Information and Software Technology 70 (2016) 155–175

Contents lists available at ScienceDirect

Information and Software Technology

journal homepage: www.elsevier.com/locate/infsof

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

ARTICLE INFO

Article history:
Received 16 April 2014
Revised 18 October 2015
Accepted 19 October 2015
Available online 30 October 2015

Keywords:
Graphical User Interfaces
Layout inference
Wireframes
Model-driven engineering
Reverse engineering
Fluid layout

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.

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 [1]. In these cases, the position of widgets was expressed in terms of absolute 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 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

* Corresponding author. Tel.: +34 868884610.
E-mail addresses: osanchez@um.es (Ó. Sánchez Ramón), jesus.sanchez.cuadrado@uam.es (J. Sánchez Cuadrado), jmolina@um.es, jesus.gmolina@gmail.com (J. García Molina), jean.vanderdonckt@uclouvain.be (J. Vanderdonckt).

http://dx.doi.org/10.1016/j.infsof.2015.10.005
0950-5849/

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 [2] 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 is 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.

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