A layout inference algorithm for Graphical User Interfaces

Óscar Sánchez Ramón[a], Jesús Sánchez Cuadrado[b], Jesús García Molina[a,*], Jean Vanderdonckt[c]

[a] *Faculty of Informatics, Campus of Espinardo, 30100, Murcia, Spain*
[b] *Superior Polytechnic School, Autonomous University of Madrid, Francisco Tomás y Valiente, 11, 28049, Madrid, Spain*
[c] *Louvain School of Management, Catholic University of Louvain, B-1348, Louvain-La-Neuve, Belgium*

Abstract

Context: Graphical User Interface (GUI) toolkits currently provide *layout managers* which lay out widgets in views according to certain constraints that characterise each type of layout manager. In some scenarios such as GUI migration and the automated generation of GUIs from wireframes, the layout of views is implicitly expressed through the use of coordinates. In these cases, it is desirable to represent the layout explicitly in terms of layout managers.

Objective: To represent a coordinate-based GUI in terms of a set of layout managers, in order to provide different alternative solutions for a given view and select the best alternative.

Method: The layout inference process consists of two phases. Firstly, the coordinate-based positioning system is changed to a relative positioning system based on directed graphs and Allen relations. Secondly, an exploratory algorithm based on pattern matching and graph rewriting is applied in order to obtain different layout solutions. The algorithm has been evaluated through a case study related to the automatic generation of fluid web interfaces from wireframes, involving 20 IT professionals.

Results: The case study showed that the layout obtained is faithful to the original views in 97% of cases, and maintains its proportions when resized in 84% of views. The majority of the participants were satisfied with the results and found the approach useful.

Conclusions: The layout manager representation obtained from the coordinate-based GUIs can be used to generate fluid layouts. The algorithm has two main features that overcome the limitations of the existing approaches: independence of specific layout managers and ability to generate several alternative solutions.

Keywords: Graphical User Interfaces, Layout Inference, Wireframes, Model-Driven Engineering, Reverse Engineering, Fluid layout.

1. Introduction

Graphical User Interface (GUI) toolkits currently provide *layout managers* which lay out widgets in views according to certain constraints that characterise each type of layout manager. For example, a *FlowLayout* manager in Java Swing arranges widgets in a row, one after the other. Layout managers are useful to implement GUIs that are well displayed under certain conditions such as different screen dimensions and resolutions, or different window sizes. The use of layout managers became a widespread practice in the late 1990s owing to the great variety of devices and screen dimensions available. However, in some scenarios, GUIs use absolute coordinates, and a layout inference process is thus needed to recover the layout implicitly defined in GUIs of this nature.

One of these scenarios is the migration of legacy software. Views were formerly created by using visual editors that allowed developers to drag and drop widgets from a palette onto a canvas to a particular position (which was sometimes almost arbitrary). This is the case of those applications created using Rapid Application Development (RAD) environments [10]. In these cases, the position of widgets was expressed in terms of abolute or relative coordinates (normally pixels), signifying that these views were only optimised for a particular window size. Hence, legacy systems that are migrated to new technologies or platforms frequently

*Corresponding author. Tel.: +34 868884610
Email addresses: osanchez@um.es (Óscar Sánchez Ramón), jesus.sanchez.cuadrado@uam.es (Jesús Sánchez Cuadrado), jmolina@um.es (Jesús García Molina), jean.vanderdonckt@uclouvain.be (Jean Vanderdonckt)

switch from a coordinate-based positioning system to a relative one handled by layout managers. In order to switch the positioning system, layout inference is required to extract an explicit representation of the layout which is implicitly expressed in the positions of the widgets.

Another scenario in which GUIs are defined in terms of coordinates is the design of user interfaces by means of wireframes or mockups, prior to implementation. These artefacts are created with visual tools, which are used by stakeholders to discuss and refine a GUI design for a new application. Once the GUI design has been validated, it is frequently discarded and the implementation starts again from scratch. Nonetheless, these artefacts can be reused to generate the final GUI code, which also requires a layout inference process.

In this article we present a general approach that can be used to infer an explicit layout from GUIs in which the layout is implicitly expressed in coordinates. The layout obtained is represented as a composition of layout managers.

The present work is based on a previous one [15] in which we defined a model-based approach with which to migrate the GUIs in RAD applications to modern platforms. However, there was a series of shortcomings that limited its applicability to scenarios other than RAD applications, such as the fact that the different parts of the GUI needed to be delimited by frames, widget alignment was not considered, and notably the algorithm used to recognise the high-level layout was simple and was only able to detect layouts that clearly matched one of the predefined layout types. One of the shortcomings of the previous algorithm, which was based on a greedy strategy, was its inability to detect complex layout compositions which as an important limitation in other scenarios such as the layout recognition in wireframes. This has motivated the design of a new layout inference algorithm that should satisfy three main requirements: i) it must provide a solution consisting of a composition of layout managers, ii) it must work well with GUIs which have parts which are not delimited by frames, and iii) it must be independent of any particular application scenario (i.e. more general than our original algorithm).

Our new solution is an exploratory algorithm that is able to extract different alternative layout compositions for a given GUI design, based on a set of layout managers, and then select the best alternative. Furthermore, whereas the previous approach was focused on a reverse engineering scenario, in particular the migration of RAD applications, the new algorithm is more sophisticated and obtains better results in other, more unconstrained scenarios, such as the layout inference from wireframes.

This work contains two main contributions, the first of which is the layout inference algorithm and the data structure that supports it. Our algorithm has two significant advantages over existing proposals, which are a novelty in layout inference approaches: i) the user can choose the subset of layout managers that will be used to compose the layout; and ii) the algorithm not only outputs the best layout composition (according to certain assessment criteria), but also returns different alternative layout compositions which may be valuable for developers. Our approach also has two important features already considered in other proposals: i) it is independent of the source language or tool in which the GUI was programmed or designed, and is independent of the target technology or toolkit in which the new GUI will be implemented; and ii) it allows some degree of imprecision when placing or spacing widgets, which will be corrected in the new GUI. The second contribution is a case study involving 20 IT professionals, in which the layout inference approach is used to generate fluid web interfaces from wireframes. The results of the case study show that our algorithm produces accurate layouts and that the approach is useful in practice. Finally, our approach is supported by a tool, implemented as an Eclipse plug-in, that is available for downloading at `http://www.modelum.es/guizmo`.

The paper is organised as follows. Section 2 presents some key concepts that are used throughout the paper, along with some scenarios in which layout inference is relevant, and introduces the features of a layout. Section 3 presents the models that are the input and output of our layout inference approach, which is explained in Section 4. The performance evaluation of our implementation is presented in Section 5, and a case study in which the layout inference solution is used to generate fluid interfaces from wireframes can be found in Section 6. Finally, the related work is discussed in Section 7 and the conclusions and future work are given in Section 8.

2. Background and motivation

2.1. Key concepts

A user interface (UI) is the part of a software/hardware system that is designed to interact with users. A *Graphical User Interface* (GUI) is an interface that uses computing graphics such as icons and menus (e.g. the Android GUI). User interfaces have a static part that is related to the presentation of the information

(i.e. the structure, the layout, the usability, the accessibility or the aesthetics) and a dynamic part that is related to their behaviour when the user interacts with them (i.e. the events that are triggered and perform actions and/or changes in the interface). The area of interest of this work is focused on the static part of GUIs, and particularly their layout.

The *layout* of a graphical user interface is the spatial distribution of the elements in the views of the application. *Views* are the graphics that are displayed on device screens (windows in desktop applications, web pages in web applications or views in mobile applications). The elements laid out in the views are widgets or visual controls (e.g. buttons or combo boxes). Those widgets that can contain other widgets (i.e. they have nested widgets) are called *containers*. In this respect, views are also containers and are actually the topmost components in the hierarchy of the GUI elements.

The first *GUI toolkits* located widgets by means of a pair of coordinates that set the reference point in a corner of the screen. The coordinate-based positioning system has been used for a considerable amount of time, and it is in fact still possible to create GUIs by placing widgets with absolute coordinates. However, when monitors with different screen sizes and resolutions arrived on the market, coordinate-based GUIs were not smartly displayed in these unexpected canvases. Libraries of layout managers then appeared in many programming languages in order to overcome this shortcoming.

A *layout manager* is a software component that automatically lays out the widgets on a view based on relative relations and restrictions specified by the programmer. A layout manager is useful for creating GUIs that are properly displayed on screens that have different features, or that adapt to the user's preferences (changes of fonts, view sizes, etc.). *Swing*[1] is one of the most popular widget toolkits in Java, and offers layout managers such as *BoxLayout* that arranges components in a left-to-right or top-to-bottom flow, or *GridLayout*, which lays out the components in a rectangular grid.

Web pages use a relative positioning system based on flows of vertical and horizontal elements, which is provided by HTML and Cascading Style Sheets (CSS). In HTML, there are inline items and block items. Inline items are laid out in the same way as the letters in words in a text, one after the other across the available space until there is no more room, and a new line is then started below. Block items stack vertically, like paragraphs and the items in a bulleted list. CSS allows us to modify which elements are displayed inline or as a block, and also allows us to specify which elements are floated. The latter are elements that are taken out of the normal flow and shifted to the left or right as far as possible in the space available. The CSS layout system can be considered as a variant of the *BoxLayout* in which it is possible to mix vertical and horizontal layouts, elements can be directly attached to the right or left borders of the container and, by default, the elements do not take up all the empty space inside the container.

Several front-end web frameworks have appeared on top of CSS, such as Bootstrap[2], which offers CSS styles and Javascript components with which to create appealing and consistent web interfaces. Bootstrap provides a grid layout system in which the content is arranged in rows composed of up to 12 columns that can be merged, thus allowing developers to indicate how the content is distributed in the columns for each row.

Table 1 shows some classical desktop layouts that are available for Java Swing and establishes mappings to modern layouts for the web platform (ZK, JSF) and the mobile platform (Android, iOS). Mappings between layouts are not exact but there are certain features that make some of them more powerful than their equivalent. However, this proves that the ideas behind classical layout managers are still present in modern layouts.

The main issue addressed in this work is the inference of the layout of coordinated-based GUIs and its explicit representation, i.e. shifting from a coordinate-based positioning system to a relative positioning system based on layout managers.

2.2. Layout inference scenarios

The recovery of the implicit layout of a GUI is a mechanism that could be useful for GUI development in general. Four scenarios in which such a mechanism to infer and explicitly represent the layout of a GUI could be useful are briefly introduced as follows.

Layout-preserving migration. Application users are sometimes averse to change, and some migration projects therefore have a requirement which specifies that the original GUI layout must be preserved in the target application. Another example of the utility of layout preservation would be the case of generating a mock application which tracks user input in order to generate test cases [11].

[1] http://docs.oracle.com/javase/tutorial/uiswing/

[2] http://getbootstrap.com/

Java Swing	ZK	JSF PrimeFaces + CSS	Android	iOS
FlowLayout	hlayout		LinearLayout	
BoxLayout	hlayout, vlayout, columnlayout		LinearLayout, ListView	
GridLayout			GridView	
BorderLayout	borderlayout	Layout	RelativeLayout	
GridBagLayout	tablelayout, anchorlayout	YUI grid (CSS)	RelativeLayout	AutoLayout

Table 1: Mapping between the most common Java Swing layouts and modern GUI frameworks

Reengineering for GUI adaptation. Migrating to a new GUI technology implies that it is necessary to take advantage of the target technology's features (e.g. usability standards, high-level layout models of modern GUI toolkits, etc.). A particular case of this category would be the migration to technologies with constraints related to the visualisation display, such as mobile devices. Given an explicit layout model, the developer could either refactor the layout manually or (semi-)automatic transformations could be applied.

Quality improvement. Perfective maintenance tasks may be required to improve the system quality, such as the detection of usability issues, non-visible widget removal, GUI resizing and beautification [7]. An explicit representation of the layout would enable this analysis, and would also permit refactoring.

Final GUI generation. Forward engineering approaches with which to develop new systems can also benefit from layout inference. In software development methodologies, GUI designs are validated in the early stages of development with a mockup (a plain mockup is shown in Figure 1), which is a GUI representation that is created before the final product, thus allowing stakeholders to check it. The same approach used in reverse engineering can be applied to the development of a new system just by taking mockups as source artefacts. Final GUIs for different platforms or technologies can then be generated from the GUI representations.

2.3. Layout features

Shifting from a coordinate-based positioning system to a relative positioning system based on layout managers is not a straightforward task, but it requires the extraction of different features from a GUI. These features will be the basis to recover an explicit representation of the layout, as will be presented in Section 4.

Visual structure. This is related to the human perception of the widget arrangement, and is a key feature as regards allowing the content of a view to be adapted to different text or screen sizes. The visual structure is detected by analysing the positions of all the elements in the view in order to recognise the 'shapes' they form and how they are visually grouped. Two examples of visual structures are a horizontal flow of widgets or a grid of elements. Note that this feature is in some respects subjective, and different alternative visual structures may therefore be equally valid for the same view.

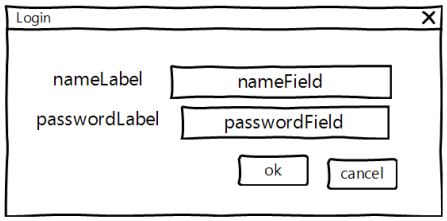

Figure 1: Login window created with WireframeSketcher.

For example, in the login view shown in Figure 1, *nameLabel* and *nameField* form a line (a *FlowLayout* in Java AWT), *passwordLabel* and *passwordField* form a second line, and the *ok* and *cancel* buttons form a third line. Another possible layout would be to put *nameLabel* and *passwordLabel* in one column, and the remaining widgets in another column.

Some widgets are sometimes surrounded by a rectangle because they are related to the same topic, or there are groups of widgets that are visually distant from other groups. In these cases, the groups should be identified and handled as a unit.

Another feature related to the visual structure is the association between labels and components, because labels are frequently associated with other components such as text fields or combo boxes. The semantic relationship between the labels and the associated components may be interesting when generating the final UI. For example, in HTML, the *for* attribute is used to associate labels with other components, and when the user clicks on a label with the mouse the browser will automatically shift the focus to the corresponding field.

