \underset{y}{/} D) * A \underset{x}{|} (B \underset{y}{/} D) \tag{8}$$

$$\overset{\text{Tiling Distributivity (Axiom 8)}}{=} (A \underset{y}{/} C) \underset{x}{|} (B \underset{y}{/} D). \tag{9}$$

Axiom 9 (Zero Fragment)**.** There is an element $0 \in \mathit{Fragment}$ which is the absorbing element of $|$, $/$ and $*$:

$$S \,\alpha\, 0 = 0 \quad \text{and} \quad 0 \,\alpha\, S = 0 \quad \text{with} \quad \alpha \in \{|, /, *\}. \tag{10}$$

This extends the notion of 0 from fragments to all layout specifications and clarifies that the zero fragment is indeed the zero in all the three semigroups (for the $*$, $|$ and $/$ operators). From semigroup theory, we know that a zero has to be unique, and hence the zero-fragment notation is unambiguous. 0 denotes an infeasible layout specification, i.e., a specification with conflicting constraints as described in Section 6.

Axiom 10 (Cancellation)**.** The tiling operators are cancellative:

$$A \underset{x}{|} B = A \underset{x}{|} C \neq 0 \Rightarrow B = C \;\text{ and }\; B \underset{x}{|} A = C \underset{x}{|} A \neq 0 \Rightarrow B = C$$

and

$$A \underset{y}{/} B = A \underset{y}{/} C \neq 0 \Rightarrow B = C \;\text{ and }\; B \underset{y}{/} A = C \underset{y}{/} A \neq 0 \Rightarrow B = C.$$

Note that without the tabstobs, both tiling operators, as binary operators, are cancellative semigroup operators. The tiling operator semigroups have an involution; this involution expresses the symmetry of the tiling operators with respect to reflection, parallel to the axis. The tiling operator semigroups are isomorphic because both dimensions are treated identically, as are both directions in every dimension.

Axiom 11 (Infeasibility)**.** If a fragment contains the same tabstop on opposite sides (i.e., on the left and the right, or the top and the bottom), it is the zero fragment 0, i.e.,

$$A \underset{x}{|} B \underset{x}{|} C = A \underset{x}{|} 0 \underset{x}{|} C \quad \text{and} \quad A \underset{y}{/} B \underset{y}{/} C = A \underset{y}{/} 0 \underset{y}{/} C.$$

In other words the tabstop x cannot be at two different places at the same time.

Axiom 12 (Tabstop Uniqueness)**.** The tabstop between a fragment A and all fragments that are tiled to A on one side is unique:

$$C \underset{l}{|} A * B \underset{m}{|} A \neq 0 \;\Rightarrow\; l = m \quad \text{and} \quad A \underset{l}{|} B * A \underset{m}{|} C \neq 0 \;\Rightarrow\; l = m \tag{11}$$

$$\text{and} \qquad C \underset{l}{/} A * B \underset{m}{/} A \neq 0 \;\Rightarrow\; l = m \quad \text{and} \quad A \underset{l}{/} B * A \underset{m}{/} C \neq 0 \;\Rightarrow\; l = m. \tag{12}$$

Definition 7 (Chain)**.** A fragment c that contains either only $/$ or only $|$ is called a chain.

For example:

$$c = \underset{x_1}{A_0} \,|\, \underset{x_2}{A_1} \,|\, ... \;|\; \underset{x_{n-1}}{A_{n-1}} \,|\, \underset{x_n}{A_n}.$$

Proposition 1. A chain containing a tabstop twice is a zero fragment.

Proof. This proof only describes the horizontal case; the vertical case is identical. A chain that contains a tabstop x twice can be written as:

$$c = \underset{a_0}{A_0} \,|\, \ldots \;|\; \underset{a_{i-1}}{A_i} | \underset{x}{A_{i+1}} \;|\; \ldots \;|\; \underset{a_{i+1}}{A_j} | \underset{x}{A_{j+1}} \;|\; \ldots \;|\; \underset{a_{k-1}}{A_k}$$

With the fragment B,

$$B = \underset{a_{i+1}}{A_{i+1}} \;|\; \ldots \;|\; \underset{a_{j-1}}{A_j},$$

this can be written as

$$c = \underset{a_0}{A_0} \,|\, \ldots \;|\; \underset{a_{i-1}}{A_i} | \underset{x}{B} | \underset{x}{A_{j+1}} \;|\; \ldots \;|\; \underset{a_{k-1}}{A_k}.$$

This is zero because B is enclosed by the same tabstops (Axiom 11):

$$c = \underset{a_0}{A_0} \,|\, \ldots \;|\; \underset{a_{i-1}}{A_i} | \underset{x}{0} | \underset{x}{A_{j+1}} \;|\; \ldots \;|\; \underset{a_{k-1}}{A_k}.$$

According to Axiom 9 this means $c = 0$. □

Example 2. The chain

$$\underset{x}{A} | \underset{y}{B} | A$$

contains A twice but does not contain a tabstop twice. However, the layout specification is zero because the fragment $\underset{x}{A} | B$ can be duplicated using the Idempotence axiom (Axiom 3):

$$\underset{x}{A} | \underset{y}{B} | A * \underset{x}{A} | B.$$

Due to the Concatenation axiom (Axiom 6) this becomes

$$\underset{x}{A} | \underset{y}{B} | \underset{x}{A} | B,$$

which is zero because of Proposition 1. We capture this in another proposition.

Proposition 2. If there is a chain in a specification that contains a fragment twice, the layout specification is zero.

Proof. This proof only describes the horizontal case; the vertical case is identical. A chain that contains the non-zero fragment B twice can be written as:

$$c = \underset{a_0}{A_0} \,|\, \ldots \;|\; \underset{a_{i-1}}{A_i} | \underset{m\ n}{B} | \underset{a_{i+1}}{A_{i+1}} \;|\; \ldots \;|\; \underset{a_{j-1}}{A_j} | \underset{u\ v}{B} | \underset{a_{j+1}}{A_{j+1}} \;|\; \ldots \;|\; \underset{a_{k-1}}{A_k}.$$

From Axiom 12 it follows that $m = u$ and $n = v$, i.e.,

$$c = \underset{a_0}{A_0} \,|\, \ldots \;|\; \underset{a_{i-1}}{A_i} | \underset{m\ n}{B} | \underset{a_{i+1}}{A_{i+1}} \;|\; \ldots \;|\; \underset{a_{j-1}}{A_j} | \underset{m\ n}{B} | \underset{a_{j+1}}{A_{j+1}} \;|\; \ldots \;|\; \underset{a_{k-1}}{A_k}.$$

From Proposition 1 it follows that $c = 0$. □

Example 3. Fragments and tabstops can occur in multiple chains without the chains becoming 0. For example, with

$$c_0 = A_0 \underset{a_0}{|} \ldots \underset{a_{i-1}}{|} A_i \underset{x}{|} A_{i+1} \underset{a_{i+1}}{|} \ldots \underset{a_m}{|} A_m \tag{13}$$

$$c_1 = B_0 \underset{b_0}{|} \ldots \underset{b_{j-1}}{|} B_j \underset{x}{|} B_{j+1} \underset{b_{j+1}}{|} \ldots \underset{b_n}{|} B_n \tag{14}$$

$$c_2 = A_0 \underset{a_0}{|} \ldots \underset{a_{i-1}}{|} A_i \underset{x}{|} B_{j+1} \underset{b_{j+1}}{|} \ldots \underset{b_n}{|} B_n \tag{15}$$

$$c_3 = B_0 \underset{b_0}{|} \ldots \underset{b_{j-1}}{|} B_j \underset{x}{|} A_{i+1} \underset{a_{i+1}}{|} \ldots \underset{a_m}{|} A_m \tag{16}$$

if $c_0 \neq 0$, $c_1 \neq 0$, $c_2 \neq 0$, $c_3 \neq 0$, then $c_0 * c_1 * c_2 * c_3 \neq 0$.

In order to reason if layoutSpecs are equivalent, we introduce a layout specification normal form. To do so we first define the notion of a primitive tiling.

Definition 8 (Primitive Tiling)**.** A fragment p is called a *primitive tiling* if

$$p = A|B \quad or \quad p = A/B \quad or \quad p = A$$

with $A, B \in Areas$.

Definition 9 (Normal Form)**.** A layoutSpec $\hat{s}$ is in *normal form* if 1) there are are primitive tilings $p_0, \ldots, p_n$ such that $\hat{s} = p_0 * \cdots * p_n$, and 2) applying the tiling idempotence axiom (Axiom 4) to any primitive tiling of $\hat{s}$ or applying the parallelism axiom (Axiom 7) to any two primitive tilings of $\hat{s}$ does not yield any fragments that are not already in $\hat{s}$.

Example 4. Assuming $A, B, C, D \in Area$, the normal form of the grid layout in Example 1 is

$$s = A * B * C * D * A \underset{x}{|} B * A \underset{x}{|} D * C \underset{x}{|} D * C \underset{x}{|} B * A \underset{y}{/} C * A \underset{y}{/} D * B \underset{y}{/} C * B \underset{y}{/} D.$$

Proposition 3 (Normal Form Uniqueness)**.** Each layoutSpec s has a unique normal form $\hat{s}$ with $s = \hat{s}$ (up to duplicate primitive tilings and permutation of the primitive tilings in $p_0 * ... * p_n$).

Proof. First we show that each layoutSpec s has a normal form. The layoutSpec s consists of fragments. From the inductive definition of fragments (Definition 4) follows that each fragment can be broken down into primitive tilings. Then the tiling idempotence axiom (Axiom 4) and the parallelism axiom (Axiom 7) can be applied to transform the layoutSpec s to normal form $\hat{s}$.

