SCIENTIFIC PAPERS, UNIVERSITY OF LATVIA, 2012. Vol. 787
COMPUTER SCIENCE AND INFORMATION TECHNOLOGIES 106–124 P.

Calculating The Layout For Dialog Windows Specified As Models

Sergejs Kozlovics

University of Latvia, Faculty of Computing
Raiņa blvd. 19, LV-1586, Rīga, Latvia
Institute of Mathematics and Computer Science, University of Latvia
Raiņa blvd. 29, LV-1459, Rīga, Latvia

sergejs.kozlovics@lumii.lv

Abstract In model-driven engineering, dialog windows can be described as models conforming to Dialog Metamodel. To make dialog models simpler, Dialog Metamodel does not require to specify exact coordinates and dimensions of graphical dialog elements. To transform a dialog model to a running dialog window a solution for laying out dialog elements is needed. We present an algorithm that calculates the layout by means of quadratic optimization.

Keywords: dialog metamodel, dialog engine, graphical user interface, GUI, layout, quadratic optimization.

1 Introduction

Graphical User Interface (GUI) is an essential part of many software products. There are numerous libraries for creating GUI dialogs and forms such as VCL[1], Windows::Forms[2], wxWidgets[3], GTK+[4], QT[5], Java Swing[6], etc.

[1] Visual Component Library. Available in Delphi and C++ Builder (currently maintained by CodeGear; previously maintained by Borland followed by Inprise) .

[2] Developed by Microsoft for the .NET framework. The open-source cross-platform .NET implementation Mono tries to re-implement Windows::Forms for various platforms.

[3] An open-source cross platform library providing "a truly native look and feel" for the applications.

[4] An open-source library used by the GNOME desktop environment in Linux.

[5] An open-source library used by the KDE desktop environment in Linux.

[6] A de facto GUI standard in Java. Developed by Sun, currently maintained by Oracle.

While GUI elements (buttons, input fields, check boxes, etc.) provided by these libraries are common, the libraries themselves are based on different conventions and have different APIs (Application Programming Interfaces). When a graphical dialog window is created directly using such a library, the developers need to know the API very well. Moreover, they have either to provide concrete coordinates for GUI elements (which may require certain calculations), or to specify their relative layout by library-specific facilities (e.g., by means of Java layout managers) requiring extra knowledge and skills. To help the developers, GUI designers were invented. With a GUI designer (such as Swing Designer, Qt Designer, etc.) dialog windows can be specified graphically just in a few clicks, and the GUI designer will generate the code for the corresponding GUI library.

Still, the benefits provided by GUI designers are not applicable to all cases. If the content of a dialog window (not only data, but also GUI elements) is computed at runtime, a GUI designer can be used to create only a carcass of a window; the content has to be filled up by means of the corresponding library API. Thus, the developer still needs a deep understanding of the GUI library and has to perform a non-trivial job of laying out GUI elements at runtime, especially when a complex window is being generated. Moreover, the traditional approach "generate GUI code→compile→run" is undesirable since the compiler may not be available on a user PC, and the generation process followed by launching a compiler or an interpreter may take a while.

In this paper we present a solution for creating graphical dialog windows automatically at runtime. The characteristic features of the proposed solution are:

- *The initial specification of a window is represented as a model.* We may think of the model as of abstract syntax of the sketch of a window on a sheet of paper. Some benefits of using a model are: independence on a particular GUI library or a particular programming language; the ability to create (generate) a model at runtime; the ability to specify dialog windows using concepts similar to concepts used in the sketch on a paper; support for model-driven development.
- *Developers need to specify only essential layout information in a model.* For instance, if the sizes of an input field and a button, as well as the spacing between them, have not been specified, these values will be chosen automatically in a way that contributes to a nice look of the resulting window.
- *A model is automatically transformed to a running dialog window.* The coordinates and unspecified dimensions of GUI elements are calculated automatically. When the window is being resized, dimensions and positions of elements will adapt to the window, preserving a nice look when possible.

In 2010 we have already published a metamodel[7] for specifying dialog windows [1]. In Section 2 we provide a brief summary of that Dialog Metamodel. The rest of the paper concentrates on the process of transforming a given dialog model to a running dialog window. We call the module that performs that process *Dialog Engine*[8]. In Section 3 we list certain agreements on layout information specified in a model and describe how Dialog Engine chooses the values for unspecified dimensions and positions of GUI elements. These automatically chosen values become a part of the given dialog model. In Section 4 we show how Dialog Engine transforms this (adjusted) dialog model to an input of the quadratic optimization solver. The solver returns the coordinates of GUI elements, and Dialog Engine displays the window. Section 5 is devoted to related work, and Section 6 concludes the paper.

2 A Glance At Dialog Metamodel

Fig. 1 depicts Dialog Metamodel (we published this metamodel in *Acta Universitatis Latviensis* in 2010 [1]). Each GUI element is either a separate component (such as a button, an input field, etc.; see class *Component* and its subclasses on the left in Fig. 1), or a container (a component containing other components; see class *Container*). Containers and components logically form a tree structure (see the composition between *Component* and *Container* in Fig. 1): each container has an ordered list of components lying within it. The layout of child components is determined by the type of the container. We distinguish 13 container types encircled by the rounded rectangle on the right in Fig. 1. Table 1 explains these container types. These are invisible containers used to construct the structure of a dialog window. The detailed explanation of other (visible) containers as well as of particular GUI components can be found in the paper mentioned above [1].

The initial dialog window (represented by the class *Form*) is a vertical box by default, but it can be transformed to a container of any other type (say, *HorizontalBox*) by adding the corresponding container as the only child, and putting all other components inside that child.