Spacing and sizing. The spacing between the widgets

in the view is also relevant. We must distinguish between the gaps and the margins. We call the spacing between the single widgets (e.g. the separation between a label and a text field) *gaps*, while *margins* are the distances between the single widgets and their container. Note that gaps and margins are either *horizontal* or *vertical*, depending on the axis on which they are observed.

The size of the widgets is another feature that must be extracted. Gaps, margins and sizes should be expressed in relative units, such as percentages regarding the container element, signifying that the measures are independent of the concrete screen of the device.

Alignment. The alignment is either horizontal or vertical, and it is defined for one widget with regard to other widgets, or is defined for a widget with regard to the container. For instance, in Figure 1 the widgets *nameField*, *passwordField* and *cancel* are aligned to the right with regard to each other.

A careful observation of these three widgets shows that they are not perfectly aligned, although one would expect them to be aligned. When dealing with the layout, we should therefore accept some degree of misalignment, i.e. the analysis of the positions of the elements must be flexible.

In addition to extracting all the aforementioned information about the layout, it is desirable for a layout inference approach to have two qualities. The first quality is independence of the source and target platforms and GUI technologies, thus allowing the same solution to be applied to any GUI. The second quality is flexibility, signifying that the approach can admit some degree of imprecision when analysing the coordinates of the elements and is still able to infer a good layout.

3. Input and output models

This section provides a description of the data structures (represented by means of class diagrams) that are the input and the output of the layout inference approach described in Section 4. The Structure model is the input and the Layout model is the output. Both models are independent of a concrete technology. There is an intermediate model, the Tile model, which will be explained in Section 4.1.

3.1. Input: Structure model

Figure 2, which has been simplified in order to ease explanations, shows the *Structure model* used to represent the logical structure of a view (i.e. the hierarchy

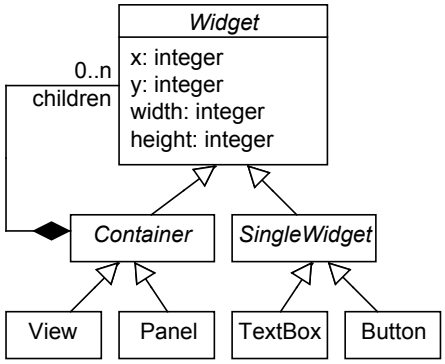

Figure 2: Basic GUI model.

of the GUI elements). All the elements (and the view itself) are considered to be *Widget*s. *SingleWidget*s such as *TextBox*es or *Button*s are elements that are not composite. *Container*s such as *View*s (e.g. a window or a web page) or *Panel*s are elements that are used to visually group other elements. Every element in a view is located by means of the *x* and *y* coordinates, and has a *width* and a *height*.

We impose two semantic restrictions on this model: i) overlapped *Widgets* are not permitted, and ii) *Containers* and *SingleWidgets* must not exist at the same level (i.e., two *Widgets* having the same parent are either both *SingleWidgets* or both *Containers*). The second restriction is imposed as a means to structure the GUI in a uniform manner, given that views are divided into parts which are disjointed and complementary. Each view therefore contains several separate *Containers* (which can in turn contain more *Widgets*), and each *Widget* belongs to a unique *Container*. The goal of this design decision is twofold. On the one hand, we believe that a UI is conceptually composed of related parts like a puzzle in such a way that there are no widgets outside a part. This might be interesting when, for example, distributing the user interface in several devices. Furthermore, it makes the structure of the UI uniform and eases the later algorithms. Nevertheless, transformations can be applied to adapt any GUIs that do not satisfy the constraint ii), as shown in [15].

3.2. Output: Layout model

The *Layout model* defines the design of the views, i.e. how widgets are spatially arranged in views. The design is expressed in terms of layout managers, which is a better representation system than others such as coordinates or positioning based on boxes (e.g. pure HTML). The Layout model can be seen in Figure 3. Note that the layout types have been inspired by the Java Swing

layout managers shown in Table 1, but they are not exactly the same. More particularly, our *FormLayout* can be considered as a particular case of the *GridBagLayout* in which there are horizontal sequences of widgets that are aligned in columns.

In the Layout model we have defined *LayoutElements* that can be *ElementNodes* or *Layouts*. An *ElementNode* represents a *Widget* (either a *Container* or a *SingleWidget*) that is managed by a *Layout*.

Layouts are arranged hierarchically, signifying that the layout of an element can be a composition of layouts. The set of predefined layouts currently supported is: *FlowLayout*, *BorderLayout*, *GridLayout*, *FormLayout* and *CustomLayout*. A *FlowLayout* is a horizontal flow or a vertical flow depending on the *type* attribute. A *BorderLayout* is a layout that places the content in five areas: *top*, *bottom*, *left*, *right*, *centre*. Not all five areas of the *BorderLayout* have to be occupied. A *GridLayout* arranges the elements in a grid of $numRows \times numCols$ cells of equal size (only the *numCols* attribute is stored). A *FormLayout* is a more complex layout that is applied to rows of elements in which some of the elements are vertically aligned in more or less the same way. This layout contains a list of vertical *FlowLayouts* and additionally defines some *AlignedColumns* in such a way that every element must belong to just one *AlignedColumn*. *lnodes* represents the elements aligned to the left bound of the column and *rnodes* is the same for the right bound. *nodes* include all the elements contained in the column (aligned or not). *CustomLayout* is used if the GUI layout cannot be composed of a composition of the predefined layouts.

A *Layout* also includes some other attributes that are useful as regards tuning the design defined by the predefined set of layouts. The *hAlignment* and *vAlignment* attributes are used to indicate how a layout is horizontally and vertically aligned. We have two different cases: i) when the *Layout* is associated with a *Container*, the alignment is relative to that Container; ii) when the *Layout* is part of a more complex layout (it is nested in another *Layout*), then the alignment is relative to the enclosing *Layout*. We refer to a *Layout* that is part of a complex *Layout* as an *intermediate layout*.

hSize and *vSize* are the percentages of the horizontal and vertical space (respectively) taken up by the element, with regard to the *Container*. Adjacent elements are commonly separated by horizontal or vertical *Gaps* (empty space), and *Margins* represent the distance between a layout and the bounds of the *Container*. It is worth noting that *Margins* are only applicable to the *Layouts* associated with a *Container*, and not to their children *Layouts*. In the case of both *Gaps* and *Margins*, the distances are measured using percentages of horizontal or vertical distances with regard to the *Container*.

Each *Widget* of the Structure model is reproduced in the Layout model by a *LayoutInfoTreeNode* which contains the reference to it (*element* reference). As mentioned in Section 2.3, there may be different visual structure compositions that might result in users perceiving a similar layout. It may therefore be possible to use different layout alternatives to lay out the same *Container*, signifying that every *LayoutInfoTreeNode* that is associated with a *Container* will have a set of possible layouts (*alternatives*). Each *Layout* contains a reference (*refNode*) to the *LayoutInfoTreeNode* to which the layout is applied, with the exception of intermediate layouts, which are not linked to any *Widgets*.

Each alternative has a *fitness* value (between 0 and 1) that serves to compare the alternatives among them and discover which of them are better than others. The closer to 1, the better a solution is. Fitness values are meant to be used to compare different alternatives for the same *Container*, and should not be used to compare solutions for different *Containers*.

4. Layout inference

In this section we present our layout inference process, which generates an instance of the Layout model from an instance of the Structure model. It consists of two main stages, each of which is composed of three steps that are depicted in Figure 4. The first stage changes the positioning system from coordinates to relative positions between the elements. The outcome of the first stage is a graph representation of the view with relative positions. The second stage takes that graph and applies a pattern matching algorithm in order to obtain a tree of layout managers (the Layout model that was presented in Figure 3). These two stages are explained in the following two subsections.

4.1. Creating the Tile model

As an intermediate step in the transition from absolute coordinates to a layout representation using layout managers, we create a representation of the GUI using a relative positioning system based on the spatial relations among widgets, which we call a *Tile model*. This representation will be the basis for the layout inference algorithm.

The creation of the Tile model (step 1) will be presented in Section 4.1.1, the relative positioning (step 2) will be explained in Section 4.1.2, and Section 4.1.3 will

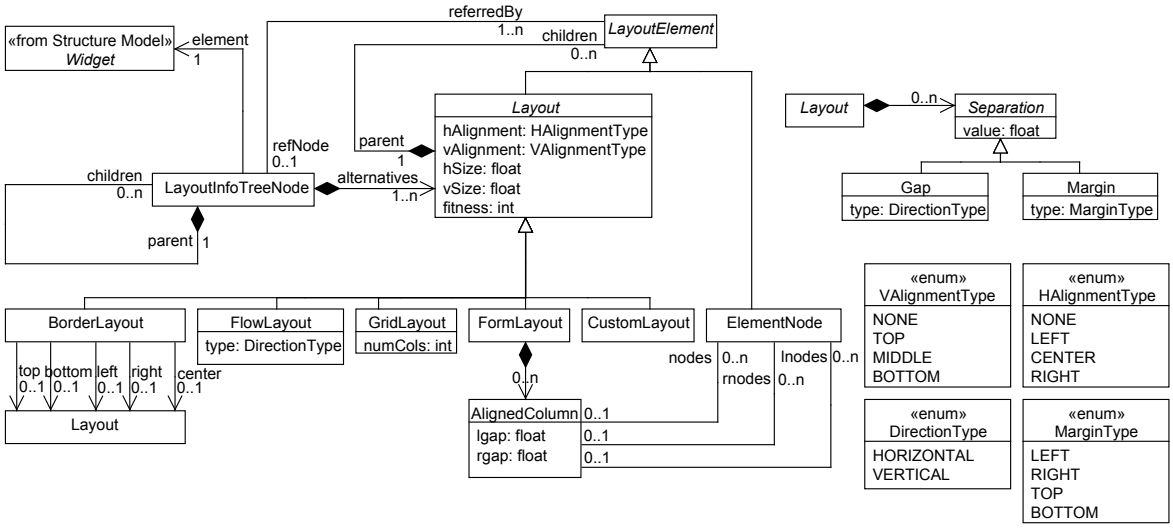

Figure 3: Layout model.

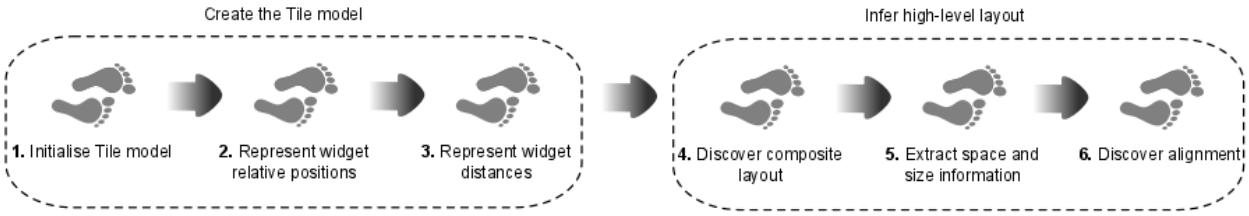

Figure 4: Steps to explicitly infer the layout information.

explore the representation of the distance between elements (step 3).

4.1.1. Initialising the Tile model

We propose to represent a view by means of a nested, attributed, relational acyclic directed graph, in which the nodes can be digraphs and both the nodes and the edges have attributes. The data structure that defines these graphs is the class diagram presented in Figure 5, which we call the *Tile model*. This representation is focused on making positions between elements explicit, which is very useful as regards detecting layout patterns.

The model contains two main classes, *TileNode* and *Relation*. A *TileNode* represents a rectangular area of a view that contains a widget or a group of widgets. As can be seen, *TileNodes* contain information about the coordinates of the area they take up. Although our inference algorithm is applied to the Tile model as a graph, coordinates are still needed to calculate certain attributes such as the margins of a widget with regard to the container size.

A *WidgetNode* is a *TileNode* that represents the area of a widget (either *SingleWidget* or *Container*) and contains the reference to that *Widget* in the Structure model. A *LayoutNode* is a *TileNode* that represents the area of a group of widgets that are laid out according to a certain layout *type*. As will be shown later, at the beginning of the layout inference process we have a graph that only contains *WidgetNodes* while at the end of the process (after applying the rewriting) we have a graph composed of *WidgetNodes* and *LayoutNodes*. From here on, *TileNodes* will be indistinctly referred to as *tiles* or *nodes*.

Relation represents a spatial relation between two *TileNodes* (*source* and *target*) by means of three attributes. The first attribute is the Allen relation for the X-axis (*xRelation*), the second attribute is the Allen relation for the Y-axis (*yRelation*) and the third parameter is the closeness level between a pair of connected *TileNodes* (*closeness*).

The Allen's Interval algebra [1] can be used to express the spatial relations among objects. In our case, the basic relations of this algebra serve to represent the relative positioning of nodes (e.g. if a node is to the right of or below another node) and the alignment of one node with regard to another. This algebra will be

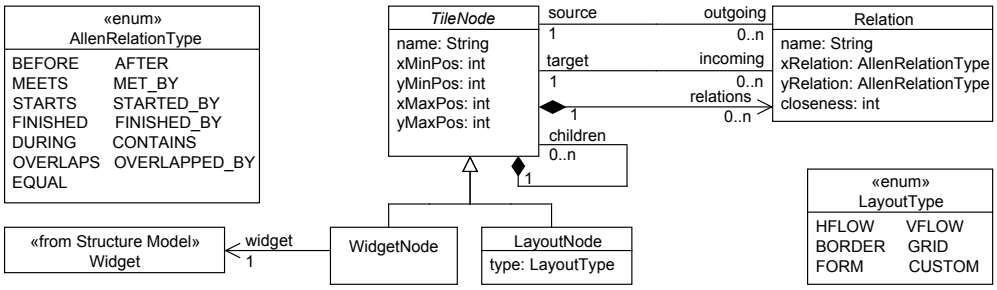

Figure 5: Tile model

explained in more detail in Section 4.1.2.