To show that this normal form $\hat{s}$ is unique we assume there is a second normal form $s = \hat{s}'$ that, w.l.o.g, has one or more primitive tiles that are not present in $\hat{s}$. The only axioms that could generate these missing primitive tiles from existing primitive tiles in $\hat{s}$ are Axiom 4 and Axiom 7. However, from the definition of the normal form (Def. 9) these primitive tiles are already part of $\hat{s}$. This means that $\hat{s}'$ must differ from $\hat{s}$; $\hat{s}' \neq \hat{s} \neq s$ which contradicts the assumption. □

Knowing that every layoutSpec s has a unique normal form (Proposition 3) we can define the set of primitive tilings in s.

Definition 10 (*primitiveTilings*(s))**.** We define the set $primitiveTilings(s) = \{p_0, ..., p_n\}$, where $p_0, ..., p_n$ are primitive tilings such that $\hat{s} = p_0 * ... * p_n$ is the normal form of s. We say that a layoutSpec s is in a layoutSpec s' if $primitiveTilings(s) \subseteq primitiveTilings(s')$.

Definition 11 (*areas*(s))**.** The set of areas in a layoutSpec s is defined as: $areas(s) = primitiveTilings(s) \cap Area$.

Definition 12 (Algebraic Equivalence)**.** Two layoutSpecs s_0, s_1 are called algebraically equivalent, or equivalent for short, if $primitiveTilings(s_0) = primitiveTilings(s_1)$.

In other words, two algebraically equivalent layoutSpecs have the same normal form (up to duplicate primitive tilings and permutation of the primitive tilings in $p_0 * ... * p_n$).

Definition 13 (Outer Areas of a Fragment)**.** For a primitive tiling p with $p \in Area$, the sets of left, top, right and bottom outer areas of p are

$$LeftAreas(p) = TopAreas(p) = RightAreas(p) = BottomAreas(p) = \{p\}.$$

For a fragment B with $B \notin Area$, B can either be written as $B = C \underset{x}{|} D$ or as $B = C \underset{y}{/} D$. For $B = C \underset{x}{|} D$,

$$LeftAreas(B) = LeftAreas(C), \quad RightAreas(B) = RightAreas(D),$$

$$TopAreas(B) = TopAreas(C) \cup TopAreas(D), \quad BottomAreas(B) = BottomAreas(C) \cup BottomAreas(D),$$

and for $B = C \underset{y}{/} D$,

$$TopAreas(B) = TopAreas(C), \quad BottomAreas(B) = BottomAreas(D),$$

$$LeftAreas(B) = LeftAreas(C) \cup LeftAreas(D), \quad RightAreas(B) = RightAreas(C) \cup RightAreas(D).$$

Definition 14 (Outer Areas of a LayoutSpec)**.** The set of left outer areas of a layoutSpec s is defined as

$$LeftAreas(s) = \{p \in areas(s) \mid \text{there is no } p' \in Area \text{ such that } p'|p \in primitiveTilings(s)\}.$$

The set of right outer areas of a layoutSpec s is defined as

$$RightAreas(s) = \{p \in areas(s) \mid \text{there is no } p' \in Area \text{ such that } p|p' \in primitiveTilings(s)\}.$$

The vertical case is defined analogously.

Proposition 4 (Connectivity)**.** If layoutSpec s is non-zero then the following set *connectedAreasLeft* contains all areas in s:

$$connectedAreasLeft(s) = LeftAreas(s) \cup \{areas(L|C) \quad | \; L \in LeftAreas(s) \text{ and } L|C \text{ is a chain in } s\}.$$

The same is true for $RightAreas(s)$ and for the vertical case, that is, the set of all chains that start from the outer areas on one side (left, right, top or bottom) of a layoutSpec s contains all areas of s.

Proof. Let us assume there is an area A in s that is not in *connectedAreasLeft*. There must be $(L_1|A)$ with $L_1 \in areas(s)$ as A would otherwise be in $LeftAreas(s)$ and hence in *connectedAreasLeft*. Using the same argumentation we can recursively build a chain of areas $c = L_{n-1}| \ldots |L_1|A$ with $L_i \in areas(s)$. Because s is non-zero, c cannot contain an area twice (Proposition 2). Because $areas(s)$ is finite, there must be an area L_n with $c = L_n|(L_{n-1}| \ldots |L_1|A) = L_n|C$ and $L_n \in LeftAreas(s)$. Therefore, A is in *connectedAreasLeft*, which contradicts the assumption. □

5.1. Rectangular Layouts

In the previous layout examples, some connections between areas and tabstops were omitted. These are the connections to the outer tabstops of the layout. To describe these connections, the four implicit outer areas l, t, r, b as depicted in Figure 11 can be used. These four areas are "virtual" areas to describe the outer tabstops; they do not represent real objects in a GUI layout. For example, the full specification for a layout of two fragments A and B in a row is

$$l|\underset{x}{A}|B|r * t/\underset{x}{(A|B)}/b.$$

However, this specification is quite verbose and a convention is used to simplify the notation by omitting the outer areas l, t, r and b. Areas that have no further connection in a certain direction are implicitly connected to the outer areas in this direction. For example, the simplest layout specification s contains only a single area A: $s = A$. Since there are no fragments connected to any side of A, A is connected to all four outer areas. Another advantage of this convention is that a layout specification is very easy to extend. For example, when extending the layout A by tiling an item B to the right, the layout simply becomes $A|B$ and it becomes unnecessary to insert B between A and the outer area r. Using this convention, only rectangular layouts can be described. However, this is not a limitation when considering some areas as "empty spaces" as we discuss in Section 5.3.

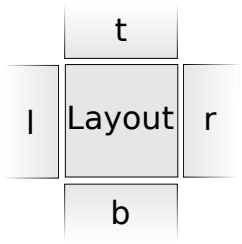

Figure 11. A layout is surrounded by four outer "virtual" areas.

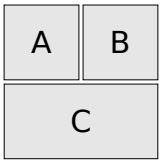

Figure 12. Layout with the specification $(A\underset{x}{|}B)\underset{y}{/}C$.

Example 5. The rectangular layout shown in Figure 12 can be completely specified as $(A\underset{x}{|}B)\underset{y}{/}C$.

Example 6. A more complex example that cannot be described using a single fragment is the pinwheel (Figure 13). One possible way to specify the pinwheel using multiple fragments is shown below:

$$(A \underset{x_1}{|} (B \underset{y_1}{/} E)) \underset{y_2}{/} D * (E \underset{y_2}{/} D) \underset{x_2}{|} C * B \underset{y_1}{/} C. \tag{17}$$

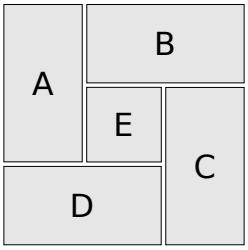

Figure 13. This entangled pinwheel layout cannot be specified as a single fragment.

5.2. Tabstop Index Convention

A tabstop has to be named only if it appears more than once. Otherwise, the tabstop index in the tiling operator can be simply omitted, e.g., $A\underset{x}{|}B \rightarrow A|B$. The tabstops without indices are all distinct. Such shorthands for distinct arguments are well-known. This is, for example, comparable to an underscore variable _ in the Prolog programming language. We will use this convention in the following.

Example 7. The layout specification of the grid layout in Figure 14 can be written as

$$(A\underset{x}{|}B)/(C\underset{x}{|}D).$$

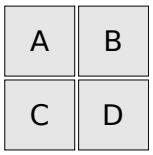

Figure 14. Grid layout.

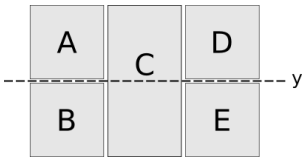

Figure 15. Layout with a horizontal tabstop crossing the middle area.

However, a layout consisting solely of the first row can be written without a tabstop index:

$$A|B.$$

Example 8. Using the index convention, the pinwheel layout from Example 6 can be written as

$$(A|(\underset{y_1}{B}/E))\ \underset{y_2}{/}D * (\underset{y_2}{E}/D)|C * \underset{y_1}{B}/C.$$

Example 9. The layout specification in Figure 15 can be written as

$$(\underset{y}{A}/B)|C|(\underset{y}{D}/E).$$

5.3. λ - Elements

A layout can contain regions that are not occupied by visible content such as widgets. An empty rectangular region is associated with a so-called λ-element. λ-elements are areas from the set *Area* which are introduced for three reasons. Firstly, λ-elements are needed to describe layouts containing empty space. Secondly, as described in Section 5.1, layouts always have a rectangular shape. λ-elements make it possible to describe non-rectangular layouts by filling the surplus space. Thirdly, a non-overlapping layout specification can be created by tiling all empty regions with λ-elements. Compared to other, non-empty areas, λ-elements have slightly special semantics (Section 6). In the following, we differentiate between areas that are λ-elements (empty spaces) and non-empty areas (such as widgets). λ-elements are usually adjacent to at least one non-empty area, as opposed to just other λ-elements.

6. Constraint Semantics

In this section, we show that tiling algebra can be represented using linear constraints. This means a layoutSpec can be translated into a constraint-based layout model such as the Auckland Layout Model (ALM) [2] (see Section 2.2). Using tiling algebra instead of a constraint system to describe a layout makes the layout specification more compact. For example, a simple horizontal linear layout with three layout items can be specified as $A|B|C$; using ALM this would already result in 18 constraints (six constraints per layout item: two minimum, two preferred and two maximum size constraints). In the following, we first define the constraint semantics of tabstops, areas, fragments, layoutSpecs, the $*$ operator and the tiling operators. Then we show that all tiling algebra axioms hold in the domain of constraints. The $[]$ operator denotes the constraint semantics of an algebraic term.