Dialog Metamodel contains also special classes called *events* (rounded rectangles in Fig. 1) and *commands* (ellipses in Fig. 1). These classes are used to ensure communication between Dialog Engine and other modules (usually, model transformations; hereinafter we refer to these modules as transformations). Transformations create commands for Dialog Engine, and Dialog Engine creates events that can be handled by transformations. Events and commands are temporary objects — they are deleted just after the desired action has been performed.

[7] A metamodel is a language for specifying models.
[8] That is why Dialog Metamodel is also called Dialog Engine Metamodel.

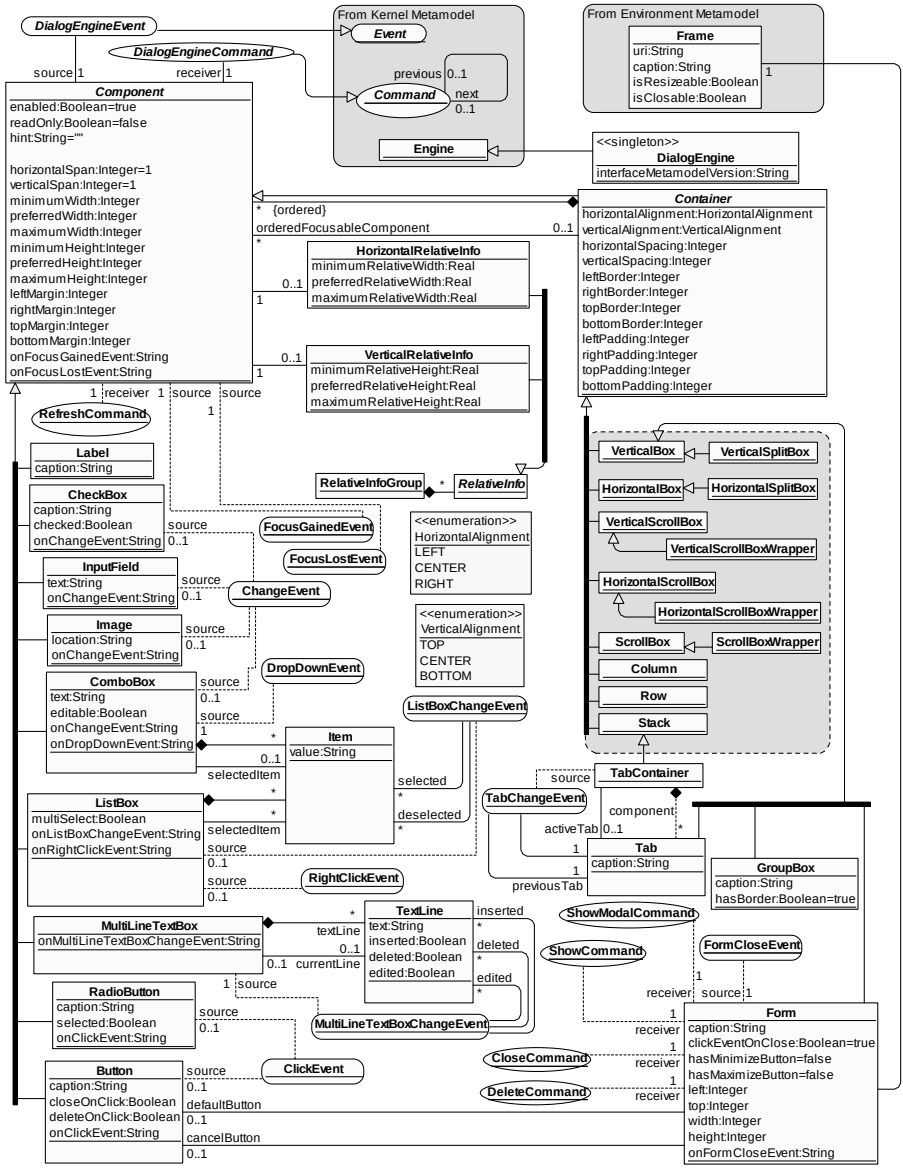

Fig. 1: Dialog Metamodel.

Table 1: Laying out child components in different container types.

Container Type	**The layout of child components**
VerticalBox	Vertical, components are laid one beyond another.
VerticalSplitBox	Components are laid one beyond another, but visible splitters are inserted between components. Each splitter can be moved by the user to enlarge a component on one side and to lessen a component on the other side.
HorizonalBox	Horizontal, components are laid one after another.
HorizontalSplitBox	Similar to *VerticalSplitBox*, but components are laid horizontally, and splitters are vertical lines.
VerticalScrollBox	Similar to *HorizontalBox*, but with a vertical scrollbar. Components are laid out horizontally one after another until the border of a scrollbar is reached. The remaining components continue at the next "row".
VerticalScrollBoxWrapper	Can be used to add a vertical scroll bar for a component without scrollbars. Technically, the same as *VerticalScrollBox*, but the default values for layout information differ to provide better look.
HorizontalScrollBox	Elements are laid out like columns in a newspaper. "Columns" can be scrolled horizontally by means of a horizontal scrollbar.
HorizontalScrollBoxWrapper	Can be used to add a horizontal scroll bar for a component without scrollbars.
ScrollBox	The same as *VerticalBox*, but one or two scrollbars may appear, when the visible part of this scroll box is not able to accommodate all the children. Can be turned into a *HorizontalBox* by adding a *HorizontalBox* as a child.
ScrollBoxWrapper	Can be used to scroll horizontally and vertically a component without scrollbars.
Column	Similar to *VerticalBox*, but components inside two neighbouring columns are aligned to form a table-like structure (have the same y-coordinates).
Row	Similar to *HorizontalBox*, but components inside two neighbouring rows are aligned (have the same x-coordinates).
Stack	Components are laid out like cards, occupying the same space on a dialog window. Used to implement tabs (see *TabContainer*).