The distances between widgets are not represented by means of absolute or relative units, but rather by calculating the so called *closeness level*, which is a means to classify widget distances into groups. The distances that belong to the same group are more or less similar. This will be used by our algorithms to prioritise close widgets over more distant widgets when applying pattern matching to detect layouts. Please note that only one attribute is sufficient to represent this feature (*closeness*), because we do not allow widget overlapping, and it thus measures the distance on only one axis according to the Allen relations. More aspects of the meaning of the closeness levels will be discussed in Section 4.1.3.

The Tile model is created as follows. A *WidgetNode* is created for each *Widget* of the Structure model. *WidgetNodes* keep a reference to the *Widget* from which they are created, along with a copy of the coordinates of the element. The containment hierarchy of the Structure model is therefore replicated when creating the Tile model. For instance, if there is a *View* which aggregates three *children Panels* in the Structure model, then there will be a *WidgetNode* containing three *children WidgetNodes* in the Tile model. A *Relation* is created for each pair of adjacent *WidgetNodes*. A tile t_1 is *adjacent* to another tile t_2 if and only if i) the projection on the X-axis or Y-axis of both tiles is overlapped and ii) there is not a tile t_3 between t_1 and t_2 (see [15] for a more in-depth explanation). The Allen relations for the X (*xRelation*) and Y (*yRelation*) axis are calculated for each *Relation* created, and the closeness level (*closeness* attribute) is assigned to it.

4.1.2. Representing widget relative positions

In this section we explain how the Allen's Interval algebra [1] has been used to express the relative positions between the widgets and how they are obtained.

Figure 6 shows the basic relations between intervals and their meaning applied to line segments. For example, if *A MEETS B* then this means that the A interval is before the B interval and that the end of A 'touches' the beginning of B (i.e. there is no blank space between them). Note that all the relations (with the exception of *EQUALS*) have an opposite relation, e.g. if *A MEETS B* this implies that *B MET_BY A*. In order to represent the spatial relations of 2D objects we need two relations, one for the projection of the node on the X-axis and another for the projection of the node on the Y-axis.

Basic relation	Meaning	Opposite relation
BEFORE		AFTER
MEETS		MET_BY
STARTS		STARTED_BY
DURING		CONTAINS
FINISHES		FINISHED_BY
OVERLAPS		OVERLAPPED_BY
EQUALS		-

Figure 6: Allen relations

Note that *Relations* in Tile models are directed, i.e. they distinguish between the source and the target node of the *Relation*. Given that Allen relations are defined on ordered pairs of intervals (line segments in our case), the pair *(source, target)* indicates how to interpret the Allen relations. For instance, let $r(t_1, t_2)$ be an Allen relation between *TileNodes* t_1 and t_2 and *r.xRelation* = $Before$ signifying that t_1 is before t_2 with regard to the projections on the X-axis. The pairs of *TileNodes* in a *Relation* are ordered as follows: the target *TileNode* is always to the right or below the source *TileNode*. As stated previously, *TileNodes* are arranged in a hierarchy, signifying that a *TileNode* can contain some other *TileNodes* and *Relations*.

Each *Relation* is thus represented by two Allen relations: one for the X axis (*xRelation*), which is based on the comparison between the y-coordinates of both *TileNodes*, and another for the Y axis (*yRelation*), which is based on the comparison of the x-coordinates. As already indicated, one Allen relation between intervals allows relative positions between pairs of objects to be represented in one dimension. Two relations are therefore needed to represent the relative position of two widgets (represented as boxes) in a bidimensional space. The comparisons of the positions needed to calculate the Allen relations are carried out using a margin m that is parameterised, signifying that for a pair of widgets w_1 and w_2, $w_1.x = w_2.x$ if $w_1.x \in (w_2.x - m, w_2.x + m)$. By default the margin of the comparisons has been set to 10 pixels. This allows the negative effect of misalignment to be avoided.

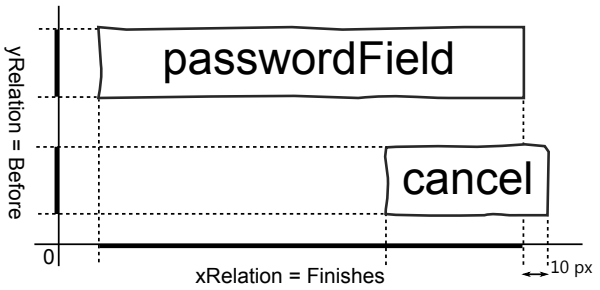

Figure 7: Allen relation example for a pair of widgets

Figure 7 shows the Allen relations that describe the *cancel* tile regarding the *passwordField* tile, extracted from the view example in Figure 1. Given that the projection of the *passwordField* on the Y-axis precedes the projection of *cancel*, then the *yRelation* is *Before*. With regard to the X-axis, the projection of *cancel* exceeds the end of *passwordField* in 10 pixels. If we were to be strict (the comparison margin would be set to 0 pixels), then the relation that describes the relative position of both projections would be *Overlaps*. However, as we have set the comparison margin to 10 pixels, the excess is not significant, and both projections can thus be considered to end at the same point. In this case *xRelation* is therefore *Finishes*.

4.1.3. Representing widget distances

This is step 3 in Figure 4. The closeness levels provide *Relations* with meaningful distances. The levels are obtained by taking the distance measured in pixels and mapping it into a finite set of values (levels).

One simple way in which to classify distances in levels might be to set fixed ranges. For instance, Figure 8

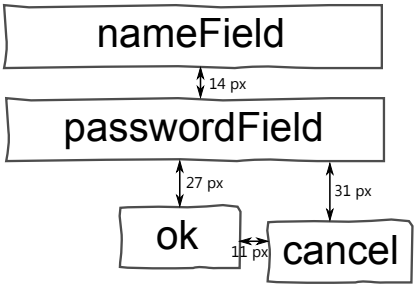

Figure 8: Example of closeness levels.

shows a portion of a view with four widgets (extracted from Figure 1). If we establish that a distance between 1 and 30 pixels is mapped into a *Very_Close* level, and that a distance greater than 30 pixels is mapped into *Close*, then we would obtain that the closeness level between *passwordField* and *ok* (*Very_Close*) would be different to the closeness level between *passwordField* and *cancel* (*Close*). Since the difference in the distances between *passwordField-ok* and *passwordField-cancel* is not significantly different when a user sees the view, they should be tagged with similar levels.

As can be deduced from the example, the classification of the distances should not be accomplished using absolute distances but rather variable limits that depend on the data set. In this respect, a group of nodes that are more or less at the same distance should always be in the same group, and the closeness level defines a partitioning of the nodes into groups according to the distance.

We address this issue by applying a clustering algorithm, which performs a dynamic partition of the set of distances in the view. The partitioning of the distances is then used to classify the *Relations*. The details of this process can be found in Algorithm 1.

The algorithm first obtains all the distances (vertical and horizontal) of the relations and uses them to create a single cluster (lines 2 to 5). In the example in Figure 8, we initially have $BestPartition = \{\{11, 14, 27, 31\}\}$.

The population standard deviation (σ) of the distances is used to measure whether the cluster is sufficiently homogeneous, i.e. the distances in the cluster can (to a certain degree) be considered similar. If the standard deviation of the initial cluster is greater than the maximum closeness cluster deviation (*maxDev*, which is by default 15, but for the current example will be taken as being 8), then it is necessary to split the cluster (line 7). In the example, $\sigma_{\{11,14,27,31\}} = 8.44 > maxDev$, signifying that the distances have to be split into clusters.

In order to perform the clustering of distances, we

Algorithm 1 Closeness assignment algorithm

```
 1: procedure AssignCloseness(Relations, maxDev)
 2:     AllDistances ← getAllDistances(Relations)
 3:     nClusters ← 1
 4:     Cluster ← AllDistances
 5:     BestPartition ← {Cluster}
 6:
 7:     if σ_Cluster > maxDev then
 8:         partitionOK ← false
 9:         while ¬partitionOK do
10:             nClusters ← nClusters + 1
11:             for i ← 1, Num_Iterations do
12:                 Clusters ← kMeans(AllDistances, nClusters)
13:                 if isBestPartition(Clusters, BestPartition,
                        SumOfSquaredErrors()) then
14:                     BestPartition ← Clusters
15:                 end if
16:             end for
17:             if ∀C ∈ BestPartition, σ_C ≤ maxDev then
18:                 partitionOK ← true
19:             end if
20:         end while
21:     end if
22:
23:     SortedPartition ← sort(BestPartition)
24:     closeness ← 1
25:     PartitionMap ← {}
26:     for all Cluster ∈ SortedPartition do
27:         range ← getRange(Cluster)
28:         PartitionMap[range] ← closeness
29:         closeness ← closeness + 1
30:     end for
31:
32:     for all relation ∈ Relations do
33:         d ← getDistance(relation)
34:         relation.closeness ← PartitionMap[d]
35:     end for
36: end procedure
```

have selected the k-means algorithm [9] (line 12), with the euclidean distance as a similarity function. Given that k-means is a divisive algorithm, the number of clusters must be passed as a parameter. However, we do not know the number of clusters a priori. We therefore apply the k-means algorithm several times (lines 7 to 20), while increasing the number of clusters in each iteration (line 10) until the stop condition. This condition is that the standard deviation of every cluster is less than *maxDev* (line 17).

Because k-means is a heuristic algorithm, it is very fast, but it could fall into a local maximum. In order to obtain a better clustering, the algorithm is executed multiple times (lines 11 to 16) with different random starting conditions. The number of iterations, *Num_Iterations* variable, is by default 20, and we keep the best solution according to the intra-cluster homogeneity criterion, which is the sum of the squared errors (line 13). To continue with the example, the k-means algorithm is applied with $nClusters = 2$ and the output is: $BestPartition = \{\{11, 14\}, \{27, 31\}\}$, and we thus obtain that $\sigma_{\{11,14\}} = 1.5 < maxDev \wedge \sigma_{\{27,31\}} = 2 < maxDev$, so the clustering loop stops.

After the clusters have been obtained, they are sorted and a numerical tag is assigned to each one (lines 23 to 30). The lesser the values (distances) of the cluster, the lesser the numerical value of the tag (the lower distance group is tagged with *1*). For each cluster, we obtain the minimum and a maximum value in pixels (lines 27). In the example, *PartitionMap* maps each range to a closeness level: $(-\infty, 14] = 1$ and $[15, +\infty) = 2$. Once this entire process has been accomplished, we iterate over the *Relations* and for each one we use *PartitionMap* to discover which closeness level must be assigned to the *Relation* by comparing the distance with the ranges.

4.1.4. Tile model example

The graph in Figure 9 is the Tile model derived from the example window shown in Figure 1. Tiles (nodes) have been represented with ellipses that include the name of the widget, while relations (edges) have been represented with arrows with three attributes: the Allen relation for the X axis (*xAllenInterval*), the Allen relation for the Y axis (*yAllenInterval*), and the *closenessLevel*. The coordinates and dimension of the nodes have been omitted.

Since all the distances between the nodes are more or less similar, the clustering algorithm groups all the distances in just one cluster. This unique group is assigned the closeness level of 1. As mentioned in Section 4.1.2, the comparisons of the positions take into account a certain amount of margin, which is the reason why the *xAl-*

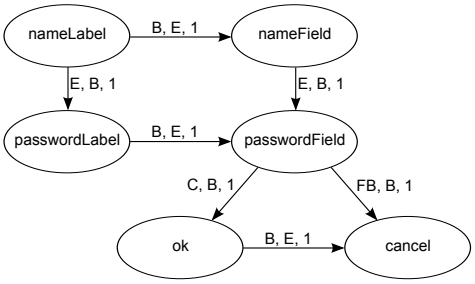

Figure 9: Graph representation of the login window example. *B=BEFORE, E=EQUALS, C=CONTAINS, FB=FINISHED_BY*

lenInterval of the relation between *nameField* and *passwordField* is *EQUALS* although the projection of the coordinates in the X axis is not exactly the same for both widgets.

4.2. Inferring a high-level layout

In this section we present the algorithm that we have defined to discover the structure of the layout in terms of layout managers (step 4 in Figure 4). First, we describe the patterns used to detect each of the layout managers introduced in Section 3.2. The algorithm is presented in two steps: we firstly provide an overview of the general structure of our algorithm, and we then go on to explain it in detail. An example with which to illustrate how the algorithm works is presented afterwards. Finally, we compare the algorithm with our previous work by analysing the new results obtained for a previous case study.

4.2.1. The layout patterns

In this section we describe the layout patterns used to detect the predefined layout types that were introduced in Section 3.2. It is worth highlighting that the inference algorithm is not restricted to this particular layout set but will work with any subset of the available layout types.

(Horizontal / Vertical) *FlowLayout*: selects a sequence of nodes that are connected by only one outgoing edge with the *xRelation* / *yRelation* equal to *BEFORE* or *MEETS*.

BorderLayout: searches for subgraphs that match all or some of the five areas of a star topology: top, bottom, left, right, centre. The patterns currently supported for the *BorderLayout* can be seen in Figure 10. The areas are detected by considering not only the edges of the graph, but also the relative distances to the container. For the *top*, *bottom*, *left* and *right* areas, there must be a distance that is lower than a certain value (15% by default) from the container bounds. When detecting a BorderLayout with only the top-bottom areas or left-right areas, the distance between areas relative to the container size must be greater than a value (20% by default) regarding the container bounds.

GridLayout: represents a rectangular grid topology of $n \times m$ nodes. It starts from a given node (see Figure 11(a)) and matches all the nodes on the right (see Figure 11(b)) that are connected by a single edge. It then recursively matches all the nodes from top to bottom that are likewise connected by a single edge (Figure 11(c) is a partial match, but not a valid final match). The final match is eventually the biggest subset of the matched nodes which represent a rectangular grid (in Figure 11(d) the final match is the 3×2 matched grid). There is a constraint that the nodes inside the grid cannot contain edges whose target node is outside the grid, only the border nodes of the grid are allowed to have connections to the nodes outside the grid. Additionally, for a *GridLayout* to be matched, the closeness level of all the edges must be the same.