Definition 15 (Tabstop)**.** A tabstop (Def. 1) is a real-valued variable that represents the position of the tabstop in a layout, either on the x- or the y-axis.

Definition 16 (Area)**.** An area A (Def. 2) has two vertical tabstops A_{left}, A_{right} and two horizontal tabstops A_{top}, A_{bottom}. The semantics of an area A is the conjunction constraint

$$minConstraints \wedge prefConstraints \wedge maxConstraints$$

with $minConstraints$ being the hard constraints

$$A_{right} - A_{left} > minWidth \quad \wedge \quad A_{bottom} - A_{top} > minHeight,$$

$prefConstraints$ being the soft constraints

$$A_{right} - A_{left} = prefWidth \quad \wedge \quad A_{bottom} - A_{top} = prefHeight$$

and $maxConstraints$ being the soft constraints

$$A_{right} - A_{left} \leqslant maxWidth \quad \wedge \quad A_{bottom} - A_{top} \leqslant maxHeight$$

with $minWidth, minHeight, prefWidth, prefHeight, maxWidth, maxHeight$ being real numbers $\geqslant 0$.

Definition 17 (λ-Elements)**.** The semantics of a λ-element is the same as for an area with the differences that $minWidth = minHeight = 0$ and that a λ-element has no preferred and no maximum size constraints.

Note that the minimum constraints $minConstraints$ ensure a size greater than zero even though $minWidth = minHeight = 0$. This means the width and height of a λ-element can be infinitesimally small but must be greater than zero. Later we exploit this property when we show that Axiom 11 (Infeasibility) holds in the domain of constraints, i.e., that infeasible layoutSpecs lead to conflicting constraints.

Definition 18 (Fragment and LayoutSpec)**.** The semantics of a fragment and a layoutSpec (Def. 4 and Def. 5) is given as a logical conjunction of linear constraints. The semantics of the zero fragment is a constraint specification that has no solution

$$[0] =_{def} false.$$

Definition 19 ($*$ operators)**.** The semantics of the $*$ operator (Def. 6) is the logical conjunction.

$$[A * B] =_{def} [A] \wedge [B].$$

Definition 20 (Tiling Operators)**.** The semantics of the tiling operators are the following constraints:

$$[\underset{x}{A|B}] =_{def} \quad [A] \wedge [B] \wedge \Big(\bigwedge_{a \in RightAreas(A)} a_{right} = x \Big) \wedge \Big(\bigwedge_{b \in LeftAreas(B)} b_{left} = x \Big),$$

$$[\underset{y}{A/B}] =_{def} \quad [A] \wedge [B] \wedge \Big(\bigwedge_{a \in BottomAreas(A)} a_{bottom} = y \Big) \wedge \Big(\bigwedge_{b \in TopAreas(B)} b_{top} = y \Big).$$

Tiling means that all outer areas of the fragments on the side of the given tabstop are connected to that tabstop, e.g., $\underset{x}{A|B}$ means that the right of A and the left of B share the same tabstop. Note that the above assignments between tabstops (e.g., $a_{right} = x$) are equivalent to replacing the relevant border tabstops of the tiled areas with the tabstop they are tiled to (e.g., a_{right} by x; all constraints need to be updated accordingly). In our implementations of the algebra we are replacing the border tabstops as this results in fewer constraints overall.

In the following we prove that Axiom 1-12 are valid in the domain of constraints. We start with the axioms for the $*$ operator.

Associativity (Axiom 1). Logical conjunctions are associative: $[(A*B)*C] = ([A] \wedge [B]) \wedge [C] = [A] \wedge ([B] \wedge [C]) = [A * (B * C)]$. □

Commutativity (Axiom 2). Logical conjunctions are commutative: $[A * B] = [A] \wedge [B] = [B] \wedge [A] = [B * A]$. □

Idempotence (Axiom 3). Logical conjunctions are idempotent: $[A * A] = [A] \wedge [A] = [A]$. □

Tiling Idempotence (Axiom 4). The equations

$$[\underset{x}{A|B}] = [\underset{x}{A|B}] \wedge [A] \wedge [B] = [\underset{x}{A|B * A}] = [\underset{x}{A|B * B}]$$

$$[\underset{y}{A/B}] = [\underset{y}{A/B}] \wedge [A] \wedge [B] = [\underset{y}{A/B * A}] = [\underset{y}{A/B * B}]$$

follow directly from the constraint semantics of the tiling operators (Def. 20). □

Tiling Associativity (Axiom 5). We have to show that $[(A\underset{l}{|}B)\underset{m}{|}C] = [A\underset{l}{|}(B\underset{m}{|}C)]$:

$$
\begin{aligned}
[(A\underset{l}{|}B)\underset{m}{|}C] &= [A\underset{l}{|}B] \wedge [C] \wedge \Big(\bigwedge_{ab \in RightAreas(A|B)} ab_{right} = m \Big) \wedge \Big(\bigwedge_{c \in LeftAreas(C)} c_{left} = m \Big) && (18)\\
&= [A] \wedge [B] \wedge \Big(\bigwedge_{a \in RightAreas(A)} a_{right} = l \Big) \wedge \Big(\bigwedge_{b \in LeftAreas(B)} b_{left} = l \Big) \wedge \\
&\quad [C] \wedge \Big(\bigwedge_{ab \in RightAreas(A|B)} ab_{right} = m \Big) \wedge \Big(\bigwedge_{c \in LeftAreas(C)} c_{left} = m \Big) && (19)\\
&= [B] \wedge [C] \wedge \Big(\bigwedge_{b \in RightAreas(B)} b_{right} = m \Big) \wedge \Big(\bigwedge_{c \in LeftAreas(C)} c_{left} = m \Big) \wedge \\
&\quad [A] \wedge \Big(\bigwedge_{a \in RightAreas(A)} a_{right} = l \Big) \wedge \Big(\bigwedge_{bc \in LeftAreas(B|C)} bc_{left} = l \Big) && (20)\\
&= [A] \wedge [B\underset{m}{|}C] \wedge \Big(\bigwedge_{a \in RightAreas(A)} a_{right} = l \Big) \wedge \Big(\bigwedge_{bc \in LeftAreas(B|C)} bc_{left} = l \Big) && (21)\\
&= [A\underset{l}{|}(B\underset{m}{|}C)]. && (22)
\end{aligned}
$$

Equation 20 follows because $RightAreas(A|B) = RightAreas(B)$ and $LeftAreas(B) = LeftAreas(B|C)$. □

Concatenation (Axiom 6). We have to show that $[A\underset{l}{|}B * B\underset{m}{|}C] = [(A\underset{l}{|}B)\underset{m}{|}C]$:

$$
\begin{aligned}
[A\underset{l}{|}B * B\underset{m}{|}C] &= [A] \wedge [B] \wedge \Big(\bigwedge_{a \in RightAreas(A)} a_{right} = l \Big) \wedge \Big(\bigwedge_{b \in LeftAreas(B)} b_{left} = l \Big) \wedge \\
&\quad [B] \wedge [C] \wedge \Big(\bigwedge_{b \in RightAreas(B)} b_{right} = m \Big) \wedge \Big(\bigwedge_{c \in LeftAreas(C)} c_{left} = m \Big) && (23)\\
&= [A\underset{l}{|}B] \wedge [C] \wedge \Big(\bigwedge_{ab \in RightAreas(A|B)} ab_{right} = m \Big) \wedge \Big(\bigwedge_{c \in LeftAreas(C)} c_{left} = m \Big) && (24)\\
&= [(A\underset{l}{|}B)\underset{m}{|}C]. && (25)
\end{aligned}
$$

Equation 24 follows because $RightAreas(B) = RightAreas(A|B)$. □

Parallelism (Axiom 7). We have to show that $[A\underset{x}{|}B * C\underset{x}{|}D] = [A\underset{x}{|}D * C\underset{x}{|}B]$:

$$
\begin{aligned}
[A\underset{x}{|}B * C\underset{x}{|}D] &= [A] \wedge [B] \wedge \Big(\bigwedge_{a \in RightAreas(A)} a_{right} = x \Big) \wedge \Big(\bigwedge_{b \in LeftAreas(B)} b_{left} = x \Big) \wedge \\
&\quad [C] \wedge [D] \wedge \Big(\bigwedge_{c \in RightAreas(C)} c_{right} = x \Big) \wedge \Big(\bigwedge_{d \in LeftAreas(D)} d_{left} = x \Big) && (26)\\
&= [A\underset{x}{|}D * C\underset{x}{|}B]. && (27)
\end{aligned}
$$