When a model transformation needs to show a dialog window for a given dialog model, it creates either a *ShowModalCommand*, or a *ShowCommand* instance depending on whether the dialog needs to be modal or modeless. The control is passed to Dialog Engine, which displays the corresponding window on the screen. Execution of a *ShowModalCommand* finishes only after the window is closed. On the contrary, *ShowCommand* returns immediately after displaying the window, which remains visible while the transformation continues.

When a dialog window is active, certain events (usually, corresponding to user clicks and keystrokes) can occur. On each such event, Dialog Engine creates some *Event* instance. In Dialog Metamodel (Fig. 1), attributes starting with "on" (e.g., *onClickEvent*) are used to specify model transformations that Dialog Engine will call on the corresponding events. After the transformation processes the event, the event instance is deleted. When processing events, transformations can issue new commands to Dialog Engine, e.g., a command to show another dialog, or a *RefreshCommand* to refresh an already running dialog or its part (a subtree rooted at the given component) when some changes have been made to the corresponding dialog model.[9]

On certain events (e.g., when processing a *ClickEvent* of some "Close" button, or on a *FormCloseEvent*, which occurs when the user clicks the "X" button), a transformation can issue a *CloseCommand* to ask Dialog Engine to close an active dialog window. If the dialog is modal, *CloseCommand* also indicates that execution of the corresponding *ShowModalCommand* must finish.

After closing the dialog window, a *DeleteCommand* can be used to delete a dialog model that is not needed any more.

In this paper we focus at layout information common to all GUI components, even to components that may be added later. This layout information is found in classes *Component* (excluding the first three and the last two attributes), *HorizontalRelativeInfo*, *VerticalRelativeInfo*, and *Container*. The next section explains what do these attributes mean[10] and how to choose the unspecified values of these attributes.

3 Specifying The Layout Of GUI Elements

The *horizontalSpan* and *verticalSpan* attributes of the Component class are used in rows and columns, where a component needs to span multiple cells. Without these two attributes, it could be impossible to lay out containers as depicted in Fig. 2(a). Fig. 2(b) shows how this structure can be specified with the appropriate values of *horizontalSpan* and *verticalSpan*. The default value for both these attributes is 1 meaning that a component occupies a single cell, when placed in a row or column container.

[9] Thus, a call stack similar to function call stack can occur.

[10] We repeat some relevant information published earlier [1].

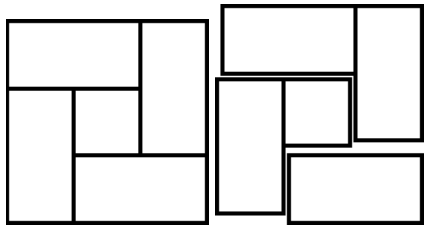

Fig. 2: (a) An example of five containers that cannot be laid out using horizontal and vertical boxes only. (b) The arrangement of the same five containers using rows. The first row contains two components: the first one spans two columns (*horizontalSpan*=2, *verticalSpan*=1), and the second one spans two rows (*horizontalSpan*=1, *verticalSpan*=2). The first component of the second row spans two rows (*horizontalSpan*=1, *verticalSpan*=2); neither rows nor columns are spanned by the second component (*horizontalSpan*=1, *verticalSpan*=1). The third row has only one component that spans two columns (*horizontalSpan*=2, *verticalSpan*=1).

The *minimumWidth* and *minimumHeight* attributes specify strict lower bounds (in pixels) for the width and height, respectively. The *preferredWidth* and *preferredHeight* attributes specify the preferred width and height. These values are not strict: when preferred dimensions of a component conflict with other, more important, constraints, deviations from the preferred dimensions are allowed. The *maximumWidth* and *maximumHeight* attributes specify the upper bound for the dimensions. These constraints are considered strict, but with the following exception: when other strict constraints are unsatisfiable, maximum width and height are allowed to increase by a minimal value to satisfy those constraints. The reason for introducing non-strict constraints relies on the following principle: if all the constraints are unsatisfiable, it is better to show a dialog window that violates some non-important layout constraints than to throw an exception and leave the user without the dialog window at all.

> *Example.* If a button has *minimumWidth* set to 100, but *maximumWidth* set to 0, the actual width of the button will be 100, i.e., the maximum width will increase by 100 pixels — the minimal value that satisfies the *minimumWidth* constraint. In this case, the *preferredWidth* value does not matter, since the width is determined by more strict minimum and maximum constraints.

The meaning of attributes for specifying margins, borders, spacings, and paddings (from classes *Component* and *Container*) is depicted in Fig. 3 (only horizontal values are depicted; vertical values have similar meaning).