FormLayout: this is a pattern that has been devised to arrange *SingleWidgets*, not *Containers*. The pattern firstly detects a vertical *FlowLayout* composed of a list of (more than one) horizontal *FlowLayouts*. Secondly, it is necessary to check that at least two of the elements are vertically aligned. The widgets shown in the example in Figure 1 match the *FormLayout*. This pattern searches for *alignment marks*, which are imaginary vertical lines to which some of the widgets are aligned. In the example, we have one alignment mark between *passwordLabel* and *nameField*, and another on the right-hand border of the *cancel* button. These alignment marks are later used to define the bounds of the *AlignedColumns* (see Layout model in Figure 3). For example, *nameLabel* and *passwordLabel* would form one *AlignedColumn*, while the remaining widgets would form another. Not all the widget types are permitted in a *FormLayout*, but only those typically found in a form (e.g. *ComboBoxes* and *CheckBoxes* are permitted, but *ImageContainers* are not).

A special layout is defined in the metamodel, the *CustomLayout*. As indicated in Section 3.2, this is not actually a layout, but it rather signifies that no combination of the selected layout patterns can be applied to the original graph to attain a solution. When this occurs, developers are responsible for programming the layout by hand. For example, the distribution of wid-

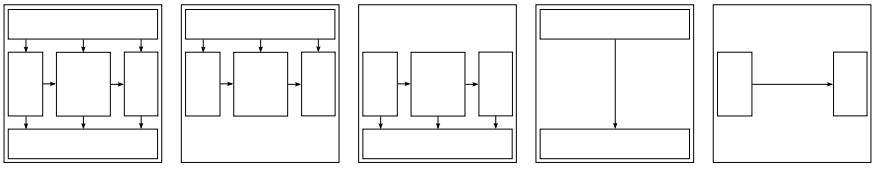

Figure 10: Border layout supported patterns.

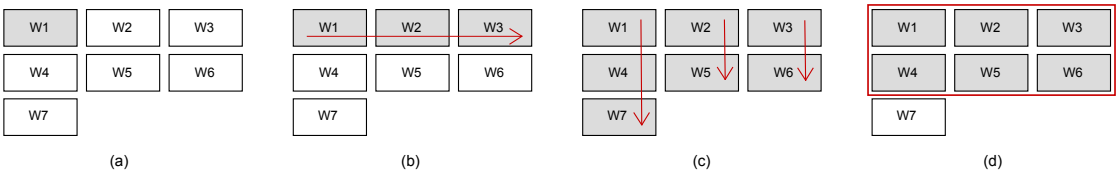

Figure 11: Grid layout matching example.

gets shown in Figure 12(a) will be discarded by the pattern matching engine because the merging of any two of these nodes would cause the area of these nodes to overlap with other nodes. Another example is the graph in Figure 12(b), which will not match any of the predefined patterns because they are disjoint tiles (none of them is either on the right or below the other one). In these two cases, a *CustomLayout* would be generated.

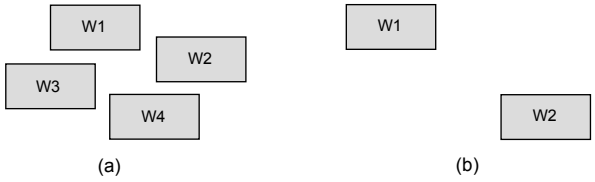

Figure 12: Examples of widgets that do not match any pattern

It is worth noting that some patterns such as the *BorderLayout* are more likely to be used when arranging containers, whereas other layouts such as the *FormLayout* are devised to work with single widgets. *FlowLayout* (vertical or horizontal) is the most general layout and can be used for both, containers and single widgets. *GridLayouts* can also work in both cases.

4.2.2. Overview of the layout inference algorithm

We shall first provide motivation for the use of our algorithm based on the view in Figure 1. Let us also suppose that the set of layout patterns is formed of Horizontal Layout (*HFlow*, represented by dashed boxes), Vertical Layout (*VFlow*, represented by dotted boxes) and Form Layout. There are three possibilities to arrange the widgets of the view depending on the order in which the layout patterns are applied. If we start searching for horizontal flows of elements (see Figure 13), we will find three matches: {*nameLabel, nameField*}, {*passwordLabel, passwordField*}, and {*ok, cancel*}, signifying that the view can be reduced to three composite elements (one for each match) which now match a vertical flow of elements. At this point, we would have a solution, as all the widgets in the view have been matched. There is a second alternative solution (see Figure 14), which is obtained if we start searching for vertical flows. In this case, two matches are found {*nameLabel, passwordLabel*}, and {*nameField, passwordField*}, which then can be joined by a horizontal flow, as well as {*ok, cancel*}. All these nodes can be finally matched by a vertical flow. A third solution can be obtained by applying a Form layout pattern, in this case all the widgets would fit this pattern in a single match.

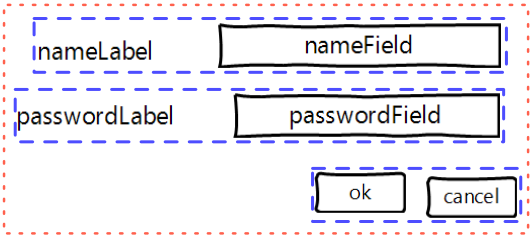

Figure 13: layout solution {*HFlow, VFlow*} for the window in Figure 1.

This example illustrates some key ideas to be considered in the design of an algorithm for inferring a high-level layout: i) a solution will be composed of a sequence of layout patterns, which expresses the order in which patterns must be applied to reach that solution, ii) when several nodes match a pattern, these elements can be replaced by a single node which represents the

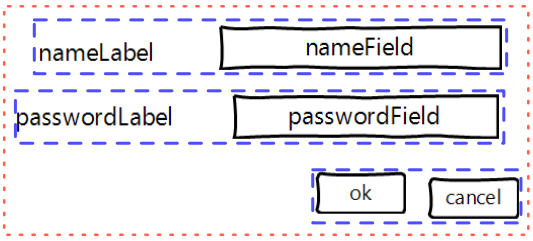

Figure 14: Layout solution {*VFlow, HFlow, VFlow*} for the window in Figure 1.

matched layout, and the process of discovering composite layouts can continue from that, iii) several alternative solutions are possible, so we need a fitness function that indicates how good the solution is.

Bearing these ideas in mind, we have devised a layout inference algorithm which applies a graph rewriting strategy to the Tile model in order to discover a set of layout solutions (i.e., the Layout model) for an input view. The algorithm, sketched in the following pseudocode, explores the search space using a trie tree to keep the partial solutions obtained in the search process implemented as an iterative backtracking approach.

Algorithm 2 Sketch of the layout inference algorithm

```
TrieTree = "create trie tree with root node"
"mark root node as unexplored"
while "there are unexplored nodes" do
    Actual = "next node"
    while "no layout pattern is matched and
           there are unexplored closeness levels" do
        for each layout pattern do
            "match pattern against current graph held in Actual"
            if "there is a match" then
                "perform graph rewriting"
                "create new node and update TrieTree"
                "mark the node as unexplored"
            end if
        end for
        "Increase closeness level"
    end while
end while
```

The algorithm begins matching the available layout patterns against the original graph, stored in the root node. For each layout pattern that is successfully matched we rewrite the graph by reducing the matched nodes into a single node that represents the matched layout. The rewritten graph, which is a copy of the original graph, is stored in a new node of the tree, which is marked as unexplored. The process continues until all the search space has been explored. This search must also take into account the closeness levels assigned to each widget. Thus, we try to match the layout patterns at level 1 (which means that we just take the edges at level 1). If none of the layout patterns achieve a match, then we try at closeness level 2 (which means that only the nodes tagged with levels 1 or 2 are considered), and so forth until there is a match or there are no more closeness levels. The end of the process is a trie tree in which the paths from the root to the leaves represent the sequences of layouts that lead to solutions. For instance, the solution trie tree that is created during the exploration of the graph in Figure 9 can be seen in Figure 15. The nodes refer the graph at that point of the exploration, and the edges represent the layout pattern that has been applied at a certain closeness level. Nodes with a stop icon represent wrong paths, and are not actually stored in the tree. Solution nodes are leaf nodes containing a a graph with a single node. A solution is defined by means of a path from the root node to a solution node, and the fitness value for the solution is stored in that solution node.

The trie tree constructed by the algorithm contains all feasible layout compositions using the available layout types, except for those that are beyond a closeness level for which a solution has already been found (i.e., solutions at closeness level $i + 1$ are not tested if a feasible solution is found at closeness level i). The rationale is to reduce the search space by prioritising layout compositions in which widgets are as close as possible. Reasoning by induction it can be shown that the algorithm proceeds by generating partial solutions attached to the nodes of the trie tree, so that a specific solution is a sequence of layouts, which can be recovered from the trie by a traversal from the root to a given leave node (a solution node).

- *Initialization*. At the beginning of the algorithm the trie contains only the root node (the complete graph), which is unexplored.
- *Base case (sequences of size 1)*. m layout sequences of size 1 are generated, according to the layout patterns matched against the complete graph. The generated sequences are represented as children of the root node. These new nodes are scheduled to be processed in the following steps of the algorithm, so that the next m iterations precisely extend these layout sequences to generate new layout sequences of size 2 (if there are matches).
- *General case (sequences of size n)*. At each step, the algorithm processes a trie node generated in a previous step, which represents a sequence of size $n - 1$. The layout patterns are matched against

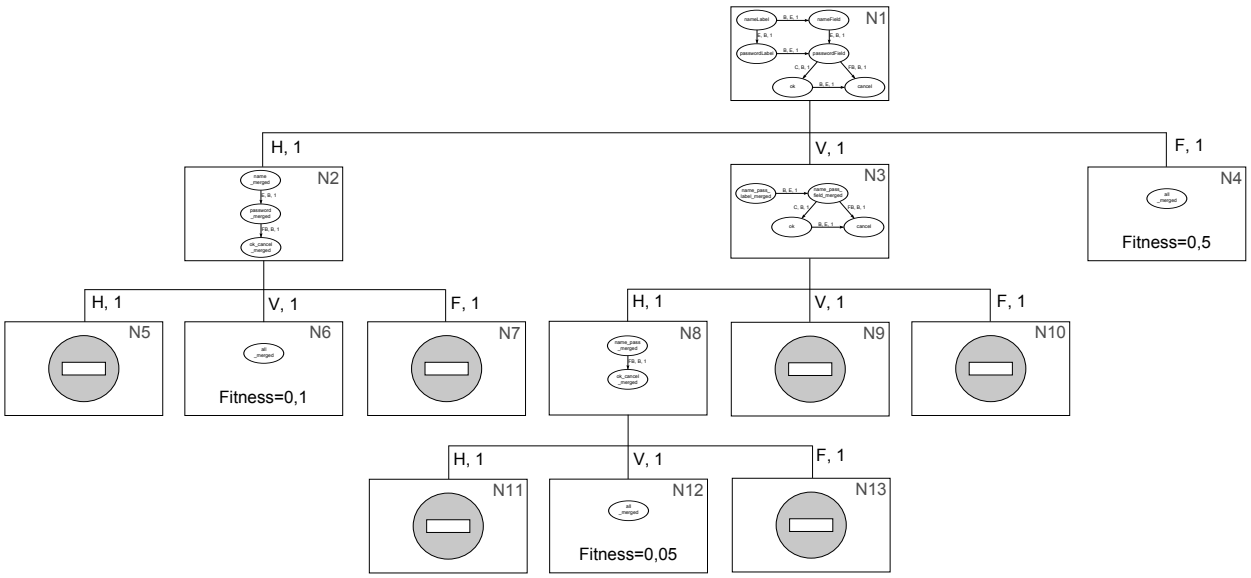

Figure 15: Solution trie tree for the login window example. (Nodes with the 'no way' sign are not stored)

the reduced graph held by the currently processed node, thus generating *r* new nodes. Each new node represents a partial solution as a sequence of layout compositions of size *n*.

- *Termination*. The algorithm always terminates since new sequences are generated based on a previously generated sequence, and it holds that for every branch of the trie either no more matches are found at certain depth or the graph has completely been reduced (i.e., a solution has been found).

4.2.3. A detailed description of the algorithm

Algorithm 3 is a more detailed version of the layout inference algorithm presented above. In this section we shall provide a detailed explanation of how this algorithm works, by dealing with the main design elements that the general schema exposes: input and output, iteration, pattern matching process, rewriting of the graph, creating a Layout model from the set of solutions, and the assessment of the layouts.

Input and output.

Given a Tile model, the algorithm is executed for each *WidgetNode* associated with a *Container* of the view. The algorithm receives two other inputs in addition to the container: the set of layout patterns to be used, and the number of closeness levels that appear in the relations of the graph. The layout inference is presented in Algorithm 3 as a function *InferLayout* with three arguments: *cNode* (a *WidgetNode*), *layoutSet* (a set of types of layouts that can participate in the solution) and *nCLevels* (number of closeness levels).

The algorithm generates a trie tree containing the result of the search. The set of alternative solution sequences can be obtained by taking the set of paths from the root node to the solution nodes (i.e. trie nodes whose graph contains only one node).

Backtracking traversal.

As we outlined in the overview of the algorithm, we use an iterative backtracking backed up with a trie tree structure to explore the search space, which is detailed in this section. Each node of our trie tree represents a layout composition described by an ordered sequence of pairs <*layout type, closeness level*> that are applied to the original match and can lead to a potential solution. The nodes store the graph that results from applying that sequence. An edge between two nodes (parent and child nodes) explicitly indicate the layout and closeness level that must be applied to reach the child node from the parent node. For example, node *N8* in Figure 15 contains the graph that is obtained after applying the vertical flow layout and horizontal flow layout at closeness level 1 in this order.