The vertical case works analogously. □

Tiling Distributivity (Axiom 8). We have to show that $[(A\underset{x}{|}B)\underset{y}{/}C] = [A\underset{x}{|}B * A\underset{y}{/}C * B\underset{y}{/}C]$:

$$[(A\underset{x}{|}B)\underset{y}{/}C] = [A\underset{x}{|}B] \wedge [C] \wedge \Big(\bigwedge_{ab \in BottomAreas(A|B)} ab_{bottom} = y\Big) \wedge \Big(\bigwedge_{c \in TopAreas(C)} c_{top} = y\Big) \tag{28}$$

$$= [A] \wedge [B] \wedge \Big(\bigwedge_{a \in RightAreas(A)} a_{right} = x\Big) \wedge \Big(\bigwedge_{b \in LeftAreas(B)} b_{left} = x\Big) \wedge$$
$$[C] \wedge \Big(\bigwedge_{ab \in BottomAreas(A|B)} ab_{bottom} = y\Big) \wedge \Big(\bigwedge_{c \in TopAreas(C)} c_{top} = y\Big) \tag{29}$$

$$= [A] \wedge [B] \wedge \Big(\bigwedge_{a \in RightAreas(A)} a_{right} = x\Big) \wedge \Big(\bigwedge_{b \in LeftAreas(B)} b_{left} = x\Big) \wedge$$
$$[A] \wedge [C] \wedge \Big(\bigwedge_{a \in BottomAreas(A)} a_{bottom} = y\Big) \wedge \Big(\bigwedge_{c \in TopAreas(C)} c_{top} = y\Big) \wedge$$
$$[B] \wedge [C] \wedge \Big(\bigwedge_{b \in BottomAreas(B)} b_{bottom} = y\Big) \wedge \Big(\bigwedge_{c \in TopAreas(C)} c_{top} = y\Big) \tag{30}$$

$$= [A\underset{x}{|}B * A\underset{y}{/}C * B\underset{y}{/}C]. \tag{31}$$

Equation 30 follows because

$$\Big(\bigwedge_{ab \in BottomAreas(A|B)} ab_{bottom} = y\Big) \Leftrightarrow \Big(\bigwedge_{a \in BottomAreas(A)} a_{bottom} = y\Big) \wedge \Big(\bigwedge_{b \in BottomAreas(B)} b_{bottom} = y\Big).$$

The proofs for the other cases are analogous. □

Zero Fragment (Axiom 9). We have to show that $S\,\alpha\,0 = 0$ and $0\,\alpha\,S = 0$ with $\alpha \in \{|, /, *\}$. $[0]$ is defined as false, and hence is a multiplicative zero for the logical conjunction. Since all three operators are defined as logical conjunctions, one zero argument leads to the result being zero. □

Cancellation (Axiom 10). W.l.o.g. we only prove that | is left-cancellative. From the semantics of $A\underset{x}{|}B = A\underset{x}{|}C$ follows:

$$[A] \wedge [B] \wedge \Big(\bigwedge_{a \in RightAreas(A)} a_{right} = x\Big) \wedge \Big(\bigwedge_{b \in LeftAreas(B)} b_{left} = x\Big) \tag{32}$$

$$= [A] \wedge [C] \wedge \Big(\bigwedge_{a \in RightAreas(A)} a_{right} = x\Big) \wedge \Big(\bigwedge_{c \in LeftAreas(C)} c_{left} = x\Big) \tag{33}$$

$$\Rightarrow [B] \wedge \Big(\bigwedge_{b \in LeftAreas(B)} b_{left} = x\Big) = [C] \wedge \Big(\bigwedge_{c \in LeftAreas(C)} c_{left} = x\Big). \tag{34}$$

To prove that $[B] = [C]$ we have to show that

$$\Big(\bigwedge_{b \in LeftAreas(B)} b_{left} = x\Big) = \Big(\bigwedge_{c \in LeftAreas(C)} c_{left} = x\Big). \tag{35}$$

Assuming this is not true we can say, w.l.o.g., that there must be a constraint $b_{left,i} = x$ which is not in the right term of equation 35. To fulfill Equation 34 this missing constraint must be in $[C]$, i.e., there must be an area $c_i \in C$ that has $b_{left,i}$ either as a left or a right tabstop. However, we know from Proposition 4 that the set of chains starting from the areas $LeftAreas(C)$ contains all areas in C and because of $A\underset{x}{|}C$ there must be a chain $A\underset{x}{|}c_l|...|\underset{m}{c_i}|\underset{n}{...}$ with

$c_l \in LeftAreas(C)$. Since either m or n is the tabstop $b_{left,i} = x$ this results in $A\underset{x}{|}c_l|...|\underset{x}{...}$ which is zero because of Proposition 1. This contradicts the requirement of Axiom 10 that $A\underset{x}{|}C \neq 0$.

□

Infeasibility (Axiom 11). We have to show that $A\underset{x}{|}B\underset{x}{|}C$ is infeasible. If B is not an area we can write it either as $B = b_l|b_r$ or as $B = b_t/b_b$ with $b_l, b_r, b_t, b_b \in Fragment$. In the former case we get a chain $A\underset{x}{|}b_l|\underset{x}{b_r}|C$ and in the later case we get the chains $A\underset{x}{|}b_t|C$ and $A\underset{x}{|}b_b|C$. Recursively following this argumentation we eventually get chains in the form $A\underset{x}{|}b_0\,|\underset{t_0}{b_1}\,|\underset{t_1}{\ ...\ }|\underset{t_{n-2}}{b_{n-1}}|\underset{t_{n-1}}{b_n}|\underset{x}{C}$ with $b_i \in Area$. This is also true if $B \in Area$. This leads to the minimum width constraint system:

$$t_0 - A_{right} > minWidth_0 \ , \ t_1 - t_0 > minWidth_1 \ , \ ... \ , \ t_{n-1} - t_{n-2} > minWidth_{n-1} \ , \ C_{left} - t_{n-1} > minWidth_n$$

Summing up all these constraints yields that $C_{left} - A_{right} = x - x = 0 > \sum_i minWidth_i$. Since all $minWidth_i$ are non-negative this is an infeasible constraint. □

Tabstop Uniqueness (Axiom 12). We only prove that from $[C\underset{l}{|}A * B\underset{n}{|}A]$ follows $l = n$. The proofs for the other cases are analogous.

$$\begin{aligned}[C\underset{l}{|}A * B\underset{n}{|}A] \quad = \quad & [C] \wedge [A] \wedge \Big(\bigwedge_{c \in RightAreas(C)} c_{right} = l \Big) \wedge \Big(\bigwedge_{a \in LeftAreas(A)} a_{left} = l \Big) \wedge \\ & [B] \wedge [A] \wedge \Big(\bigwedge_{c \in RightAreas(B)} b_{right} = n \Big) \wedge \Big(\bigwedge_{a \in LeftAreas(A)} a_{left} = n \Big). \qquad (36)\end{aligned}$$

(37)

Therefore

$$\bigwedge_{a \in LeftAreas(A)} a_{left} = l = n$$

so all fragments tiled to the left of A are tiled to the same tabstop. □

7. Layout Edit Operations

So far the algebra only describes static layout specifications. In this section, a small set of edit operations is introduced and we show how these operations can be used to modify a layout specification in a sound way. This means a layout can be edited while leaving the layout in a solvable and non-overlapping state.

The overall idea for editing an algebraic layout specification is to have a layout that is densely tiled with non-empty areas (layout items) and λ-elements (empty spaces). A new area can then be inserted into a layout by replacing an existing λ-element with the new area, and removing items works analogously (Section 7.3). However, because in general a suitable λ-element for an insertion is not available, i.e., a λ-element with the right connections, operations are needed to transform λ-elements into the desired shape and size. This can be achieved by splitting and merging λ-elements (Section 7.1). Furthermore, edit operations are needed which extend or shrink a layout by adding or removing outer λ-elements (Section 7.2). A complex sound layout can be generated by extending and modifying an already sound layout, e.g., a layout that only contains one area. After presenting the edit operations, the layout soundness requirements are defined (Section 7.4) and it is shown how these edit operations can be applied in a sound way (Section 7.5).

7.1. Splitting and Merging λ-Elements

In the following we describe how λ-elements can be split and merged. Furthermore, the extending and shrinking of a layout by a λ-element is discussed. For brevity only the horizontal cases are described; the vertical cases work analogously.

Definition 21 (Splitting)**.** The split operation splits a λ-element λ_0 at a tagstop $\underset{x}{|}$ into two new λ-elements λ_1, λ_2 by substituting λ_0 by $\lambda_1 \underset{x}{|} \lambda_2$:

$$\lambda_0 = \lambda_1 \underset{x}{|} \lambda_2.$$

Definition 22 (Merging)**.** The merge operation merges two tiled λ-elements $\lambda_1 \underset{x}{|} \lambda_2$ into a new λ-element λ_0 by substituting $\lambda_1 \underset{x}{|} \lambda_2$ by λ_0:

$$\lambda_1 \underset{x}{|} \lambda_2 = \lambda_0.$$

7.2. Tiling, Extending and Shrinking LayoutSpecs

So far it is only possible to tile fragments. To describe the tiling of complex layoutSpecs such as two pinwheels we define how layoutSpecs can be tiled. We only define the horizontal case; the vertical case is defined analogously.

Definition 23 (Tiling LayoutSpecs)**.** Horizontal tiling of two layoutSpecs s_0 and s_1 is defined as

$$s_0 \underset{x}{|} s_1 = s_0 * s_1 * \underset{(R,L) \in RightAreas(s_0) \times LeftAreas(s_1)}{\circledast} R \underset{x}{|} L.$$

In other words, every right area of s_0 is tiled with every left area of s_1. Analogously, vertical tiling of two layoutSpecs s_0 and s_1 is defined as

$$s_0 \underset{y}{/} s_1 = s_0 * s_1 * \underset{(B,T) \in BottomAreas(s_0) \times TopAreas(s_1)}{\circledast} B \underset{y}{/} T.$$

We will use the following two notions to modify (extend and shrink) a layoutSpec.

Definition 24 (Extending LayoutSpecs)**.** We call the tiling of a layoutSpec with a λ-element extending a layout.