The semantics of Dialog Metamodel states that there exists *gravity* between each container and its child components. That means that children edges tend to "stick" to the corresponding parent edges. However, if a child component

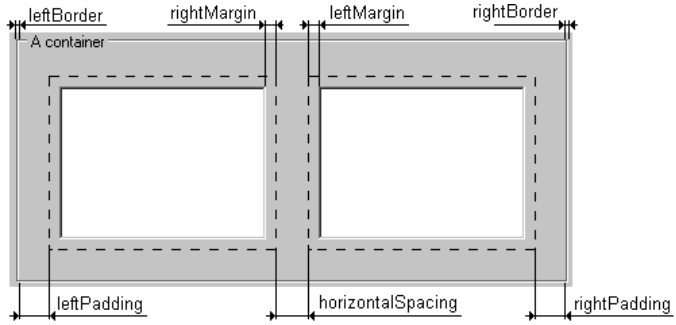

Fig. 3: An example illustrating what do values for margins, borders, paddings and spacings mean.

reaches its *maximumWidth* or *maximumHeight* constraint, the gravity does not work any more. In this case we have to specify how such children have to be put within the parent. The *horizontalAlignment* and *verticalAlignment* attributes of the *Container* class come to aid here. These values specify the horizontal alignment (left, center, or right) as well as vertical alignment (top, center, or bottom) for child components when they cannot not be stretched (by means of "gravity") any more.

Not only absolute, but also relative dimensions are supported by Dialog Metamodel. Relative widths and heights that relate to each other are grouped (see class *RelativeInfoGroup*). For example, to specify that widths of some three components have ratio 2:3:4, we attach three *HorizontalRelativeInfo* instances, one to each component, and set the values of *preferredRelativeWidth* to 2, 3, and 4, respectively. Finally, we attach all the three *HorizontalRelativeInfo* instances to a *RelativeInfoGroup* instance. Note that there is no need to specify more than one *HorizontalRelativeInfo* and more than one *VerticalRelativeInfo* for each component: if a particular width or height of some component appears in several groups, then these groups depend on each other and may be replaced by a single group by adjusting the ratio.

The minimum and maximum relative widths and heights are useful to control relative dimensions of components, when the dialog is being resized. An example is given in Fig. 4. The button 1 is not resizeable, and the preferred width ratio for both buttons is 1:1. If the user resizes the form, and the preferred ratio 1:1 could not be met, the button 2 is allowed to be up to two times wider (*maximumRelativeWidth*=2) or shorter (*minimumRelativeWidth*=0.5) than the button 1.

Like preferred absolute dimensions, preferred relative dimensions are not strict. However, minimum and maximum relative sizes are strict, but they are used in cooperation with the corresponding absolute sizes: having both relative and absolute minimum and maximum bounds, we can make absolute bounds

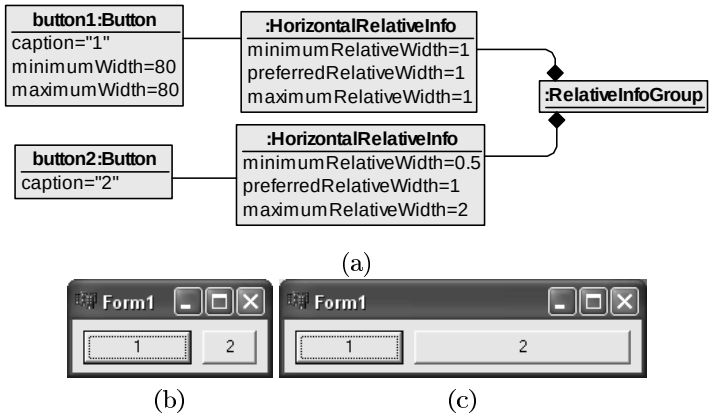

Fig. 4: An instance (a) demonstrating the usage of minimum and maximum relative sizes. The minimum (b) and the maximum (c) sizes of button 2.

more strict and forget about the relative bounds (see the next section for details).

Specifying all the constraints and sizes from above forces the developer to think more on layout than on the content of a dialog window. Thus, we allow the developer to leave all values unspecified, while preserving the possibility to specify important values (or all, if all are important). A question arises: how to choose values, if they have not been specified? The answer depends on a particular component or container. Table 2 lists some examples. The principles shown in Table 2 can be applied, when choosing values for other GUI elements.

4 Using Quadratic Optimization To Obtain Coordinates

This section explains how, given a dialog instance specified according to Dialog Metamodel, the quadratic optimization can be used to lay out GUI components in that dialog window.

4.1 The QMDC and the Extended QMDC Problems

The problem of quadratic minimization subject to difference constraints (QMDC) is as follows. Given n variables $x_0, x_2, \ldots, x_{n-1}$, minimize the quadratic function

$$\sum_{0 \le i < n} a_i x_i^2 + \sum_{0 \le i < j < n} b_{ij} x_i x_j + \sum_{0 \le i < n} c_i x_i$$

subject to difference constraints

$$x_i - x_j \ge d_{ij}, \text{ where } 0 \le i, j < n.$$

Table 2: How Dialog Engine chooses values for unspecified layout attributes.

Component or container	**Values**
Button (a non-resizeable component)	minimumWidth=preferredWidth=maximumWidth =TextWidth(caption)+c_1; minimumHeight=preferredHeight=maximumHeight =TextHeight(caption)+c_2; all margins=0; for all buttons within the same container: all relative sizes set to 1 (this ensures that buttons within the same container have equal sizes regardless their labels)
InputField (horizontally resizeable, but vertically non-resizable)	minimumWidth=c_3; preferredWidth=c_4; maximumWidth=∞; minimumHeight=preferredHeight=maximumHeight =TextHeight(text)+c_5; all margins=0

HorizontalBox (resizeable container with no visible borders and zero paddings; horizontal layout of children)	minimumWidth=minimumHeight=0;
	maximumWidth=maximumHeight=∞;
	all margins=0;
	horizontalAlignment=verticalAlignment=CENTER;
	horizontalSpacing=c_6; verticalSpacing is not applicable;
	all borders=0;
	all paddings=0;

GroupBox (a visible container with borders and padding; vertical layout of children)	minimumWidth=TextWidth(caption)+c_7;
	minimumHeight=TextHeight(caption)+c_8;
	maximumWidth=maximumHeight=∞
	all margins=0;
	horizontalAlignment=verticalAlignment=CENTER;
	horizontalSpacing is not applicable; verticalSpacing=c_9;
	leftBorder=c_{10}; rightBorder=c_{11};
	topBorder=TextHeight(caption)+c_{12}; bottomBorder=c_{13};
	leftPadding=c_{14}; rightPadding=c_{15}; topPadding=c_{16};
	bottomPadding=c_{17};

Note 1. If preferred size for a container has not been specified, Dialog Engine is allowed to leave that value unspecified; in this case this preferred value will not be considered by quadratic optimization, and the preferred size of the container will be determined by inner components. For non-containers (final components such as a button or an input field) preferred sizes still have to be chosen by Dialog Engine.