The nodes are iterated in a breadth-first fashion by means of a *queue*, in such a way that when there is a successful matching, we store the new, unexplored node at the end of the *queue*. In every iteration, the node in

Algorithm 3 Layout inference algorithm

```
1:  function InferLayout(cNode, LayoutSet, nCLevels): TrieTree
2:      TrieTree ← createTrie()
3:      rootNode ← createTrieNode(cNode)
4:      TrieTree.setRootNode(rootNode)
5:      Queue ← {rootNode}
6:
7:      while ¬isEmpty(Queue) do
8:          pNode ← Queue.dequeue()
9:          closeness ← 1
10:         genLevel ← true
11:
12:         while genLevel ∧ closeness ≤ nCLevels do
13:             for all layoutType ∈ LayoutSet do
14:                 pattern ← getLayoutPattern(layoutType)
15:                 Matches ←
                        match(pNode.graph, pattern, closeness)
16:                 if ¬isEmpty(Matches) then
17:                     graph ← copy(pNode.graph)
18:                     for all match ∈ Matches do
19:                         mergeNodes(graph, match, pattern)
20:                     end for
21:                     tNode ← createTrieNode(graph)
22:                     pNode.addChild(tNode, pattern, closeness)
23:                     genLevel ← false
24:                     if graph.size = 1 then
25:                         layout ← createLayout(graph)
26:                         layout.fitness ← fitness(layout)
27:                         tNode.layout ← layout
28:                     else
29:                         Queue.enqueue(tNode)
30:                     end if
31:                 end if
32:             end for
33:
34:             closeness ← closeness + 1
35:         end while
36:
37:     end while
38: end function
```

the first position of the *queue* is retrieved (line 8), and when there are no more branches to explore, the *queue* will be empty and the loop finishes (line 7).

The loops in lines 12 and 13 are used to iterate over the closeness levels and layout types. For a given node we try to match the layout patterns at the lowest closeness level (level 1), and if there is no matching, we try each layout pattern at level 2 (i.e., we will take all the edges tagged with levels 1 or 2), until there is a match or there are no more closeness levels in the graph. The *genLevel* variable controls if there has been a match of any of the layout patterns, so if it is set to *true* it means that there has been no match and the next closeness level must be tried. The iteration stops when there is a layout pattern that succesfully match (genLevel will be *false*) or all the layouts have been tried at the different closeness levels (*closeness* is greater than *nCLevels*).

Matching patterns.

The matching of every layout pattern is accomplished in line 15. Algorithm 4 shows how the pattern matching engine works. The *match()* function has tree arguments: the current graph (*graph*), the maximum value of closeness level (*closenessLimit*), and the layout pattern to be tried out (*pattern*).

Algorithm 4 Pattern matching engine algorithm

```
1:  function Match(graph, pattern, closenessLimit): Matches
2:      Matches ← {}
3:
4:      for all node ∈ graph.Nodes do
5:          match ← pattern.try(node, closenessLimit)
6:          if ∃ match ∧ ¬match.partOf(Matches) then
7:
8:              for all m ∈ Matches do
9:                  if match.contains(m) then
10:                     Matches.remove(m)
11:                 end if
12:             end for
13:
14:             NonMatchedNodes ← graph.minus(match)
15:             if ¬match.overlaps(NonMatchedNodes) then
16:                 Matches.add(match)
17:             end if
18:         end if
19:     end for
20:
21:     for all match1, match2 ∈ Matches do
22:         if intersection(match1, match2) ≠ ∅ then
23:             Submatches ← splitIntersection(match1, match2)
24:             replace(Matches, match1, match2, SubMatches)
25:         end if
26:     end for
27: end function
```

The pattern matching engine tries to match the pattern

against the graph starting from each node, because the starting node of a match cannot be determined beforehand (line 5). Not all the edges are considered when applying the pattern matching, but only those with a closeness level that is equals or less than the *closenessLimit* value (the rest of edges are ignored).

A consequence of matching the patterns on every node is that there may be matches that are part of other matches. In this case, the pattern matching engine returns the longest match. Therefore, when a new match is detected, we check that the rest of matching is not included in the new one (lines 8 to 12).

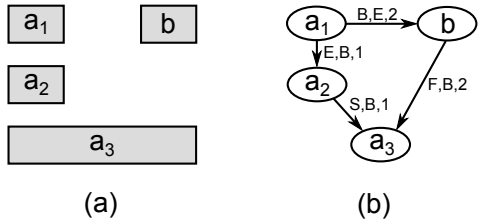

Figure 16: Example of non-valid match for the Vertical Flow Layout pattern.

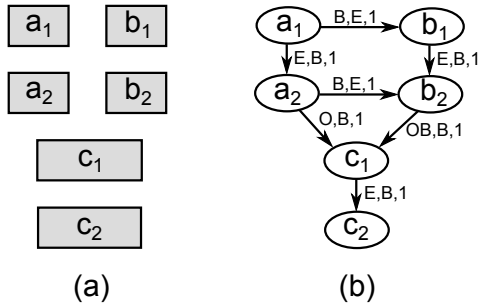

Figure 17: Example of match split for the Vertical Flow Layout pattern.

On the other hand, not all the matches performed by the matching engine are valid. There are two constraints that must be ensured: i) the area delimited by the matched nodes does not enter the area occupied by another node outside the match, and ii) no nodes are shared by different matches. In order to explain the first constraint (lines 14 to 17), let us consider the graph in Figure 16(b) which corresponds to the layout of widgets represented in Figure 16(a). In this example, if the matching engine were to attempt to match a vertical column of nodes (Vertical Flow Layout), it would perform the following match: $M_1 = \{a_1, a_2, a_3\}$. The edges $e_1(a_1, b)$ and $e_2(b, a_3)$ would be tagged with level 2, which is a higher closeness level than the edges $e_3(a_1, a_2)$ and $e_4(a_2, a_3)$ (level 1), and this would lead to the pattern matching engine to ignore the edges $e_1(a_1, b)$ and $e_2(b, a_3)$, and b could therefore no longer be matched. As can be seen, the rectangular area composed of the nodes of the match would enter the area taken up by b. In order to avoid conflict, the match is discarded.

The second constraint ensures that disjoint matches are obtained (lines 21 to 26). When we have nodes that are shared by two or more matches, we convert the shared nodes into a new match, and we remove these nodes from the remaining matches, thus obtaining two or more new matches. For instance, if we match a vertical column of nodes (Vertical Flow Layout) against the graph presented in Figure 17(b) that reflects the layout of Figure 17(a), then we will have two matches: $M_1 = \{a_1, a_2, c_1, c_2\}$ and $M_2 = \{b_1, b_2, c_1, c_2\}$. However, $\{c_1, c_2\}$ are conflicting nodes since they are shared by both matches. We therefore split the two matches into three matches, namely $M'_1 = \{a_1, a_2\}$, $M'_2 = \{b_1, b_2\}$ and $M'_3 = \{c_1, c_2\}$. When certain matches are split it is necessary to check that every submatch still fits the layout pattern, otherwise it is discarded.

Graph rewriting.

When there are matches for a pattern on the current graph (line 16 in Algorithm 3), the nodes of each match are merged into one node by applying a graph rewriting process (*mergeNodes()* in line 19). This works as follows:

- A new *LayoutNode* is created, which will represent the joining of the match. The new node is marked with the layout type (*type* attribute) that has been applied.
- All the matched nodes are removed from the original graph and included in the new node as children. The coordinates of the new node represent the area that contains all its children.
- All the edges between a pair of matched nodes (which are now children of the new node) are kept.
- All the edges from a non-matched node that starts or ends in a matched node now refer to the new node.
- If there are more than two edges between the new node and another non-matched node, then the edges are replaced with a new edge and the Allen relations are recalculated. The closeness level is the minimum level of the replaced edges.

After merging the graph nodes, the resulting graph is stored as a child of the current node *pNode* in the layout trie tree, setting the layout pattern and the closeness level that has been applied in the parent–child edge. Note that if there are matches, the genLevel flag is disabled for not trying the layout patterns at the next closeness level. It is also worth noting that if there are no layout pattern matches, no new node is added to the queue.

Creating the new layout.

This is done when the number of remaining nodes of the rewritten graph is 1, and thus a solution has been found (line 24, Algorithm 3). The new layout hierarchy is created based on the information of the rewritten graphs obtained following the path from the solution leave node to the root node (that is, the sequence of matched layout patterns).

To make the explanation clearer, in the following we sometimes say '*Widgets*' or '*Container*' when we are actually referring to 'the *LayoutElement* associated with that *Widget* or *Container*'.

The spacing and sizing properties are set (step 5 in Figure 4) for each *Layout*. *hSize* and *vSize* are the horizontal and vertical percentages of space occupied by the *Widgets* in comparison to their *Container*. *Gaps* (either horizontal or vertical) are created for each pair of adjacent *Widgets*. *Margins* are calculated for the *Layouts* (intermediate layouts or otherwise) that are children of a *Layout* associated with a *Container*.

The *createLayout()* (line 25) function also represents the alignment in an explicit manner (step 6 in Figure 4). *hAlignment* and *vAlignment* represent the horizontal and vertical alignment regarding the area of the enclosing layout, i.e. the minimum area that is large enough to contain all the widgets of the parent layout. For *Layouts* or *ElementNodes* nested in a horizontal *FlowLayout*, only *vAlignment* is set (the horizontal position is controlled by the layout manager) except in the case of there being one horizontal *FlowLayout* inside another one, when *hAlignment* is also set. Similarly, for *Layouts* or *ElementNodes* nested in a vertical *FlowLayout*, only *hAlignment* is set but only in the case of one vertical *FlowLayout* being nested in another. The value used to decide whether the bound of an element (top, bottom, left or right) is aligned is 15% by default. For example, if we have a horizontal *FlowLayout* associated with a container whose width is 100 pixels, and it contains an *ElementNode* associated with a label with coordinates $(10, 20)$, then $hAlignment = LEFT$ for the *ElementNode* because $10/100 < 0.15$.

Specific attributes must be initialised for some of the layouts defined in the Layout model. For instance, for a *BorderLayout* the nodes that correspond to the predefined areas (top, bottom, left, right, centre) are set by analysing the incoming and outgoing relations of that node. A *FormLayout* also has its own attributes. The nodes that comprise a *FormLayout* are analysed to identify the vertical alignment marks, which are distances in the X-axis that coincide (with some margin of tolerance) with the left or right bound of at least two nodes. A mark is represented as a percentage of the relative distance to the left bound of the *Container*. When the alignment marks have been detected, the nodes can be classified in the *AlignedColumns*, which are defined by contiguous alignment marks.

Assessing the new layout.

The algorithm generates several layout compositions that can be used to express the design of the original view. However, if we intend to use our algorithm in an automated tool for generating final GUIs (the case study in Section 6 will show an example of this), it will be necessary to choose one of them. Furthermore, according to certain criteria some solutions may be better than others. We consider the *best solution* in a solution set to be the layout composition (which is a layout tree) that best fits the following three criteria: i) it is made up of the minimum number of layouts that are required to represent the design, ii) it minimises the depth of the layout tree, and iii) if possible, specific layouts (*GridLayout*, *BorderLayout*) are used instead of generic layouts (*FlowLayout*, *FormLayout*).

The idea behind criteria i) and ii) is to keep the layout as simple and homogeneous as possible. This is useful as regards improving the maintenance of the views, and in a mobile design (e.g. Android UI design) it is important to keep the layout hierarchy as shallow as possible because it makes the layout draw faster if it has fewer nested layouts. The reason for criterion iii) is that some types of layouts express the design of the view better than others. For example, if we had a view structured in 2x3 parts, a *GridLayout* would express that structure more clearly than a vertical *FlowLayout* composed of three horizontal *FlowLayouts*.

Each solution of the algorithm is therefore assessed by using a fitness function (line 26) and is assigned a fitness value that is calculated using the following formula:

$$fitness = \frac{1}{\sum_{i=1}^{n} w_i * (d_{max} - d_i + 1)}$$

where n is the total number of layouts in the solution

represented by the layout tree, w_i is the weight of the i-th layout, d_{max} is the depth of the layout tree and d_i is the depth of the i-th layout. The weight of a layout is obtained as follows:

- For each *FlowLayout* or *FormLayout* we add 2, but not in the case of a *FlowLayout* nested in a *FormLayout* which is ignored (because it is part of the *FormLayout* and should not be counted twice).
- For each *GridLayout* or *BorderLayout* we add 1.

The fitness function used has the following properties: it has a value between 0 and 1, and given two solutions, it will return a higher value for the best of them; it tends towards zero as we increase the number of layouts of the solution (first criterion); deeper layouts are penalised in favour of shallow layouts (second criterion); *BorderLayout* and *GridLayout* obtain a better score than *FlowLayout* and *FormLayout* (third criterion).

4.2.4. Algorithm configuration

The layout inference algorithm depends on several parameters that can be configured, which are:

Layout types: the subset of layout types that will be used in the layout inference algorithm. All of them are selected by default.

Maximum closeness deviation: the maximum standard deviation of the distances allowed in every cluster. The smaller the value, more sensitive it is to distances between widgets. It is 15 by default.

Horizontal/vertical alignment margin: this represents the horizontal/vertical distance of a widget as regards the bounds of its container that leads the widget to be considered as aligned to that bound. It is 15% by default.

Comparison margin: expressed in pixels, this is used to provide some flexibility when performing comparisons between widgets (e.g. when detecting whether two adjacent widgets are aligned to the left). Its default value is 10 pixels.

When using the layout inference tool, these parameters have to be configured. The layouts to be used will depend on the target technology, as we will typically select those layout types that are close to the layout set used in the target technology. *Horizontal and vertical alignment margins* will usually remain unaltered. *Closeness cluster deviation* and *comparison margin* may in some cases require a litte manual tuning in some cases when the interfaces generated do not resemble the original ones or if they are not resized properly. In order to tune these parameters, we recommend testing three values: the default value, one value above the default value, and one value below it. For example, we could maintain the default value of the *comparison margin*, which is 10, and we would try out a value of 12 for the *closeness clustering deviation*, followed by a value of 18. Values for the *comparison margin* outside the range of between 3 and 20 do not obtain good results. Similarly, the *closeness clustering deviation* will be in the range of between 5 and 30.