Definition 25 (Shrinking Chains)**.** Omitting a λ-element λ_s either at the beginning of a chain (shrinking on the left side) or at the end of a chain (shrinking on the right side), is called shrinking λ_s from a chain. The vertical case is defined analogously.

For example, shrinking the chain $\lambda_s|(A/B)|C$ results in $(A/B)|C$.

7.3. Removing and Inserting

The following two operations modify a layoutSpec by removing an area from or inserting an area into an existing layoutSpec.

Definition 26 (Removing an Area)**.** To remove an area from a layoutSpec, the area is replaced by a new λ-element; i.e., the λ-element is substituted for all occurrences of the area in the layoutSpec.

Definition 26 ensures that a layout is always completely filled, either with areas or with λ-elements, and is thus overlap-free. Note that we cannot remove whole fragments at once as they may not be rectangular. Furthermore, a new fragment can be inserted into a layoutSpec by replacing an existing λ-element with the new fragment. To specify where the fragment should be inserted, all four surrounding tabstops have to be named. Alternatively, four surrounding areas can be given to specify the insertion position, as in the insertion operator in the following Def. 27. The insertion can be applied if there is a λ-element with the same adjacent areas as specified in the insertion operator. Note that Axiom 7 (Parallelism) may be applied to find such a fitting λ-element. In case a fragment should be inserted adjacent to a layout border, no area needs to be specified in this direction.

Definition 27 (Inserting a Fragment)**.** An insertion of a new fragment X into a λ-element can be written as ${}_L\overset{T}{\underset{B}{X}}{}_R$ where X is inserted so that $L|X|R$ and $T/X/B$. If any of L, T, R, B are omitted they default to the respective implicit outer areas l, t, r, b.

Sometimes only a part of a λ-element should be used for the insertion of a fragment. For example, it should be possible to insert a fragment just into a corner of a λ-element. It is merely important that at least one horizontal and one vertical tabstop of the new fragment is connected to a border of the λ-element so that its position is well-defined. To connect a fragment to a side or a corner of a λ-element, the λ-element can be replaced by a fragment that has the desired properties. In Def. 28 we extend our notation for insertion so it can be specified which borders of the inserted fragment are not directly connected to the borders of the λ-element. The corresponding indices are marked by an asterisk $*$.

Definition 28 (Loose Insertion)**.** An insertion of a new fragment X into the top-left corner of a λ-element can be written as ${}_L\overset{T}{\underset{B*}{X}}{}_R*$, meaning that the λ-element is replaced by $(X|\lambda_0)/\lambda_1$, with λ_0 and λ_1 being two new λ-elements. Similarly, an insertion of a new fragment X into the left side of a λ-element can be written as ${}_L\overset{T}{\underset{B}{X}}{}_R*$, meaning that the λ-element is replaced by $(X|\lambda_0)$. Insertions into all other corners and sides are analogous. Indices with a $*$ are called *loose insertions*, and normal indices are called *tight insertions*.

Example 10. In the following example we show how the layout specification

$$s = \underset{x}{(A|B)}/\underset{x}{(C|\lambda|D)}$$

can be created using the edit operations:

1. Starting with the initial layoutSpec $s = A$, s is extended on the right side: $s = \underset{x}{A|\lambda_0}$.
2. λ_0 is replaced by B by inserting ${}_A\overset{t}{\underset{b}{B}}{}_r$, with t, r, b being the virtual outer areas (Section 5.1). The resulting layoutSpec is then extended to the bottom: $s = \underset{x}{(A|B)}/\lambda_1$.
3. λ_1 is split at the tabstop x: $s = \underset{x}{(A|B)}/\underset{x}{(\lambda_2|\lambda_3)}$.
4. C is loosely inserted with ${}_l\overset{A}{\underset{b}{C}}{}_B*$: $s = \underset{x}{(A|B)}/\underset{x}{(C|\lambda|\lambda_3)}$, with l being the left outer area.
5. D is inserted with ${}_A\overset{A}{\underset{b}{D}}{}_r$: $s = \underset{x}{(A|B)}/\underset{x}{(C|\lambda|D)}$.

Note that this is only one of many possible ways to create this layoutSpec. As discussed later, more complex operations such as moving and resizing can be realized by combining removing, reorganizing of λ-elements and inserting of areas.

7.4. Soundness Requirements

There are two soundness requirements for the algebraic edit operations. Firstly, a layout specification must be solvable. Secondly, a layout specification must be non-overlapping.

Definition 29 (Solvable Layout Specification)**.** A layoutSpec is called solvable if it is non-zero.

Proposition 5 (Solvable Layouts)**.** In the domain of constraints, the constraint system for a layoutSpec is solvable if and only if the layoutSpec is non-zero.

Proof. In Section 6 we already showed that a fragment that contains the same tabstop twice leads to conflicting constraints (Tabstop Uniqueness). It remains to be shown that a non-zero layoutSpec s has a solvable constraints system. From the definition of an area (Def. 16) we can make two observations: 1) Minimum size constraints are the only hard constraints. 2) Horizontal and vertical constraints are decoupled since they do not share variables. For this

reason we only need to show that all minimum width constraints (Def. 16) in the constraint system lead to a solvable constraint system; the vertical case works analogously.

Proposition 4 states that all chains starting from the areas in $LeftAreas(s)$ contain all areas in s. From this, one can see that a layoutSpec can be written as a directed graph with areas as vertices and tabstops as edges. If the directed graph would have a cycle there would be chain that contains a fragment twice, e.g., $A|_i B|_i A|_i B$. However, because of Proposition 2 this cycle would be 0. The contrapositive conclusion is that a non-zero layoutSpec s always translates to a directed acyclic graph (DAG). From graph theory we know that a DAG has a topological ordering which also means that we can find an ordering for the tabstops in s.

We can directly give a solution for the constraint system of s by defining the positions of the ordered tabstops as following:

$$t_0 = 0$$

$$t_n = t_{n-1} + 2 \cdot max(minWidth)$$

with $max(minWidth)$ the largest minimum width of all minimum width constraints. This solution satisfies all minimum width constraints since for each $A \in areas(s)$, $A_{right} - A_{left} > max(minWidth)$. □

Definition 30 (Non-Overlapping Layout Specification)**.** A layoutSpec s is called *non-overlapping* if for any two areas $A, B \in areas(s)$ a horizontal or a vertical chain c exists which contains A and B.

Example 11. The layout fragment

$$s = A|\lambda_0|B$$

is non-overlapping because for all pairs of areas a horizontal chain exists that contains both areas. These are $A|\lambda_0$ for the pair (A, λ_0), $\lambda_0|B$ for (B, λ_0), and $A|\lambda_0|B$ for (A, B). However, the layout specification

$$s = (A|B)/C * (A|B)/D$$

is overlapping as there is no chain that contains both C and D.

Proposition 6 (Non-Overlapping Layouts)**.** In the domain of constraints, a non-overlapping layoutSpec leads to a non-overlapping layout.

Proof. Given any two areas A, B, from the requirement in Def. 30 the two areas are either in a horizontal or vertical chain. W.l.o.g. we assume they are in a horizontal chain and A is on the left hand of B. The minimum size constraint system of this chain has the constraints:

$$t_0 - A_{right} > minWidth_A \;\;,\;\; t_1 - t_0 > minWidth_1 \;\;,\;\; \ldots \;\;,\;\; t_n - t_{n-1} > minWidth_n \;\;,\;\; B_{left} - t_n > minWidth_B$$

Summing up all these constraint yields that $B_{left} - A_{right} > 0$ which means A and B do not overlap. □

Proposition 7 (Non-Overlapping Fragments)**.** A fragment is non-overlapping.

Proof. For a fragment f that is an area the statement is trivially true. From the definition of fragments (Definition 4) follows that a fragment f which is not an area can be written either as $f = A|B$ or as $f = A/B$ with $A, B \in Fragment$. For the horizontal case $f = A|B$ we have to show that there is a horizontal chain $A_i|..|B_j$ for each pair $A_i, B_j \in Area$ with A_i in A and B_j in B. We only show the horizontal case; the vertical case is analogous.

We first show that there is a horizontal chain $A_i|...|B$ for every A_i. For $A_i = A$ the statement is trivially true. If $A \notin Area$, A can be written either as $A = A_l|A_r$ or as $A = A_t/A_b$, and A_i lies within one of these sub-fragments. Thus, $A|B = A_l|A_r|B$ or $A|B = (A_t/A_b)|B = A_t|B * A_b|B * A_t/A_b$. This means that in any case there is a horizontal chain between B and A_l, A_r, or B and A_t, A_b, respectively. We now assume w.l.o.g. A_i lies in A_l which results in the horizontal chain $A_l|..|B$. We expressed A by a horizontal chain containing a smaller fragment A_l. By induction it follows that there exist a horizontal chain $A_i|...|B$. Analogously, it can be proven that there is a chain $A|...|B_j$ for every B_j in B. Thus, there is a horizontal chain $A_i|..|B_j$ for each pair A_i, B_j. □

7.5. Soundness

In this section we show how the edit operations defined in Sections 7.1–7.3 can be applied so that they leave a sound layoutSpec sound. Assuming a set of sound layout operations, the soundness of all layout specifications can be shown by induction. A single layout item is naturally sound. Furthermore, editing an initially sound layout specification using sound layout operations results in a new sound layout specification. In the following we show that the operations Def. 23 (Tiling LayoutSpecs), Def. 24 (Extending), Def. 26 (Removing) and Def. 28 (Insertion) are sound and that the operations Def. 21 (Splitting), Def. 22 (Merging) and Def. 25 (Shrinking Chains) are sound under additional conditions. We only show the horizontal case; the vertical case follows analogously.