Note 2. Constants c_i depend on the GUI library.

Note 3. The infinity constant∞ is the absolute maximum size for any GUI component (e.g., 10 000 pixels). If some size reaches ∞, the component becomes so huge that it is unreasonable to show it to the user, so Dialog Engine throws an exception.

If also the constraints in the form

$$x_i - x_j \geq d_{ij} \geq m_{ij}, \tag{1}$$

are allowed, then we get the Extended QMDC problem (EQMDC). Here d_{ij} are the desired values and m_{ij} are the minimum values. In case the constraints taking into a consideration only d_{ij} are unsatisfiable, one or more of d_{ij} values may be decreased preserving $d_{ij} \geq m_{ij}$, i.e., d_{ij} cannot be decreased by more than by $d_{ij} - m_{ij}$.

4.2 The application of EQMDC

This section explains how to transform a dialog instance to the input of the EQMDC problem. The EQMDC solver is described in Section 4.7.

The variables we need are as follows. For each component C four variables are introduced to specify the left, right, top and bottom coordinates: x^C_{L}, x^C_{R}, y^C_{T} and y^C_{B}. The variables that bound the component with its margins are $x^C_{\mathrm{L}M}$, $x^C_{\mathrm{R}M}$, $y^C_{\mathrm{T}M}$ and $y^C_{\mathrm{B}M}$. If C is a container, then the variables for storing C bounds without its border are $x^C_{\mathrm{L}B}$, $x^C_{\mathrm{R}B}$, $y^C_{\mathrm{T}B}$ and $y^C_{\mathrm{B}B}$. Finally, the variables for bounding the inner area of C (without padding) are $x^C_{\mathrm{L}P}$, $x^C_{\mathrm{R}P}$, $y^C_{\mathrm{T}P}$, $y^C_{\mathrm{B}P}$.

The following subsections explain which constraints and terms to be minimized are introduced for 1) absolute sizes; 2) relative sizes; 3) margins, borders, padding and spacing, and 4) gravity and alignment. We provide the constraints for the x-dimension only since the constraints for the y-dimension are similar.

4.3 Constraints and minimization terms for absolute sizes

The constraint concerning minimum width, obviously, is:

$$x^C_{\mathrm{R}} - x^C_{\mathrm{L}} \geq \mathit{minimumWidth};$$

But the constraint for the maximum width is being written in the extended form:

$$\begin{aligned} x^C_{\mathrm{L}} - x^C_{\mathrm{R}} &\geq -\mathit{maximumWidth} \\ &\geq -\infty. \end{aligned}$$

This is the same as

$$\begin{aligned} x^C_{\mathrm{R}} - x^C_{\mathrm{L}} &\leq \mathit{maximumWidth} \\ &\leq \infty, \end{aligned}$$

but written in the form of (1).

The preferred sizes are specified not by means of constraints, but as terms of the function to be minimized. We add to the minimization the term

$$c \cdot \left((x^C_{\mathrm{R}} - x^C_{\mathrm{L}}) - \mathit{preferredWidth} \right)^2.$$

Here the penalty for the actual width $(x^C_{\mathrm{R}} - x^C_{\mathrm{L}})$ to be distinct from the preferred width grows quadratically. This ensures that even the component cannot have

the preferred size (due to constraints), the actual size will be close to the preferred. The constant c may be determined in the experimental way taking into a consideration other terms to be minimized.

4.4 The minimization terms for relative sizes

Figure 5 shows that specifying relative sizes may easily lead to unsatisfiability.

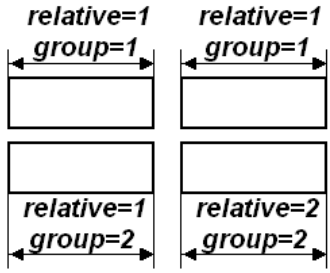

Fig. 5: An example of the unsatisfiable constrains on relative sizes. Two rows, each consisting of two components. The word *relative* may refer to any of the minimum/preferred/maximum relative width fields. The word *group* refers to the relative width group. If all these components have also some positive *minimumWidth* values, then the constraints on relative sizes cannot be satisfied.

There are two rows, each consisting of two components. The component sizes should be aligned to grid. On the one hand, all relative widths (minimum, preferred and maximum) for the first two components are set to 1, i.e., the widths of the first row components should be equal. On the other hand, the second component in the second row must be two times wider than the first component in the same row. Obviously, the only solution is when all the components have zero widths. But specifying some positive values for *minimumWidth* fields of these components gives us no solution at all. To make the layout engine flexible, we use the method described below that will find an approximate solution should such a situation as in Figure 5 occur.