4.2.5. Comparison of the inference algorithm with our previous work

The algorithm proposed in this paper can be applied to the windows created with the RAD applications described in our previous work [15]. We shall now compare the exploratory layout inference algorithm described in this paper, with the greedy algorithm proposed in such previous work. We shall use the first case study extracted from [15], which consisted of a layout-preserving migration of a business management application aimed to manage the research projects at Spanish universities, and which was developed using Oracle Forms 6, a well-known RAD environment.

We have applied the algorithm to a representative subset of Oracle Forms windows (30 out of the 107): 6 of large complexity, 5 of medium complexity and 19 of small complexity, such that the ratio of windows in the original application is maintained.

Table 2 shows the results of the evaluation according to the same metrics that were used in [15], namely:

- *Parts laid out OK*. For a distinguishable part of the window to be correct, it must contain the same widgets and it must be located in the same place as the original window.
- *Widgets laid out OK*. The widgets within each part are analysed by counting which widgets are located in the right place with regard to the container part and other widgets, and also taking into account their alignment.

It is first important to note that we had already obtained reasonably good results when using the greedy approach, and the improvement of the second approach cannot therefore be significant. This is owing to the fact

	Large (>60)	Medium (20 - 60)	Small (<20)	Total
Windows of each type (out of the total)	19.63%	16.82%	63.55%	100%
Parts laid out OK (greedy)	83.24%	98.06%	100.00%	96.38%
Parts laid out OK (exploratory)	95.24%	100.00%	100.00%	99.07%
Widgets laid out OK (greedy)	87.14%	85.61%	88.10%	87.50%
Widgets laid out OK (exploratory)	91.75%	94.80%	98.00%	97.29%

Table 2: Evaluation of the exploratory algorithm and the greedy approach in [15].

that in most cases, the GUIs created with RAD applications were often created using some systematic process, for example using a vertical flow composed of horizontal flows of widgets, which perfectly fit our predefined layouts, and the limitations of the greedy approach did not therefore have a great impact on these views.

In relation to the layout of the parts, there has been an improvement for large windows (12%), while in the case of medium windows 100% have been laid out OK (98% with the greedy version).

The percentages of widgets laid out in the exploratory version have also improved considerably: 4.61% for large windows, 9.19% for medium-size windows and 9.90% for small windows. In the greedy version, many simple (small complexity) windows were not perfectly replicated because the widgets could only be aligned with the container region, and gaps were not explicitly indicated. In the exploratory version, widgets can be aligned with regard to the containing layout and gaps are explicitly expressed. Moreover, *FormLayout* considers horizontal alignment which also helps to indicate a suitable alignment, and the widgets therefore match the appearance of the original window to a great extent.

With regard to the widgets laid out in medium and large windows, in the greedy version the layout managers defined could not be composed, signifying that complex layouts could not be properly captured (non-regular layout detection problem). Additionally, in some cases developers made full use of the empty space in large windows, which led to crammed and uncommon layouts that were wrongly recognised by the greedy approach. Since the exploratory approach allows layout nesting and explores different combinations of elements, it is able to recognise more complex layouts that the greedy approach. However, the improvement does not attain a success rate of the 100%, the reason being that, in many cases, the alternative solution that is chosen as the best alternative does not obtain a perfect layout (fitness function problem). This problem will be explained in depth in Section 6.2.

In view of the results of the comparison of both approaches, we conclude that the exploratory approach is better than the greedy approach. The latter obtained good results with RAD applications owing to the type of windows found in these applications, which are in most cases, vertical flows of horizontal lines of widgets, typically clearly segmented using frames. However, when widgets were placed in arbitrary locations or they did not fit the predefined layouts, no solution was output. On the contrary, the exploratory approach not only worked better with RAD applications but given that it is able to identify layout compositions, it will also work with complex layouts for which the greedy approach simply fails.

5. Performance evaluation

In this section we shall show the results of the performance analysis of the layout inference algorithm that we have carried out. We have considered two common scenarios: all the widgets in one container (i.e. the view itself is the only container) and the widgets arranged in several containers.

In both cases, we have generated five views containing 20, 40, 60, 80 and 100 widgets, which are arranged in groups, and each group conforms to a layout type supported by our algorithm. The groups are randomly placed but close to other groups (such that there are no significant distances between widgets), with the additional constraint that one group cannot overlap another. Moreover, the analysis has been carried out for three to five layout types. For three layouts we have applied the *HFlow*, *VFlow* and *Form layout*, and we have then added the *Grid* and *Border* layouts in that order. We decided to include the *Form* layout in all cases because its pattern matching is the most complex, whereas the complexity of the other layout types is more or less similar.

Figure 18 shows the execution times (in seconds) in the case of there being no containers in the view (the view itself is the only container) and all the widgets being close. This is the worst case as the algorithm has to deal with a single graph containing all the widgets. Figure 19 shows the result when the view is split into containers, with each container consisting of up to 20 widgets. This is an average case for the algorithm. The tests have been run in an Intel Core i5 with 4GB RAM.

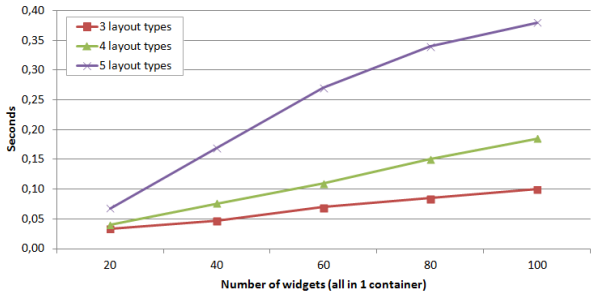

Figure 18: Execution time for widgets in a single container.

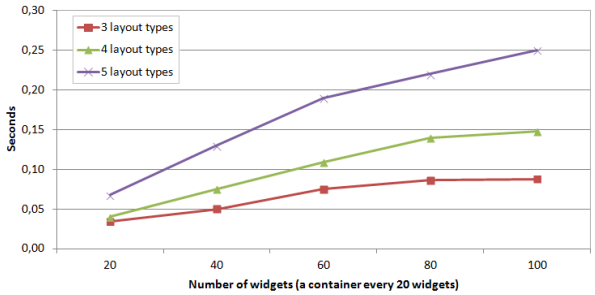

Figure 19: Execution time for widgets arranged in containers (one container for every 20 widgets).

Upon comparing both charts we can see that significantly better results are obtained when views are organised in containers. The difference in performance is owing to the fact that the pattern matching process is applied to containers, and performing pattern matching on the whole graph is therefore slower than applying it to several smaller subgraphs (one per container). Increasing the number of widgets only therefore implies analysing some more subgraphs. There would appear to be a polynomial growth in the execution time as the number of widgets increase.

The execution times measured for 5 layout types and 100 widgets are 0.38 seconds in the worst case and 0.25 seconds in the average case. These are reasonably acceptable for applications that work in batch mode (e.g. migrating many legacy GUIs together with a developer then examining and manually changing the results), but also for those that are interactive and require on-the-fly inference (e.g. a system that provides a visualisation of the final result as the user draws a mockup), provided that up to 40 widgets with no containers are drawn.

6. Case study: from Wireframes to fluid web interfaces

This section presents a case study that has been carried out to put into practice the layout inference approach described in the previous section. The goal of the case study is to automatically generate final GUI code from wireframes created with the WireframeSketcher tool. The transformation of a wireframe into a final user interface requires layout inference in order to generate the source code for a particular platform and GUI toolkit. Details about the implementation used in this case study can be found in the tool website: `http://www.modelum.es/guizmo`.

6.1. Context of the case study

In this subsection we shall present the technologies involved in the case study. More particularly, *wireframes* will be the source artefacts to which we shall apply the layout inference, and *web fluid interfaces* will be the output that we wish to generate.

6.1.1. Wireframes and fluid layouts

Designing GUIs is a crucial and complex task during software application development, which involves dealing with aspects such as functionality, accessibility and usability. An iterative process is normally applied in GUI design, in which several representations of the GUI are built at a different detail level, thus allowing users and developers to carry out experiments and discuss the structure and behaviour of the GUI. Three representations are frequently used: sketches, wireframes and mockups. Sketches are rapid, freehand drawings that show an initial design idea on the interface. Wireframes reflect how the contents are distributed on the screen (i.e. the layout of the widgets that represent the content). Mockups refine wireframes by adding details such as colours or images.

Wireframing tools commonly provide specific editors with which to create wireframes and mockups. There are also tools that can automatically generate final GUI code from wireframes or mockups. For example, Reify[3] generates web interfaces from wireframes created with the Balsamiq[4] wireframing tool. However, wireframe generation tools currently have significant limitations, as they are limited to certain types of layouts [13], they can only be applied to certain platforms such as web interfaces with CSS [16], or they are still in an immature state (like Reify).

[3] `http://www.smartclient.com/product/reify.jsp`
[4] `http://balsamiq.com/`

Wireframes do not have an explicit notion of layout, and widgets are rather dragged from a palette and placed in a particular position (which is sometimes almost arbitrary) on a canvas. Wireframes therefore only provide a *coordinate-based layout*. The transformation of wireframes into the eventual GUIs in modern platforms with explicit layout facilities poses the challenge of uncovering the implicit structure of the GUI in order to obtain an explicit representation of the layout. Although we have only used wireframes in the case study, the approach is also applicable to mockups.

In 2010 Ethan Marcotte coined the term Responsive Web Design[5] to a design philosophy aimed at crafting sites in order to provide an optimal viewing experience. This philosophy is made up of three basic principles: i) define fluid grids, ii) define flexible images, and iii) use CSS 3 media queries to change the style depending on the screen dimension. Both responsive interfaces and fluid layouts (although not necessarily grids) have since become popular.

In the case study, adaptive web interfaces (interfaces with a fluid layout) were generated by using ZK[6]. We did not address a fully-fledged responsive UI design since it implies the use of algorithms to rearrange the content. As our Layout model explicitly captures explicit information about the layout, this rearrangement is possible. This case study can thus be considered as a first step towards generating responsive web interfaces.

6.2. Evaluation of the approach

Our approach has been validated by conducting an experiment with users. 20 people working in the IT sector were asked to design a series of wireframes and then apply our layout inference tool in order to generate the source code of the final GUI. The participants were: 4 software developers, 4 software analysts, 2 project managers and 10 software engineering researchers, all of whom were familiarised with web and/or mobile interfaces.

6.2.1. Methodology

The methodology used was the following. Firstly, each person was provided with an explanatory document that s/he had to read carefully. This document explained the utility of wireframes, provided some instructions on how to use both the wireframing tool we chosen (WireframeSketcher) and our layout inference tool, and explained the task to be accomplished. After reading the document, each participant had to design 5 screens for an on-line bookstore application. These 5 screens were intended to be typical views of a web application that involve common design patterns such as master-detail or registration form (based on the evaluation presented in in [16]). The 5 screens that we requested are indicated below (the name of the view is indicated before the colon):

- *best*: it displays information about the best-sellers.
- *cart*: a shopping-cart view that shows the current state of the cart.
- *detail*: it shows detailed information about a selected book.
- *search*:it allows certain criteria to be searched for and the results of the query to be viewed
- *user*: it allows users to create a new user account.

Appendix A contains one wireframe of each of the types developed by the participants in the experience. The instructions did not specify any layout constraints, thus permitting the developers to choose the layout composition that they considered to be most appropriate for each type of content. The wireframes designed contained between 15 and 30 widgets in almost all cases.

Once the user had finished the screens, s/he was encouraged to apply our tool in order to generate the code of the final GUI and execute it to see what the view looked like. The participants were not permitted to modify the default values for the tool parametersin the first execution, but they could be altered if the result was not what the user initially expected.

The wireframes designed by the users were used to assess the approach in two ways. Firstly, each user was asked to fill in a questionnaire concerning the experience, in which s/he could indicate how good the generated view was and express whether the generated layout matched the idea s/he had in mind. The intention of this was to discover how useful the tool is from the developers point of view. Secondly, the participants were asked to submit the wireframes they had created in order to perform a quantitative analysis of the result of the layout inference.

6.2.2. Quantitative results

Table 3 shows the results of the evaluation (classified by view). *Visual resemblance* measures how well the GUI generated resembles the original wireframe (i.e.

[5] `http://alistapart.com/article/responsive-web-design`

[6] `http://www.zkoss.org/`

Screen	**best**	**cart**	**detail**	**search**	**user**	**TOTAL**
Visual resemblance	96.2%	97.6%	97.2%	99.1%	95.6%	97.4%
Parameter changes	30.0%	33.3%	30.0%	26.7%	36.7%	31.3%
Average number of layouts (best solution)	5.0	4.2	4.9	3.4	2.4	4.0
Average number of alternatives per view	2.6	4.1	3.3	3.1	4.2	3.4
Layout resizing	77.8%	84.9%	75.6%	90.5%	93.4%	84.4%

Table 3: Quantitative evaluation results.

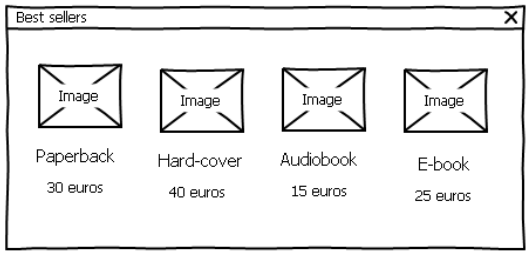

Figure 20: Example window.

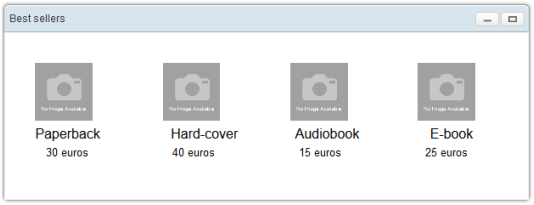

Figure 21: Example vertical-horizontal flow.