Proposition 8. Tiling and extending layoutSpecs (Def. 23, Def. 24) are sound operations.

Proof. If tiling two layoutSpec is a sound edit operation it follows directly that extending a layoutSpec, i.e., tiling a layoutSpec and a λ-element, is a sound operation.

Assuming that the layoutSpecs s_0, s_1 are sound we have to show that the layoutSpec $s_0 \underset{x}{|} s_1$ is sound. Solvable: Since all elements of $RightAreas(s_0)$ and $LeftAreas(s_1)$ by definition had no fragment tiled to the right and left side, respectively, there cannot be a conflicting chain with $c = A \underset{x}{|} B \underset{x}{|} C$ with A, B, C in $s_0 \cup s_1$. Non-overlapping: Since s_0, s_1 are non-overlapping, we only have to show that for each area L in s_0 and each area R in s_1 there is either a horizontal or a vertical chain that contains L and R. We know from Proposition 4 that the set of all chains that start from outer areas contain all areas of a layoutSpec. Thus, for every L and R it is always possible to find a horizontal chain $c = ...\underset{x}{|}L\underset{x}{|}...\underset{x}{|}R\underset{x}{|}....$ □

Proposition 9. Removal and Insertion (Definitions. 26, 28) are sound operations.

Proof. Replacing an area is trivially sound because it does not modify the topology of the layout specification. Also the loose insertion operation keeps the specification sound as it makes sure that gaps are tightly filled with λ-elements. □

Proposition 10 (Splitting)**.** Splitting a λ-element λ_0 at a connection $A \underset{x}{|} B$ (Def. 21) is sound if there is no chain $c = \lambda_0 \underset{x}{|} ... \underset{x}{|} B$ or $c = A \underset{x}{|} ... \underset{x}{|} \lambda_0$.

Proof. Solvable: When splitting λ_0 at the tabstop x, the only new connection that is introduced is $\lambda_1 \underset{x}{|} \lambda_2$. This can cause a zero specification only if there are chains of the form $c = \lambda_1 \underset{x}{|} \lambda_2 \underset{x}{|} ... \underset{x}{|} B$ or $c = A \underset{x}{|} ... \underset{x}{|} \lambda_1 \underset{x}{|} \lambda_2$. However, the precondition of splitting forbids such chains $c = \lambda_0 \underset{x}{|} ... \underset{x}{|} B$ or $c = A \underset{x}{|} ... \underset{x}{|} \lambda_0$. Non-overlapping: Because λ_0 did not overlap any other layout item, also the fragment $\lambda_1 \underset{x}{|} \lambda_2$ does not, as it fills the same area. Furthermore, the new fragment $\lambda_1 \underset{x}{|} \lambda_2$ is in itself non-overlapping. □

Proposition 11 (Merging)**.** Merging a fragment $\lambda_1|\lambda_2$ into a λ-element λ_0 (Def. 22) is sound if there exist $A, B \in Fragment$ with $A/(\lambda_1|\lambda_2)/B$, and $C, D \in Fragment$ with $C|(\lambda_1|\lambda_2)|D$ and there are no other fragments in the specification containing λ_1 or λ_2. Note that any other fragments containing λ_1 or λ_2 can usually be eliminated by using Axiom 7 for parallel fragments.

Proof. Solvable: Replacing the fragment $\lambda_1 \underset{x}{|} \lambda_2$ by λ_0 does not add any existing tabstops, and therefore cannot create any chains that contain the same tabstop twice. Non-overlapping: From the requirements it follow that the two fragments $A/(\lambda_1|\lambda_2)/B$ and $C|(\lambda_1|\lambda_2)|D$ become $A/\lambda_0/B$ and $C|\lambda_0|D$ respectively. This means that all chains that show non-overlap for $\lambda_1|\lambda_2$ are also valid for λ_0. No other chains are needed to prove that the specification is non-overlapping. □

Proposition 12 (Shrinking LayoutSpecs)**.** If a λ-element λ_s is the only element in $LeftAreas(s)$ or $RightAreas(s)$ and there is no vertical chain containing λ_s, shrinking λ_s from all chains containing λ_s (Def. 25) is a sound operation.

Proof. Since λ_s is in *LeftAreas*(s) or *RightAreas*(s), λ_s is either at the start or the end of a chain and thus shrinking λ_s is always possible. After the operation, all chains are still non-zero as no tabstops were added. Furthermore, if the fragments on the left (or right) of λ_0 were non-overlapping, then they are also non-overlapping after removing λ_0 as the chains for each pair of the remaining areas are still intact. □

8. Layout Editing Process

After specifying the layout edit operations, all necessary tools for complex layout editing are at hand. However, there are various challenges when mapping a user operation to one or more algebra operations. λ-elements can be transformed in multiple ways in order to find a suitable configuration for an operation. Choosing a desired configuration for an edit operation depends on how the user actually sees the layout, e.g., where an area should be inserted may depend on how a layout specification looks like when it is solved.

8.1. Equivalence Classes

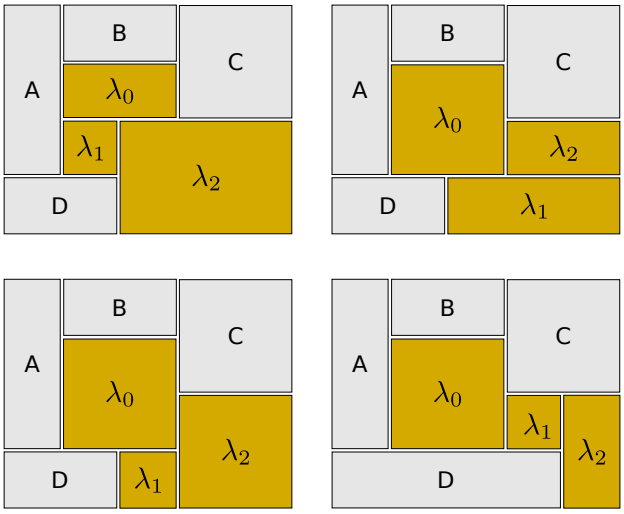

Figure 16. In general there are multiple ways to tile empty space with λ-elements. While the layout specification of the areas stays $s = A|B|C*A/D$ the λ-specification changes.

In general, there is no fitting λ-element available for an insertion operation and there are multiple ways the empty space in a layout can be tiled. For example, Figure 16 shows a layout specification with various options to tile the λ-elements. For this reason we define the set *primitiveλTilings*(s) and a λ-specification:

Definition 31. The set *primitiveλTilings*(s) is defined as a subset of *primitiveTilings*(s) containing all *primitiveTilings* of s that contain at least one λ-element. Given the set *primitiveλTilings*$(s) = \{l_0, \ldots, l_n\}$, the specification $s_l = l_0 * \cdots * l_n$ is called the *λ-specification* of s.

For example, for the specification $s = (A|\lambda)/B$: *primitiveλTilings*$(s) = \{A|\lambda, \lambda/B, \lambda\}$ and the λ-specification is $s_\lambda = A|\lambda * \lambda/B * \lambda$. As there is generally a set of different possible λ-specifications to describe the empty space in a layout, these possibilities define an *equivalence class* of possible specifications for a layout. This reflects that the connections of areas to other areas stay constant for all λ-specifications of the equivalence class (see Figure 16).

To transform between different λ-specifications within an equivalence class, split and merge operations can be used. For example, the L-shaped configuration of λ-elements in Figure 17 can be transformed by splitting the longer λ-element and merging the resulting corner piece with the shorter λ-element.

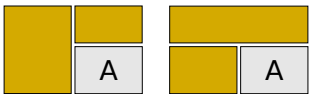

Figure 17. There are different ways to tile an L-shaped empty area.

8.2. Layout Specifications and Concrete Layouts

To describe further how the user can manipulate a layout, one has to differentiate between layout specifications, i.e., layoutSpecs, and concrete layouts. A layout specification describes how areas are connected relative to other areas (Section 5). The connections specified in the layout specification with the intrinsic sizes of the areas (i.e., the intrinsic sizes of layout items such as widgets) and size constraints for the overall layout form a complete constraint system, which is used to determine a layout as it is rendered on the screen. By solving this constraint system the actual size of the layout items can be calculated. The solved layout, as it is rendered on a screen, is called a *concrete layout*. A layout specification can have an unlimited number of concrete layouts. This is because it can be displayed at different layout sizes. Moreover, the intrinsic area sizes can change the appearance of the layout significantly.

Example 12. Figure 18 shows two different concrete layouts that have the same layout specification. The only difference between the two concrete layouts is that the preferred intrinsic area sizes differ.

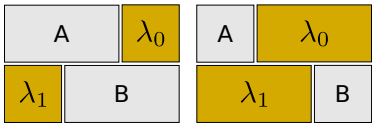

Figure 18. Two concrete layouts with the same layout specification $s = (A|\lambda_0)/(\lambda_1|B)$. On the left, areas A and B have a relatively large intrinsic preferred width. On the right, their preferred widths are relatively small.

The difference between layout specification and concrete layouts also has consequences for the operations the user is able to perform. In the example on the right of Figure 18, the user is able to insert a new item that separates the items A and B horizontally, e.g., after changing the λ-specifications:

$$(A \underset{y}{/} \lambda_3)|C|(\lambda_2 \underset{y}{/} B).$$

However, in the concrete layout on the left of Figure 18, where the preferred sizes of the areas are larger, this is not possible. While the operation is still possible in the layout specification, the user would not be able to trigger this operation in the concrete layout.