Assume there are two components, A and B, in the same relative width group (for the heights the method is similar). Let them have preferred relative widths r_1 and r_2, and let x^A_{L} and x^A_{R} be variables for the left and the right bounds of the first component as well as x^B_{L} and x^B_{R} for the second. Then, obviously, the desired equation is:

$$r_2 \cdot (x^A_{\mathrm{R}} - x^A_{\mathrm{L}}) = r_1 \cdot (x^B_{\mathrm{R}} - x^B_{\mathrm{L}}). \tag{2}$$

Since not always this equation may be satisfied by the reasons mentioned above, it is better to replace it with the approximate equation. Let's write (2) in this

form:

$$r_2 \cdot (x_{\mathrm{R}}^A - x_{\mathrm{L}}^A) - r_1 \cdot (x_{\mathrm{R}}^B - x_{\mathrm{L}}^B) \approx 0.$$

But this form may be rewritten as a term to be minimized by quadratic optimization algorithm:

$$\left(r_2 \cdot (x_{\mathrm{R}}^A - x_{\mathrm{L}}^A) - r_1 \cdot (x_{\mathrm{R}}^B - x_{\mathrm{L}}^B)\right)^2. \tag{3}$$

In case all the constraints can be satisfied, this term will be zero, and, thus, the desired relation will be hold. Otherwise, the difference between the desired and the actual relation will tend to be zero.

Since for any positive k the relative widths $k \cdot r_1$ and $k \cdot r_2$ denote the same relation as r_1 and r_2, we want the quadratic term (3) also be the same. So, prior to creating the term (3), we use the assignment

$$\begin{pmatrix} r_1 \\ r_2 \end{pmatrix} \leftarrow \begin{pmatrix} \frac{r_1}{r_1+r_2} \\ \frac{r_2}{r_1+r_2} \end{pmatrix}$$

for the normalization.

If minimum/maximum relative sizes are also specified, then they are used as follows. Let $(C_1, r_1, s_1), (C_2, r_2, s_2), \ldots, (C_n, r_n, s_n)$ be the triples where r_i denote minimum relative widths and/or heights contained in the same group. C_i are the corresponding components and s_i are equal to corresponding (absolute) minimum width and/or height values.

A coefficient k is calculated first:

$$k \leftarrow \max\left\{\frac{s_i}{r_i}\right\}.$$

Then, for each C_i we set the corresponding *minimumWidth* or *minimumHeight* to the value $k \cdot r_i$.

The same refers to maximum sizes with the following differences:

- For all the triples (C_i, r_i, s_i) with the r_i set to maximum relative width/height and s_i set to maximum width/height, r_i and s_i must be defined.
- To calculate k we use not max, but $\min\{s_i/r_i\}$.

4.5 Constraints for margins, borders, padding and spacing

We introduce the following constrains for the margins:

$$x_{\mathrm{L}}^C - x_{\mathrm{L}M}^C \geq \textit{leftMargin},$$
$$x_{\mathrm{R}M}^C - x_{\mathrm{R}}^C \geq \textit{rightMargin}.$$

If C is a container, the borders are specified as

$$x_{\mathrm{L}B}^C - x_{\mathrm{L}}^C \geq \textit{leftBorder},$$
$$x_{\mathrm{R}}^C - x_{\mathrm{R}B}^C \geq \textit{rightBorder}.$$

Finally, if C is a non-scrollable container, then we add the following constraints for padding:

$$x^C_{\mathrm{L}P} - x^C_{\mathrm{L}B} \geq \mathit{leftPadding},$$
$$x^C_{\mathrm{R}B} - x^C_{\mathrm{R}P} \geq \mathit{rightPadding}.$$

In case of a scrollable container the second constraint is not added.

The spacing between two components A and B (A on the left of B), obviously, is introduced by the constraint

$$x^B_{\mathrm{L}M} - x^A_{\mathrm{R}M} = \mathit{horizontalSpacing}$$

(equation can be replaced by two inequalities).

4.6 Gravity and alignment

Assume there are nested components (see Figure 6).

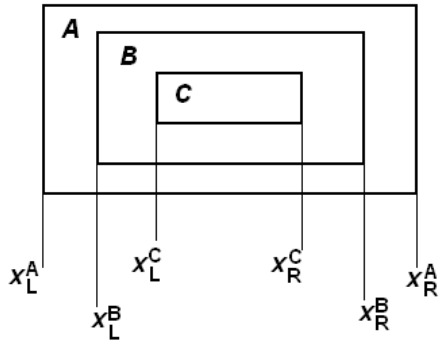

Fig. 6: The nested components. There exists gravity between the borders that forces the inner component to be resized (unless it has the maximum size specified) when the outer component is resized.

If the component C is not resizeable, we still want the component B to be resizeable along with component A. Thus, the gravity between components B and C should be less than between A and B.

Referring to Figure 6, gravity tends to minimize the following differences: $x^C_{\mathrm{L}} - x^B_{\mathrm{L}}$, $x^B_{\mathrm{R}} - x^C_{\mathrm{R}}$, $x^B_{\mathrm{L}} - x^A_{\mathrm{L}}$ and $x^A_{\mathrm{R}} - x^B_{\mathrm{R}}$. The first two differences must be "weaker". That is, if C is not resizeable, then there should be no gravity between B and C. In order to achieve this, we use the following *linear* terms to be minimized:

$$k \cdot (x^C_{\mathrm{L}} - x^B_{\mathrm{L}}) + k \cdot (x^B_{\mathrm{R}} - x^C_{\mathrm{R}}) + l \cdot (x^B_{\mathrm{L}} - x^A_{\mathrm{L}}) + l \cdot (x^A_{\mathrm{R}} - x^B_{\mathrm{R}}),$$

where $k < l$. Assume C is not resizeable, and A has just been stretched. That means we have fixed the sum $(x_{\mathrm{L}}^C - x_{\mathrm{L}}^A) + (x_{\mathrm{R}}^A - x_{\mathrm{R}}^C)$. If $k < l$, then the sum of the four terms will be minimal when the last two terms are zero. That is also the desired behaviour for the gravity. If B is a vertical box with n children, the "weight" k should be divided by n since the minimization terms added by all children sum up.