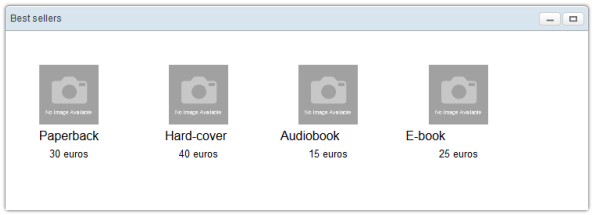

Figure 22: Example vertical-horizontal flow resized.

the accurary of the GUI generated). The number of generated widgets located in the same place are counted as being the corresponding widget in the original view. For example, Figure 20 shows a simple view that displays best sellers, while Figure 21 shows the view that has been generated from the former view. As will be observed, the the *Hard cover* and *Audiobook* labels appear to be misaligned with regard to the column, and the visual resemblance of this view is therefore $10/12 = 83\%$.

Our tool intentionally compares widgets with some degree of flexibility, signifying that minor misalignments are allowed, and widgets that are clearly misplaced or misaligned with regard to the original view are counted as errors. This is controlled by the *comparison margin* tool parameter explained in Section 4.2.4, whose default value is 10 pixels. The box plot in Figure 23 expresses how the accuracy is distributed in each mockup. In global terms, the windows generated have a high degree of accuracy (97%), and there is no significant difference between the different types of views (*search* obtained slightly better accuracy, which is not actually representative). This high accuracy is partly because if, at the first attempt, the GUI does not resemble the original wireframe we change the algorithm parameters in order to improve the result, as is explained below.

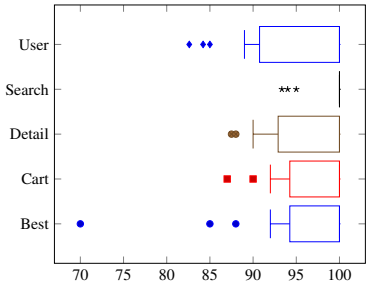

Figure 23: *Visual resemblance* for the different screens

Parameter changes expresses the percentage of views designed by the users that required changes to be made to the default values of the parameters in order to obtain a reasonable good GUI (high *visual resemblance*). As it can be seen, in many cases (31% of the views) parameters needed to be tuned and there were not remarkable differences between the different types of views. From the end-user perspective, this means that s/he would obtain a sufficiently good GUI without tuning the algorithm in 70% of cases. The process of manually tuning the parameter algorithm typically involves trying out 3 values (default value, lower value, upper value) for both the closeness cluster deviation and the comparison margin, which makes 9 possibilities. The average time that a developer needs to change these parameters, run the

tool and analyse the new view is 20 seconds, which results in 3 minutes per view. One of the current limitations of the approach, which is related to the maximum closeness cluster deviation parameter, will be explained in the following subsection.

The *average number of layouts* of the best solution counts the number of layout managers used in the best solution (i.e. the solution containing the highest fitness value). On average, a composition of 4 layout managers are required to completely define the layout of the views. A low average of layouts indicates that the best solution does not use unnecessary layouts but only those that are required (i.e. it is efficient in most cases).

The *average number of alternatives per view* represents the number of different layout compositions that are offered for each view on average. In our case we have an average of 3.4 alternatives per view.

The last row of the table (*layout resizing*) indicates whether the final GUI generated for the best solution is resized appropriately when tested. To continue with the previous example, Figure 22 shows the view generated in Figure 21 after having horizontally resized the window. In this case, the *Hard cover*, *Audiobook* and *E-book* labels are clearly misplaced when the screen is considerably resized, thus leading to a resizing rate of $9/12 = 75\%$. There are several alternative solutions for a given view which may at first glance appear to be equally valid, but which look completely different when the view is resized. A good layout solution arranges the widgets in such a way that their appearance is correct when they are resized.

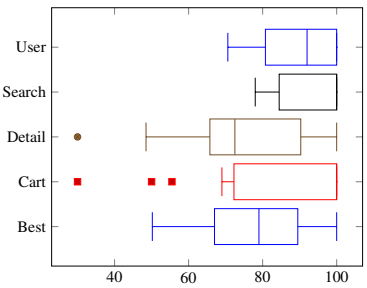

Figure 24: *Layout resizing* for the different screens

We have a success rate of 84,4% as regards view resizing, which means that there are around 15% of views for which the fitness function fails (i.e. it does not always select the best option for resizing). We have a slightly higher success rate for the *search* and *user* types of views, because most people used more or less standard form-like designs for these views, which fit our FormLayout rather than using complex combinations of other layout types. *Best*, *detail* and *cart* meanwhile obtained a lower success rate because developers formed less standard layouts. This is also reinforced by the box plot in Figure 24, which reflects that there is great variability in the accuracy for the mockup types with the worst success rate (*best* and *detail*). Using more complex fitness functions that not only take into account the number of layouts involved could result in improvements in the *best*, *cart* and *detail* types of views. The limitations of the current fitness function will be explored in greater depth in the following subsection.

6.2.3. User assessment

The users filled in a questionnaire containing five questions that summarised their experience. These were: 'Are the generated views as I expected?', 'Are the margins, gaps and alignment correct?', 'When resizing the windows, are the widgets resized appropriately?', 'Could the windows generated be used in a real application?', and 'Is the layout inference tool useful?'. The questions were graded using a 5-point Likert scale, and their results are shown in Table 4.

The vast majority of the users (85%) agree or totally agree that at first sight, the views generated resemble the original ones. These results are in line with the assessment that we tackled by manually inspecting the models and views, but we could have expected a higher rate in this question. The reason why users did not give a better mark to this question is owing to the fact that almost none of them (only 10%) changed the default parameters, so the algorithm did not always produce a very good result. If the users had tuned the parameters, a better score would probably have been achieved.

The results of 'Are the margins, gaps and alignment correct?' have a certain degree of similitude with those obtained in the previous question, since the views that are more or less similar to the original ones must have correct margins, gaps, and alignment.

With regard to resizing, 60% of the users are of the opinion (agree or totally agree) that the resizing behaviour is more or less suitable. As indicated previously, the resizing behaviour is not always suitable, which is partly related to the weakness of the fitness function. This will be explained in detail in the following subsection. The difference between the score of the quantitative evaluation (84% of success) and the score given by users is mainly owing to the fact that users did not tune the parameters, signifying that in many cases they found strange resizing.

With regard to the question 'Could the generated windows be used in a real application?' 65% of the users

Question	Strongly agree	Agree	NAND	Disagree	Strongly disagree
Are the generated views as I expected?	35% (7/20)	50% (10/20)	15% (3/20)	0% (0/20)	0% (0/20)
Are the margins, gaps and alignment correct?	25% (5/20)	45% (9/20)	25% (5/20)	5% (1/20)	0% (0/20)
When resizing the windows, are the widgets resized appropriately?	25% (5/20)	35% (7/20)	25% (5/20)	10% (2/20)	5% (1/20)
Could the generated windows be used in a real application?	25% (5/20)	40% (8/20)	25% (5/20)	5% (1/20)	5% (1/20)
Is the layout inference tool useful?	50% (10/20)	40% (8/20)	10% (2/20)	0% (0/20)	0% (0/20)

Table 4: User evaluation results. *(NAND = Neither agree nor disagree)*

agree, but others have some reservations, principally because resizing fails in some cases and users have to tune parameters that they do not feel are easy to change.

50% of the users find the tool extremely useful, and 40% think that the approach is useful. Despite the current limitations of the approach, developers believe that the tool is useful because they can reuse wireframes and save time when implementing the GUI of the application.

The questionnaire also included a text area in which users could write their suggestions if desired. A summary of their most interesting comments is shown as follows: many users found that some of the parameters of the algorithm (e.g. the cluster deviation) were not easy to understand; some of them complained that parameter tuning could be done automatically; a few suggested that users may be interested in specifying which widgets should be kept fixed (not resized).

6.2.4. Threats to validity

The threats to validity are classified in two main groups. *Threats to internal validity* refers to those decisions that may affect the results of the case study itself, while *threats to external validity* refers to circumstances that may affect the generalisation of the results.

The main threat to internal validity is that the comparison between the wireframes created by the subjects and the GUIs generated was made by the authors of this paper. In order to mitigate the risk of this comparison being driven by the expected outcome of the tool, the process was carried out by a colleage who was not involved in its implementation. Moreover, many participants did not change the default values of the algorithm but we did so in the evaluation. This could explain the discrepancies between the participant's opinions and our evaluation. Another threat is that all the participants are part of the IT sector, which implies that they already have knowledge of GUIs and layouts . Although we requested them not to think about the "the GUI to be generated", it is uncertain whether this was actually the case.

Regarding threats to external validity, one threat is that the GUIs selected for the case study may not have been sufficiently representative. When designing the case study we defined GUIs that represented common design patterns in web applications (similar to [16]) but that were not too large so that they would not be too demanding for the participants in the case study. In this respect, our results are limited to this type of applications, and cannot be extrapolated to e.g. creating GUIs for a videogame. The fact that all the participants are part of the IT sector is also a threat to external validity, since the results cannot be generalised to other types of users and scenarios, such as a client creating a mockup that is automatically converted into the final GUI.

6.2.5. Approach limitations

The current implementation of the approach has two limitations at present. The first limitation is related to the maximum closeness cluster deviation parameter, and the second limitation is the implementation of the fitness function.

Widget distance clustering.

As previously stated, the maximum cluster deviation parameter represents the maximum standard deviation of a group of distances that is admissible for them so they can all be considered similar. This parameter is not only used to perform flexible comparisons, but also drives the pattern matching phase by indicating which relations among the elements can be matched.

For example, in the fragment of the *Detail* view in Figure 25 there are two clearly different areas in the view: the form on the left and the right part composed of the *Cover* image and the *Enlarge* button. There should therefore be a layout for each part, and a layout that 'glues' both parts together. The designer that created the view left some empty space in the middle of the view on purpose, signifying that both parts can be distinguished. Given that the 'close' or 'far' concepts are subjective (they depend on human perception), we need the maximum closeness deviation parameter to be able to decide which widgets are close or far.

However, the closeness level between a pair of widgets may sometimes confuse the inference process. In

the example in Figure 25, the relation between the *Description* label and the *Author* label will have a higher closeness level (i.e. they are significantly more distant) than the *Description* label and the *Description* text area, and also higher than the *Description* text area and the *Author* text area. While this is strictly correct, the inference process should consider these relations as being equally close because they are part of a form and the user that created the view meant it to be one form, not two separate parts. In such cases, it may be necessary to carefully tune the maximum closeness deviation parameter in order to obtain a good result.

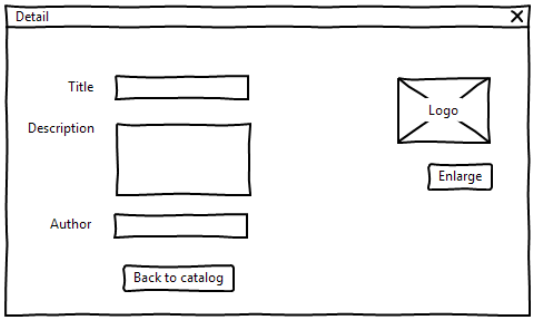

Figure 25: Example of the closeness problem.

Fitness function implementation.

Figure 20 shows a simplified *Best* view. Let us assume that we only wish to use the (vertical/horizontal) FlowLayout. This view can be laid out in two different ways (two layout alternatives), which are:

- *Alternative 1*: a horizontal FlowLayout composed of 4 vertical FlowLayouts (containing 3 widgets each).
- *Alternative 2*: a vertical FlowLayout composed of 3 horizontal FlowLayouts (containing 4 widgets each).

Alternative 1 will generate a view that will be similar to those shown in Figure 20, and which will ensure that the 4 columns maintain the same aspect ratio and alignment when resized. On the contrary, Alternative 2 leads to unaesthetic views when they are resized. Figure 21 shows the view generated for the second alternative, and Figure 22 shows it after resizing. These differences in the appearance of the GUI stem from the way that the information concerning the layout is expressed in each alternative. In the first alternative, we can specify that each widget is centred with regard to its column, but in the second, we have no direct means to specify the relative distances between the widgets in each row. We can see that some alternatives are better than others (and more particularly, that Alternative 1 is better than Alternative 2).

We therefore need to develop more complex fitness functions that not only take into account efficient layout compositions (in the sense of reducing the number of layouts that are nested) but also aesthetic criteria. More particularly, considering the homogeneity of the content of the layouts in combination with the current fitness function could lead to better results.

7. Related work

In this section we present several approaches and tools that perform some kind of inference from a GUI in which the layout is implicitly represented by means of coordinates.

In a previous work we proposed an algorithm whose objective was to infer the layout of Rapid Application Development environments (RADs) [15] [14]. Although the algorithm in question works fairly well for RAD applications, it is rather limited as regards its general application to other cases (e.g. it does not work well with mockups). This is principally because the layout inference was addressed by means of a few simple heuristics which did not allow composite layouts, which means that all the widgets in a container must fit a single layout (e.g. a grid layout). Another shortcoming included not taking advantage of alignment relationships between widgets and not grouping widgets according to their relative distances (i.e. relying on parts delimited by frames). In practice, this means that the algorithm works well when the GUI is "well structured" but is less predictable in more unconstrained scenarios. We have attempted to overcome these limitations by developing a brand-new algorithm which is far more elaborate, notably allowing layout compositions. Moreover, in the new inference algorithm the predefined layout set is configurable and provides several layout solutions, in contrast to the previous version of the algorithm which merely yielded a single layout solution or no solution if none of the predefined layouts fitted reasonably well. These improvements have been achieved through the use of an exploratory algorithm preceded by a clustering phase, whereas a greedy algorithm was proposed in the previous work.

Another related work in this area is that presented in [7], which uses a mathematical model (the *Auckland Layout Model*, ALM [8]) that is based on linear programming, to represent the GUI layout with a high ab-

straction level. It defines an algorithm that can be used to recover an ALM model from hard-coded layout information. The usefulness of the approach is illustrated by showing how to extend hard-coded GUIs to support dynamic layout adjustment. The ALM model uses an abstract concept called *tabstops* to divide the coordinate system into a grid of discrete positions, which are to certain extent related to the alignement marks that we use to create *AlignedColumns* in *FormLayouts*. The fundamental difference between this work and ours is that the ALM model defines constraints on the elements represented in mathematical expressions, whereas we use an object model that explicitly represents the layout using a tree of layout managers. Note that the ALM model does not capture the visual structures, but rather constraints on the single elements.