A similar problem occurs when looking for the right member of the equivalence class for a desired insertion. Figure 16 depicts such a case. Assume the initial concrete layout is the one in the top-left corner and the user wants to insert a new item into the center of the layout. There are several possible λ-specifications, e.g., the other three shown in Figure 16. While for all three λ-specifications λ_0 is at the right position, the resize behavior and even the topology vary. For example, the λ-specification at the bottom-right forces the right of D to be on the right side of λ_0, which may be an undesirable consequence if the user merely wants to insert an item into λ_0.

8.3. User Interaction on Layout Specifications

As discussed above, changing a layout specification may not have the desired effect in the concrete layout. A way to handle this problem is to take the concrete layout into account when modifying the layout specification. The layout specification is used by the edit operations to ensure soundness for all possible concrete layouts. To summarize: 1) A concrete layout may be considered to decide what operations should be applied to the layout specification. 2) The layout specification can be used to decide if an operation always leads to sound concrete layouts.

Note that in practice one has to consider numerical problems in the concrete layout when choosing what operations should be applied to the layout specification. For example, if two tabstops l and r are constrained to $l \leqslant r$ and are solved to $l \approx r$ it is possible that l is actually greater r within the numerical error in the concrete layout. From this concrete layout one may consider to insert a conflicting area: $...\underset{r}{|}A\underset{l}{|}....$ However, the sound edit operations will prevent such an operation.

8.4. Completeness

Using the tiling algebra as described above allows one to specify a large set of layouts. To be more specific, all non-overlapping orthogonal tilings can be described. Using λ-elements makes it possible to describe gaps between areas. The algebraic description can capture every possible connection between the areas. This can be done by combining primitive tilings (Def. 8) with the $*$ operator.

Adding areas successively to a layout specification using the edit operations described here is an intuitive way to build a concrete layout. However, it is not yet clear whether this technique is powerful enough to generate all non-overlapping orthogonally tiled concrete layouts. Example 12 illustrates a possible problem using this approach. To show completeness, i.e., that all concrete layouts can be created using the edit operations, we present the following algorithm, which builds a layoutSpec for an arbitrary concrete layout.

1. Determine the number of x and y tabstops n_x and n_y of the given layout.
2. Starting from a layoutSpec $s = \lambda_0$, extend s n_x times to the right. This results in a row with $n_x + 1$ λ-elements.
3. Extend s to the bottom by λ_k and recursively split λ_k at each x tabstop of the top row. Perform this step n_y times, resulting in a $(n_x + 1) \times (n_y + 1)$ grid structure of λ-elements.
4. Merge these λ-elements as needed to reassemble the concrete layout topology. This is possible because s has as many tabstops as needed for the concrete layout.
5. Finally, replace λ-elements with desired areas to yield the layout specification for the concrete layout.

As mentioned in the previous section, in practice numerical problems may need to be considered when determining the tabstop order from a concrete layout.

9. Applications

In this section we discuss how the proposed algebra is applied in two real-world applications, Stack & Tile and ALE, which were described in Sections 3 and 4. We describe how the application-specific edit operations are mapped to corresponding algebra operations and how this helps to achieve soundness and ease of use. We also show how special requirements of such applications can be supported within the proposed algebraic framework. There are many other applications with similar properties, e.g., for document layout, web layout, graphics design, CAD, and user interface customization. It is likely that the algebra could be applied to them with similar benefits.

9.1. Algebra for Stack & Tile

Stack & Tile allows users to create layouts comprised of windows ("Stack & Tile groups"). By using the proposed algebra as a basis, Stack & Tile inherits its capability to create a large range of useful layouts. We use the algebra to describe the tiling functionality of Stack & Tile. The stacking operation is not described by the algebra as it neither causes problems with soundness nor influences tiling. An algebraic description would not provide much added value. However, tiling can potentially cause soundness issues such as overlap and infeasibility and is most important for the range of layouts that can be created.

In Stack & Tile windows are represented by areas of the tiling algebra. To define Stack & Tile groups we first define *layout item paths*. Intuitively, on a layout item path, each pair of neighboring areas shares at least one horizontal and one vertical connection.

Definition 32 (Layout Item Path)**.** A layout item path p is a set of areas that fulfills the following properties. Two non-empty areas A and B belong to the same path p if there is one of the following fragments in the specification:

$$(A|B)/C \quad or \quad C/(A|B) \quad or \quad (A/B)|C \quad or \quad C|(A/B).$$

If A, B are on a path p and B, C are on the same path p, this implies that also A, C are on path p.

Note that in the first part of the definition, C could also be one of the implicit outer layout areas (Section 5.1) in which case C does not belong to the layout item path. It is used to make sure that A and B are aligned to at least one common horizontal and one common vertical tabstop, in a way that A and B are clearly touching on one side (i.e., not just corner to corner).

Example 13. In the specification $(A|_x B)/(C|_x \lambda)$, A, B and C are on the same layout item path. However, in the specification $(\lambda|_x B)/(C|_x \lambda)$, B and C are not on the same layout item path.

Definition 33 (Stack & Tile group)**.** A Stack & Tile group is a layoutSpec that contains exactly one layout item path.

There are two operations in Stack & Tile that need to be considered: inserting and removing of a window to or from a Stack & Tile group (Section 3.1). The algebra makes it very easy to define and implement these operations in a sound manner, as the Stack & Tile tiling operation can directly be mapped to the algebra insert operation and the Stack & Tile removing operation to the algebra removing operation described in Section 7.3. However, the insert operation has the restriction that the resulting layoutSpec must stay a Stack & Tile group, i.e., the new window must be added to the existing layout item path.

When removing a window from a Stack & Tile group, the resulting layoutSpec may not fulfill the definition of an algebraic Stack & Tile group anymore: the layoutSpec may be broken into multiple layout item paths. If this happens, the layoutSpec is split by moving each layout item path in a separate layoutSpec (Section 3.1). A simple approach to create a non-overlapping layoutSpec for each layout item path is to copy the original layoutSpec and replace all areas that do not belong to the layout item path with λ-elements. In general, this leads to a layout that is surrounded by unnecessary λ-elements. However, these superfluous λ-elements can be transformed as needed and then removed by shrinking the layout (Section 7).

9.2. Algebra for ALE

Similar to Stack & Tile, ALE aims to enable users to create a large range of GUI layouts. However, as a GUI builder, ALE needs to support much more fine-grained layout editing with more layout edit operations (Section 4.1) and more potential soundness issues. In the following, we specify GUI layouts as algebraic layoutSpecs, and widgets are atomic terms, i.e., areas. While inserting and removing of widgets can be handled with the corresponding algebraic operations, four other operations have to be discussed: moving, swapping, resizing and inserting between an existing tabstop and an existing widget. All these operations can be expressed with the proposed algebra in a fairly straightforward manner.

Moving a widget in ALE can be described by an algebraic remove and an insert operation: first, the widget in question is removed, then the resulting layout can be used to perform the insert operation. If the operation is interrupted in the GUI builder, the original layout specification is restored.

Swapping widgets simply means the corresponding elements in the algebraic specification have to be swapped.

Resizing a widget can also be described using an algebraic remove and an insert operation. When inserting the widget, only the connections at the resized borders of the widget (either one or two) change. Resizing a widget to its preferred size (by detaching a widget from a tabstop) can be done by first removing the widget and then performing a *loose insertion* of the same widget (Section 7.3). In the following a detach operation on the right widget side is described; the other sides can be described analogously. The loose insertion operator for the right detach of a widget X is

$$\begin{matrix} & T & \\ L & X & R* \\ & B & \end{matrix}.$$

Here, L, T, and B are the widgets the detached widget was originally connected to. Now a suitable widget R has to be found. To do so we search for the widget R that gives the largest empty area encompassed by L, T, R and B in the concrete layout (note that this cannot be done in the layoutSpec). Applying this loose insertion results in a new λ-element λ_{new} between the detached widget X and R.

Inserting a widget between an existing tabstop and an existing widget can be achieved by first detaching the existing widget from the tabstop, as described for the resize operation. This results in a new λ-element λ_{new}. Then, the new widget is inserted into λ_{new}.

9.2.1. Filling Gaps

As discussed in Section 4.1.1, remove, move and resize operations can cause gaps between widgets, so that a widget is not directly or indirectly connected to a horizontal or vertical layout border anymore. For example, a group of widgets can become "stranded" in the middle of the layout, only surrounded by λ-elements. To fill the gap, this floating widget group that is not connected anymore is moved into the direction of the edited widget.

To describe this problem algebraically, first, a definition of what a direct or indirect connection to a layout border means is needed. In the following, an indirect or direct connection is just called a *connection* to a layout border.

Definition 34 (Connection to a Layout Border)**.** A fragment A is connected to a layout border if there exist a chain that contains A, at least one layout border area and no λ-elements.

If a group of widgets is not connected to a layout border horizontally or vertically, a λ-specification has to be found that has a λ-element λ_c that is the only separating item either to another connected widget or to the layout border in this direction. By eliminating this λ_c, the gap can be filled. Such an elimination operation can be defined as follows.

Definition 35 (λ-Elimination)**.** Given a layoutSpec s with a fragment $A\underset{k}{|}\lambda_c\underset{l}{|}B$ then the elimination operation removes λ_c from s and replaces the tabstops k and l with a new tabstop m, i.e., $A\underset{m}{|}B$.