We should not forget to specify also

$$\begin{aligned} x_{\mathrm{L}}^C - x_{\mathrm{L}}^B &\geq 0, \\ x_{\mathrm{R}}^B - x_{\mathrm{R}}^C &\geq 0, \\ x_{\mathrm{L}}^B - x_{\mathrm{L}}^A &\geq 0, \\ x_{\mathrm{R}}^A - x_{\mathrm{R}}^B &\geq 0. \end{aligned}$$

If the children have to be left-aligned, then instead of gravity between the left container border and the first child we introduce the constraint

$$x_{\mathrm{L}}^B - x_{\mathrm{L}}^A = 0,$$

where A is a container and B is the left child. The same is when the children have to be right-aligned. However, if the children are to be centered, we add the term

$$c \cdot \left((x_{\mathrm{L}}^B - x_{\mathrm{L}}^A) - (x_{\mathrm{R}}^A - x_{\mathrm{R}}^B)\right)^2$$

to the minimization that is used to make the distances from the component to the left and right borders of the parent equal. We add also the constraints

$$\begin{aligned} x_{\mathrm{L}}^B - x_{\mathrm{L}}^A &\geq 0, \\ x_{\mathrm{R}}^A - x_{\mathrm{R}}^B &\geq 0. \end{aligned}$$

4.7 An EQMDC Solver

There exists a method for solving QMDC in a moderate time by a technique that is based on the projective gradient method [2]. The Extended QMDC problem can be reduced to the ordinary QMDC in the following way. First, a constraint graph $G = (V, E)$ is created [3, Section "Difference constraints and shortest paths"]. Here $V = \{s, v_0, v_1, \ldots, v_{n-1}\}$, where all v_i correspond to variables x_i and s is a special start vertex. Edge set E is

$$E = \{(v_i, v_j) : x_i - x_j \geq d_{ij} \geq m_{ij} \text{ is a constraint}\}$$

$$\cup\{(s, v_0), (s, v_1), \ldots, (s, v_{n-1})\}.$$

In the beginning, we consider only d_{ij} values. Rewriting (1), we have:

$$x_j - x_i \leq -d_{ij}.$$

Then, we assign the weight of the edge (v_i, v_j) to the value $-d_{ij}$, while the edges (s, v_i) have the weight 0.

Now, if G does not contain a negative cycle, then the system is solvable, and m_{ij} can be removed leaving only d_{ij}. If G does contain a negative cycle, then the weights of the edges in the cycle are increased to meet the constraints on m_{ij} (corresponding d_{ij} values are *decreased*). If the cycle cannot be eliminated, there is no solution. Otherwise, we continue until all the negative cycles are eliminated.

To achieve practically good execution time, we use the Bellman-Ford-Tarjan algorithm with the subtree disassembly method for finding the negative cycles. Since for directed acyclic graphs negative cycle detection can be performed in linear time, the strongly-connected components are searched in advance. So, the non-linear Bellman-Ford-Tarjan algorithm (which has the upper bound $O(V^3)$) is executed only on strongly-connected components, while considering the edges between these components takes linear time as these edges do not form a cycle.

5 Related Work

Since there exist algorithms for user interface layout that use linear constraints [4,5], one may be interested why the quadratic (and not linear, or, possible, cubic) optimization is preferable here. The two reasons can be mentioned: 1) it is impossible to implement some constraints (like constraints for preferred sizes) by means of a linear function only, and 2) optimizing other non-linear (cubic et al.) functions can be very time-consuming. Regarding the implementation of QMDC, any method can be used here (while we use the method by Freivalds and Ķikusts [2], one could use the method by Hosobe, for instance [6]).

It can be proven that expressive power of Dialog Metamodel with predefined types of containers is at least the same as of standard Java layout managers from the Java Swing library, including a comprehensive *GroupLayout* manager. The main idea of the proof is to show how each standard Java Swing layout manager can be simulated by means of the container types provided by Dialog Metamodel.

There is an interesting difference in resizing policy between our approach and the approach used by the QT library [7]. In QT, when a layout (such as vertical, horizontal, or grid layout) for a container is set, components inside this container are resized by default. To prevent resizing, special components called horizontal and vertical spacers can be used. Spacers act as springs that produce a counterforce for resizing. In contrast, our approach uses maximum width and height constraints to prevent resizing.

Model-driven graphical tool building platforms such as Eclipse GMF [8,9], Microsoft DSL Tools [10], Metaclipse+ [11], and others, usually provide a standard mechanism for creating dialog boxes such as property editors. These stand-

ard mechanisms permit only limited customizations of a dialog box (e.g., we can specify the names of properties and their values, but we cannot add some extra buttons). While the expressiveness of such simple dialogs is sufficient for most cases, some tools (like Microsoft DSL Tools) permit also specifying dialogs of arbitrary complexity in some object-oriented programming language (e.g. C#). But the additional knowledge and skills are required here. Moreover, the model-driven approach is lost.

6 Conclusion

Dialog Engine, mentioned in this paper, has been implemented in C++ Builder, and currently is used in tools based on the Transformation-Driven Architecture, TDA. Examples of such tools are a graphical ontology editor OWLGrEd [12] and a UML editor GradeTwo [13]. Quadratic optimization is called each time when a dialog window is being displayed on the screen, when the user resizes a dialog window, and when windows are changed and refreshed at runtime (by issuing a *RefreshCommand* from Dialog Metamodel).