In [13] the authors propose an approach with which to generate a web interface from mockups. In this work, an intermediate representation of the GUI that is independent of the source and target technologies is presented (the Core Mockup metamodel), which is similar to our model used to represent the logical structure of the GUI (Structure model). Recognition of containment relationships of the widget is supported, which is to some extent related to our region identification. The main difference between both approaches is the selection of the layout types (layout managers) used to represent the layout of the views. In the approach in question, the user specifies one layout type (e.g. *GridBagLayout*), and there is a dedicated algorithm that fits the content in that layout. In our case, we simply select the layout types and the different layout hierarchies are automatically searched for and assessed. Therefore, such approach is tightly tied to specific layout managers and it only supports one layout manager at a time, whereas our approach is more flexible.

The work presented in [16] is related to the analysis of mockups. It suggests a method that can be used to systematically encode mockups drawn in a WYSIWYG editor in fluid and elastic layouts. The method first infers the layout as a hierarchy of horizontal and vertical boxes, and then encodes the layout with rules in HTML/CSS. A backtracking approach is used which shows a certain similarity to our algorithm, in that the elements are treated as rectangular boxes, and the nodes (single widgets or groups of widgets) are joined until all the elements have been joined in one node. This approach, in contrast to ours, greatly depends on the HTML/CSS layout system, and in fact they only take into account a horizontal/vertical flow of boxes (whereas ours supports a variety of layouts). What is more, some information is not explicitly represented. For example, information regarding gaps and alignment is not explicitly extracted, but the distance in pixels to the container box is stored for each element.

A well-known example of a GUI builder with layout generation facilities is the Swing GUI builder for the NetBeans IDE, formerly known as *Matisse*[7]. It is a fully-fledged design tool that generates code that perfectly fits the design. Developers use Matisse to build GUIs without explicitly indicating layout manager parameters (they are visually specified), and it generates code based on the *GroupLayout*, a layout manager which was intentionally introduced to work with IDEs. Matisse generates code for Java Swing, particularly based on the *GroupLayout*, and is tied to the NetBeans IDE, whereas our approach is intended to be platform-independent.

Reify is a tool that imports Balsamiq mockups and automatically transforms them into interactive applications backed by code that follows best practices. Like our approach, Reify infers the layout of the views, but is at present still a beta version (there is an online demo which does not generate fluid layouts and it also fails in most cases).

There are some other works that do not strictly deal with layout inference but are in some respects related to the topic. An approach with which to migrate Windows applications to Visual Basic .NET can be found in [6]. Its aim is to replicate the GUI's look & feel by means of mapping runtime objects into .NET objects, without performing a layout recovery.

In [5], the authors propose a pixel-based approach based on the real-time interpretation of the GUI in order to identify the hierarchical model of complex widgets. This information is then used to modify an existing GUI (e.g. to translate the text of the widgets) with independence from the interface implementation.

VAQUISTA [17] is a tool which performs the reverse engineering of web pages into XIML [12] models according to flexible heuristics, and requires user interaction during the reverse engineering process. In this case, the sources are web pages written in HTML4 which are laid out with tables, and the tool maps each table cell into a target element, signifying that no high-level layout is recovered, but simply the table layout is replicated in XIML.

In [3] an approach for extracting the web content structure based on the visual representation is proposed. This simulates how users understand web layout structure based on their visual perception. The approach is

[7] `https://netbeans.org/features/java/swing.html`

to a great extent based on the nature of the HTML code and cannot be applied to coordinated-based interfaces.

Some other related works propose the reengineering of web pages, particularly in order to adapt them to mobile devices. The following two works fall into this category. In [4] an approach with which to structure web pages in a two level hierarchy is presented, in such a way that if a user selects a part of the web page, this part will be displayed with the screen size like a zoom-in. In [2], a solution for generating dynamic web migratory interfaces is explained. The authors rely on the analysis of HTML tags in order to split the original web pages into regions that are transformed into web pages with hyperlinks between them. It is worth noting that UI reengineering approaches for web pages work on DOM trees, which are tree-based representations of the HTML code, in which the GUI structure is explicitly expressed by means of HTML tags.

8. Conclusions and future work

In this paper we have presented an algorithm and a graph-based data structure (the Tile model) with which to reverse engineer GUIs with a coordinate-based layout, in order to transform them into a representation based on layout managers that can be used to generate a final GUI based on good practices. The algorithm provides several alternative layout compositions, and also estimates which is the best option. Moreover, it allows widgets to be placed with some degree of misalignment and allows the set of layout managers that will be used when composing the layout to be selected.

We have also presented a case study in which we use our algorithm to infer the layout of GUIs created with a wireframing tool in order to generate fluid web interfaces. This case study revealed that the current approach is, in some cases, somewhat limited by the maximum deviation parameter used in distance clustering, and that the fitness function needs to take into account aesthetic criteria such as homogeneity in order to obtain solutions that are better adapted to different screen dimensions. On the whole, the results drawn from the evaluation show that our approach is, in most cases, able to perform a good layout inference (97% of the views are accurately reproduced, 84% of the views are appropriately resized) and 70% users are satisfied with the tool results.

The proposed approach provides automation of the layout inference process, as well as independence from the source and target technologies or platforms. The supporting tool is available as an Eclipse plug-in, and can be downloaded from `http://www.modelum.es/guizmo`.

As future work, we plan to improve the approach in several ways. We shall explore how to automate the tuning of the parameters of the algorithm, in order to reduce the users' burden, and we shall implement fitness functions based on the homogeneity of the content and other metrics based on human perception.

We shall analyse the impact of a graph rewriting tool such as *GrGen*[8] to deal with the layout pattern matching. We are particularly interested in specifying layout patterns by using a graph language and creating them in pluggable modules so that new patterns can be defined in a modular manner without recompiling the tool.

We shall improve the supporting tool with additional source and target technologies. We particularly intend to create fully-fledged responsive web pages with ZK and other responsive frameworks (e.g. the Semantic Grid System[9]) which will be tested in different devices with different screen sizes and resolutions. We shall also study alternatives that will allow us to automatically rearrange the layout for different sizes when there are images, in order to avoid scrollbars and image cluttering, thus permitting the content to be displayed smartly.

Acknowledgement

This work has been supported by the grant 15389/PI/10 of the Fundacion Seneca (Agency of Research and Technology of the Region of Murcia, Spain). We are also grateful to the anonymous reviewers for their insightful comments which have helped to improve the final version of this work.

References

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

[2] R. Bandelloni, G. Mori, F. Paternò, Dynamic generation of web migratory interfaces, in: MobileHCI '05: Proceedings of the 7th international conference on Human computer interaction with mobile devices & services, ACM, 2005, pp. 83–90.

[3] D. Cai, S. Yu, J.R. Wen, W.Y. Ma, VIPS: a Vision-based Page Segmentation Algorithm, Technical Report, Microsoft Research, 2003.

[4] Y. Chen, W.Y. Ma, H. Zhang, Detecting web page structure for adaptive viewing on small form factor devices, in: WWW, pp. 225–233.

[5] M. Dixon, D. Leventhal, J. Fogarty, Content and hierarchy in pixel-based methods for reverse engineering interface structure, in: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, CHI '11, ACM, 2011, pp. 969–978.

[8] `http://www.grgen.net`

[9] `http://www.semantic.gs`

[6] J. Gerdes, Jr, User interface migration of microsoft windows applications, J. Softw. Maint. Evol. 21 (2009) 171–187.
[7] C. Lutteroth, Automated reverse engineering of hard-coded gui layouts, in: Ninth Australasian User Interface Conference (AUIC 2008), volume 76, ACS, 2008, pp. 65–73.
[8] C. Lutteroth, R. Strandh, G. Weber, Domain specific high-level constraints for user interface layout, Constraints 13 (2008) 307–342.
[9] J.B. MacQueen, Some methods for classification and analysis of multivariate observations, in: L.M.L. Cam, J. Neyman (Eds.), Proc. of the fifth Berkeley Symposium on Mathematical Statistics and Probability, volume 1, University of California Press, 1967, pp. 281–297.
[10] J. Martin, Rapid application development, Macmillan Publishing Co., Inc., 1991.
[11] A. Memon, I. Banerjee, A. Nagarajan, Gui ripping: Reverse engineering of graphical user interfaces for testing, in: WCRE '03: Proceedings of the 10th Working Conference on Reverse Engineering, IEEE Computer Society, 2003, p. 260.
[12] A. Puerta, J. Eisenstein, Ximl: a common representation for interaction data, in: IUI '02: Proceedings of the 7th international conference on Intelligent user interfaces, ACM, 2002, pp. 214–215.
[13] J.M. Rivero, G. Rossi, J. Grigera, J. Burella, E.R. Luna, S. Gordillo, From mockups to user interface models: an extensible model driven approach, in: ICWE, ICWE'10, Springer-Verlag, 2010.
[14] O. Sánchez Ramón, J. Sánchez Cuadrado, J. García Molina, Model-driven reverse engineering of legacy graphical user interfaces, in: Proceedings of the IEEE/ACM international conference on Automated software engineering, ASE '10, ACM, 2010, pp. 147–150.
[15] O. Sánchez Ramón, J. Sánchez Cuadrado, J. García Molina, Model-driven reverse engineering of legacy graphical user interfaces, Automated Software Engineering 21 (2014) 147–186.
[16] N. Sinha, R. Karim, Compiling mockups to flexible uis, in: Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2013, ACM, 2013, pp. 312–322.
[17] J. Vanderdonckt, L. Bouillon, N. Souchon, Flexible reverse engineering of web pages with vaquista, in: WCRE '01: Proceedings of the Eighth Working Conference on Reverse Engineering (WCRE'01), IEEE Computer Society, 2001, p. 241.

Appendix A. Wireframe examples extracted from the case study

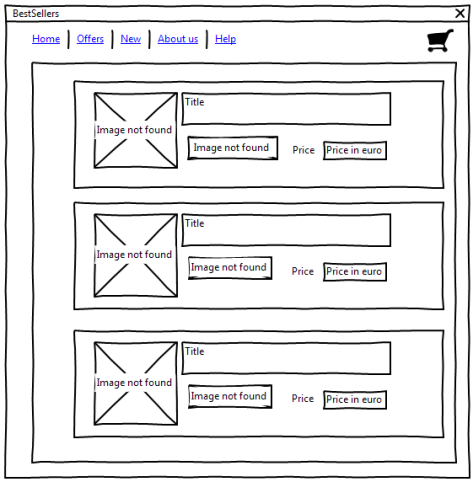

Figure A.26: Wireframe for the 'best' screen.

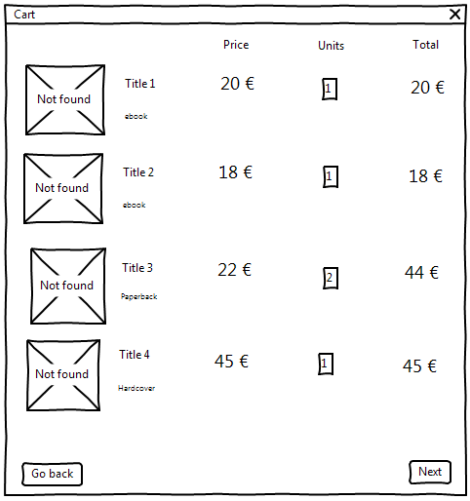

Figure A.27: Wireframe for the 'cart' screen.

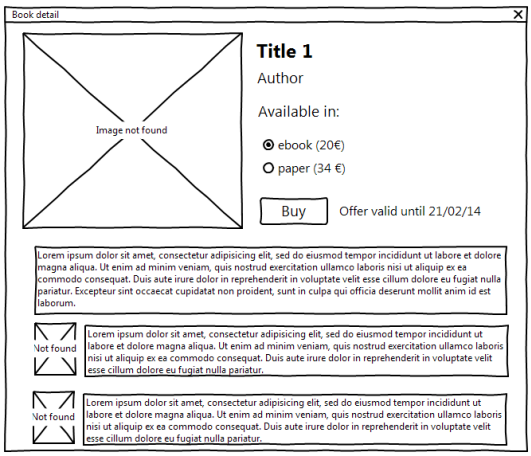

Figure A.28: Wireframe for the 'detail' screen.

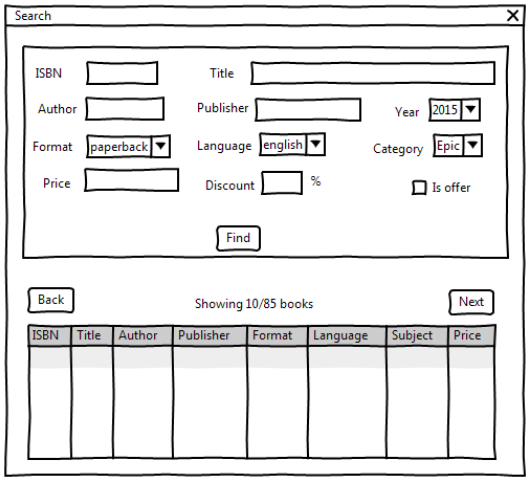

Figure A.29: Wireframe for the 'search' screen.

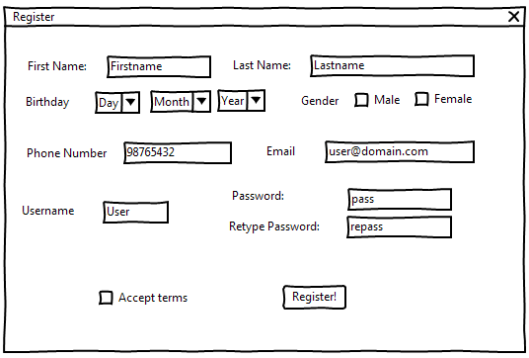

Figure A.30: Wireframe for the 'user' screen.