Proposition 13 (λ-Elimination Soundness)**.** λ-Elimination is a sound operation if s does not contain any other chain $c = A\underset{k}{|}...\underset{l}{|}B$ connecting A and B.

Proof. Solvable: Because there is no chain $c = A\underset{k}{|}...\underset{l}{|}B$ other than $A\underset{k}{|}\lambda_c\underset{l}{|}B$ before the elimination, there is also no fragment $c = A\underset{m}{|}...\underset{m}{|}B$ after the elimination. Non-overlapping: All fragments that were previously connected by the chain $A\underset{k}{|}\lambda_c\underset{l}{|}B$ are still connected by $A\underset{m}{|}B$ after the elimination, so the specification stays non-overlapping. □

10. Conclusion

We have presented an algebra for tiled layout items with the aim of making it easier to describe layout specifications and systems that modify these specifications in a sound manner. The algebra can describe arbitrary non-overlapping orthogonal tilings. By introducing vertical and horizontal tiling operators and defining rules to combine these operators, it is possible to describe layouts using the fairly intuitive notation of fragments. This makes it easier to associate a concrete layout with its abstract layout specification compared to a low-level notation based on linear constraints. The tiling operations, if seen as binary operators, are isomorphic cancellative semigroup operators with involution. A third operator, $*$, is isomorphic to a Boolean conjunction.

The algebra provides edit operations for modifying layout specifications that are sound, i.e., the operations keep a layout solvable and non-overlapping. Soundness is one of the challenges when developing layout editing systems, and using the proposed algebra as a basis for such systems helps to address this challenge. To fulfill the requirement of non-overlapping layouts, the algebra introduces λ-elements to represent empty areas with non-negative size. By tiling the whole empty space of a layout with λ-elements, the layout specification becomes non-overlapping. In general, there are many ways to tile the empty space, and we have defined λ-operations to transform between these different tilings in a sound manner. A new area can be inserted into the layout by replacing a λ-element, and similarly, an area can be removed by replacing it with a λ-element.

In two case studies, we used the algebra in the development of two real-world applications with layout editing functionality: a window manager that allows users to tile windows into groups (Stack & Tile), and a GUI builder (ALE). These applications are representative of a wide range of applications with layout editing functionality, which face similar challenges with regard to soundness, flexibility, and usability. We show that the edit operations of Stack & Tile and ALE can be defined based on the algebra with little effort so that they inherit the soundness and flexibility of their algebraic counterparts. Furthermore, specific extensions that are necessary for these systems can be expressed within the proposed algebraic framework.

There are various limitations of the tiling algebra when it comes to describing more general user interface properties. Currently, the tiling algebra can only describe tiled areas. General layout constraints, e.g., a constraint that

ensures that two areas have the same size, cannot be described in the algebra. For GUIs, tiling algebra can only describe the layout topology. For example, logical group affiliation of widgets or their functionality are not specified.

As future work, one can consider layout edit operations that affect a group of layout items. For example, the insertion of a comb-like structure into an existing layout specification. The tiling algebra has already been implemented as a generic library and is used in ALE to ensure sound edit operations. In the future, also Stack & Tile could implement the algebra for its edit operations.

References

[1] G. J. Badros, A. Borning, P. J. Stuckey, The Cassowary linear arithmetic constraint solving algorithm, ACM Transactions on Computer-Human Interaction 8 (4) (2001) 267–306.

[2] C. Lutteroth, R. Strandh, G. Weber, Domain specific high-level constraints for user interface layout, Constraints 13 (3).

[3] C. Zeidler, J. Müller, C. Lutteroth, G. Weber, Comparing the usability of grid-bag and constraint-based layouts, in: Proc. 24th Australian Computer-Human Interaction Conference, OzCHI '12, ACM, 2012, pp. 674–682.

[4] C. Zeidler, C. Lutteroth, W. Sturzlinger, G. Weber, The Auckland Layout Editor: An improved GUI layout specification process, in: Proc. UIST'13, ACM, 2013, pp. 343–352.

[5] C. Zeidler, C. Lutteroth, G. Weber, An evaluation of stacking and tiling features within the traditional desktop metaphor, in: Human-Computer Interaction INTERACT 2013, Springer, 2013, pp. 702–719.

[6] M. Wirsing, Algebraic specification languages: An overview, in: Recent Trends in Data Type Specification, Vol. 906 of Lecture Notes in Computer Science, Springer, 1995, pp. 81–115.

[7] E. Astesiano, M. Bidoit, H. Kirchner, B. Krieg-Brückner, P. D. Mosses, D. Sannella, A. Tarlecki, CASL: the Common Algebraic Specification Language, Theoretical Computer Science 286 (2) (2002) 153–196.

[8] F. Paternò, C. Santoro, One model, many interfaces, in: Computer-Aided Design of User Interfaces III, Springer Netherlands, 2002, pp. 143–154.

[9] M. F. Ali, M. A. Pérez-Quiñones, M. Abrams, E. Shell, Building multi-platform user interfaces with UIML, in: Computer-Aided Design of User Interfaces III, Springer Netherlands, 2002, pp. 255–266.

[10] Q. Limbourg, J. Vanderdonckt, B. Michotte, L. Bouillon, V. López-Jaquero, USIXML: A language supporting multi-path development of user interfaces, in: Engineering Human Computer Interaction and Interactive Systems, Vol. 3425 of Lecture Notes in Computer Science, Springer, 2005, pp. 200–220.

[11] K. Luyten, K. Coninx, An XML-based runtime user interface description language for mobile computing devices, in: Interactive Systems: Design, Specification, and Verification, Vol. 2220 of Lecture Notes in Computer Science, Springer, 2001, pp. 1–15.

[12] F. Newbery, An interface description language for graph editors, in: Visual Languages, 1988., IEEE Workshop on, 1988, pp. 144–149.

[13] H. Thimbleby, User interface design with matrix algebra, ACM Trans. Comput.-Hum. Interact. 11 (2) (2004) 181–236.

[14] R. Bardohl, GenGEd - a generic graphical editor for visual languages based on algebraic graph grammars, in: Proceedings of the IEEE Symposium on Visual Languages, VL '98, IEEE Computer Society, 1998, pp. 48–55.

[15] H. T. Bruns, M. J. Egenhofer, User interfaces for map algebra, Journal of the Urban and Regional Information Systems Association 9 (1) (1997) 44–54.

[16] A. Gupta, S. Santini, Toward feature algebras in visual databases: The case for a histogram algebra, in: Advances in Visual Information Management, Vol. 40 of IFIP The International Federation for Information Processing, Springer US, 2000, pp. 177–196.

[17] E. Jungert, S. Chang, An image algebra for pictorial data manipulation, CVGIP: Image Understanding (2) (1993) 147 – 160.

[18] J. F. Allen, Maintaining knowledge about temporal intervals, Commun. ACM 26 (11) (1983) 832–843.

[19] H. W. Guesgen, Spatial reasoning based on Allen's temporal logic, Tech. rep., International Computer Science Institute (1989).

[20] P. Balbiani, J.-F. Condotta, L. F. n. del Cerro, A tractable subclass of the block algebra: Constraint propagation and preconvex relations, in: Progress in Artificial Intelligence, Vol. 1695, Springer, 1999, pp. 75–89.

[21] A. Lahoti, R. Singh, A. Mukerjee, Multi-dimensional interval algebra with symmetry for describing block layouts, in: Selected Papers from the Third International Workshop on Graphics Recognition, Recent Advances, GREC '99, Springer-Verlag, 2000, pp. 143–154.

[22] W. Graf, Constraint-based graphical layout of multimodal presentations, in: Advanced Visual Interfaces, 1992, pp. 356–387.

[23] K. Marriott, P. J. Stuckey, Programming with constraints: an introduction, MIT press, 1998.

[24] B. Vander Zanden, B. A. Myers, D. A. Giuse, P. Szekely, Integrating pointer variables into one-way constraint models, ACM Trans. Comput.-Hum. Interact. 1 (2) (1994) 161–213.

[25] G. Weber, A reduction of grid-bag layout to Auckland layout, in: Australasian Software Engineering Conference (ASWEC), 2010, pp. 67 –74.

[26] R. Fletcher, Practical methods of optimization; (2nd ed.), Wiley-Interscience, 1987, Ch. 10.3.

[27] C. Zeidler, C. Lutteroth, G. Weber, Constraint solving for beautiful user interfaces: how solving strategies support layout aesthetics, in: Proc. 13th International Conference of the NZ Chapter of the ACM's Special Interest Group on HCI, CHINZ '12, ACM, 2012, pp. 72–79.

[28] N. Jamil, J. Mueller, C. Lutteroth, G. Weber, Extending linear relaxation for user interface layout, in: Tools with Artificial Intelligence (ICTAI), 2012 IEEE 24th International Conference on, Vol. 1, 2012, pp. 939–946.

[29] N. Jamil, D. Needell, J. Muller, C. Lutteroth, G. Weber, Kaczmarz algorithm with soft constraints for user interface layout, in: Tools with Artificial Intelligence (ICTAI), 2013 IEEE 25th International Conference on, IEEE, 2013, pp. 818–824.

[30] E. Kandogan, B. Shneiderman, Elastic windows: evaluation of multi-window operations, in: Proc. SIGCHI conference on Human factors in computing systems, 1997, pp. 250–257.

[31] H. Ehrig, B. Mahr, Fundamentals of Algebraic Specification 1: Equations and Initial Semantics, Monographs in Theoretical Computer Science. An EATCS Series, Springer Berlin Heidelberg, 2012.