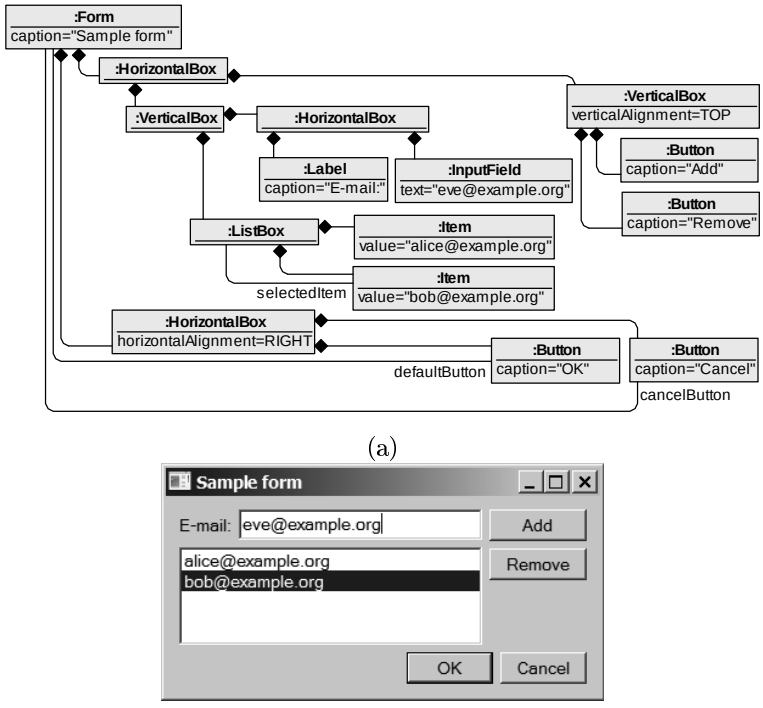

Fig. 7: (a) A model of a sample form. (b) The resulting dialog window on the screen.

An example of a dialog model that does not specify any coordinates and dimensions (only containers and some alignments) is depicted in Fig. 7(a). The resulting dialog window, produced by Dialog Engine, is depicted in Fig. 7(b). This is a real screenshot.

As noticed by Dmitrijs Logvinovs, the number of variables used in quadratic optimization can be reduced up to two times by combining them (e.g., variables for margins can be combined with the corresponding component coordinates). He also started a Java implementation that reduces the number of variables and takes an advantage of using Java reflection mechanism for loading GUI elements at runtime.

Having Dialog Metamodel and a working Dialog Engine for it, Dialog Metamodel can be used as a formal language for describing abstract syntax of dialog windows.

Acknowledgements

I thank Kārlis Freivalds for valuable personal conversations on quadratic optimization. I thank also Dmitrijs Logvinovs for certain improvements that he noticed and incorporated in the Java version of Dialog Engine.

This work has been partially supported by the European Social Fund within the project «Support for Doctoral Studies at University of Latvia».

References

1. S. Kozlovics, "A Dialog Engine Metamodel for the Transformation-Driven Architecture," in *Scientific Papers, University of Latvia*, vol. 756, pp. 151–170, 2010.
2. K. Freivalds and P. Ķikusts, "Optimum layout adjustment supporting ordering constraints in graph-like diagram drawing," in *Proceedings of the Latvian Academy of Sciences, Section B*, vol. 55, pp. 43–51, 2001.
3. T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to algorithms*. MIT Press, Cambridge, MA, U.S.A., 2nd ed., 2001.
4. G. J. Badros, A. Borning, and P. J. Stuckey, "The cassowary linear arithmetic constraint solving algorithm," *ACM Trans. Comput.-Hum. Interact.*, vol. 8, pp. 267–306, Dec. 2001.
5. A. Borning, K. Marriott, P. Stuckey, and Y. Xiao, "Solving linear arithmetic constraints for user interface applications," in *Proceedings of the 10th annual ACM symposium on User interface software and technology*, UIST '97, (New York, NY, USA), pp. 87–96, ACM, 1997.
6. H. Hosobe, "A modular geometric constraint solver for user interface applications," in *Proceedings of the 14th annual ACM symposium on User interface software and technology*, UIST '01, (New York, NY, USA), pp. 91–100, ACM, 2001.
7. "QT Developer Network." `http://qt-project.org/`.
8. "Graphical Modeling Framework (GMF, Eclipse Modeling subproject)." `http://www.eclipse.org/gmf`.

9. A. Shatalin and A. Tikhomirov, "Graphical Modeling Framework architecture overview," in *Eclipse Modeling Symposium*, 2006.
10. S. Cook, G. Jones, S. Kent, and A. Wills, *Domain-Specific Development with Visual Studio DSL Tools*. Addison-Wesley, 2007.
11. A. Kalnins, O. Vilitis, E. Celms, E. Kalnina, A. Sostaks, and J. Barzdins, "Building tools by model transformations in Eclipse," in *Proceedings of DSM'07 Workshop of OOPSLA 2007*, (Montreal, Canada), pp. 194–207, Jyvaskyla University Printing House, 2007.
12. J. Barzdins, G. Barzdins, K. Cerans, R. Liepins, and A. Sprogis, "OWLGrEd: a UML style graphical notation and editor for OWL 2," in *Proceedings of OWLED 2010*, 2010.
13. "The GradeTwo tool." `http://gradetwo.lumii.lv/`.