Automated Software Engineering manuscript No.
(will be inserted by the editor)

Model-Driven Reverse Engineering of Legacy Graphical User Interfaces

Óscar Sánchez Ramón ·
Jesús Sánchez Cuadrado ·
Jesús García Molina

Received: date / Accepted: date

Abstract Businesses are increasingly beginning to modernize those of their legacy systems that were originally developed with Rapid Application Development (RAD) or Fourth Generation Language (4GL) environments, in order to benefit from new platforms and technologies. In these systems, the Graphical User Interface (GUI) layout is implicitly provided by the position of the GUI elements (i.e. coordinates). However, taking advantage of current features of GUI technologies often requires an explicit, high-level layout model. We propose a Model-Driven Engineering process with which to perform the automatic reverse engineering of RAD-built GUIs, which is focused on discovering the implicit layout, and produces a GUI model in which the layout is explicit. As an example of the approach, we apply an automatic reengineering process to this model in order to generate a Java Swing user interface.

Keywords Graphical User Interfaces, layout, modernisation, Model Driven Engineering, Reverse Engineering, Reengineering.

Óscar Sánchez Ramón
Facultad de Informática, Campus de Espinardo, 30100, Murcia, Spain
Tel.: +34-868-884311
Fax: +34-868-884151
E-mail: osanchez@um.es

Jesús Sánchez Cuadrado
Escuela Politécnica Superior, Universidad Autónoma de Madrid, Francisco Tomás y Valiente, 11, 28049, Madrid, Spain
Tel.: +34-91-4972222
E-mail: jesus.sanchez.cuadrado@uam.es

Jesús García Molina
Facultad de Informática, Campus de Espinardo, 30100, Murcia, Spain
Tel.: +34-868-884311
Fax: +34-868-884151
E-mail: jmolina@um.es

1 Introduction

Most information systems dating from the 90's were built using Rapid Application Development (RAD) environments. The RAD methodology appeared in the early 90's as a response to the non-agile development processes that existed (Martin, 1991), and a number of integrated environments supporting third and fourth generation languages also appeared. Oracle Forms, Visual Basic or Delphi are well-known examples of RAD environments. They reduce development time by facilitating GUI design and coupling data access to graphical components. However, the evolution of these applications was hindered in the long term since bussiness logic tended to be mixed with GUI logic. This has motivated a large number of businesses to migrate their RAD legacy systems to new platforms (typically Web platforms), which better meet their needs of extensibility, maintainability or distribution, among others. Another reason for this migration is that some vendors are increasingly ceasing support in favour of other platforms.

Reengineering is a form of modernisation that consists of the systematic transformation of an existing system into a new form in order to allow quality improvements to be made to several aspects of a legacy system (Tilley and Smith, 1995). This transformation involves a three-stage process to create the new system: reverse engineering, restructuring, and forward engineering (Chikofsky and Cross, 1990). A migration is a special case of reengineering in which a legacy software system is moved from one technology to another. In this paper we shall focus on the migration of GUIs built with a RAD environment.

Migrating a legacy application to a new technology necessitates tackling three main aspects: data access, bussiness logic and graphical user interface (GUI). Various works have dealt with the migration of RAD-based legacy systems (Harrison and Lim, 1998; Andrade et al, 2006), but user interface migration has been typically regarded as a straightforward topic, in which the only concern is to establish mappings between widgets of the source and target technologies. However, dealing with current technologies and devices requires a thorough analysis of the user interface so that it can be suitably reengineered. Notably, reverse engineering the layout of the user interface (i.e. obtaining an explicit model from the spatial relationships among widgets) is crucial if subsequent reengineering activities are to be accomplished.

Model Driven Engineering (MDE), meanwhile, has arisen as a new software development paradigm in which models, which are abstract representations of aspects of software systems, drive the whole development process. In this setting, model transformations are used to convert models between different levels of abstraction, which enables automation. MDE techniques are not only applicable to the creation of new applications, but can also be used to evolve existing systems by automating evolution activities, such as reverse engineering.

Layout is the sizing, spacing, and placement of content within a window, which is a key aspect in GUI design. In this paper, we explore the concerns involved in discovering layout relationships among user interface elements. We focus on GUIs built with RAD environments, in which the layout is implicitly represented by means of the explicit position of widgets. A set of algorithms to reverse engineer the layout is shown, so that an explicit layout model is obtained. These algorithms are the core of our MDE framework for reengineering GUIs built with RAD environments, which is also presented in this work. The reverse engineering stage of the proposed framework uses models to represent the information gathered and a model tranformation chain to automate the process of generating a layout model. The approach is reusable for different legacy source platforms and target platforms, i.e., it is independent of the platform. We have evaluated these algorithms by applying them to two real legacy applications, built by two different companies, and consisting of 57 and 107 windows of different types. The results of the evaluation indicate that, on average, 95% and 87% (respectively) of the widgets are properly laid out.

The main contributions of our work are therefore: i) the model-based architecture proposed to reverse engineering the layout, ii) the metamodels (i.e. data structures) to represent the information gathered, and iii) the algorithms devised to infer the layout. A limited number of works have dealt with layout recovery (Lutteroth, 2008; Stroulia et al, 2003). Our work is novel in explicitly representing the layout extracted and using a model-driven approach to solve the layout detection problem in a platform-independent way. A preliminary version of this work was presented as a short paper at the ASE'10 conference (Sánchez Ramón et al, 2010). This paper extends it significantly by detailing the solution architecture, presenting the metamodels and algorithms involved, and performing an evaluation of the approach.

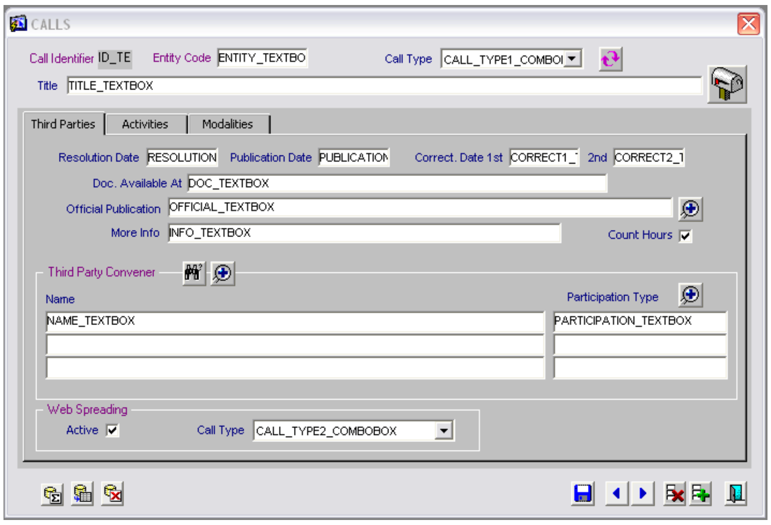
The paper is organised as follows. The following section motivates the need to migrate RAD-based legacy systems, introduces the technical context of the paper, and briefly describes the architecture of our solution. The main metamodels of the presented approach will be described in Section 3. Section 4 presents the algorithms involved in our reverse engineering approach. The following section shows implementation details of the architecture (Section 5), and Section 6 presents the evaluation of our approach. Finally, Section 7 presents the related work, and Section 8 finishes the paper.

2 Overview of the approach

In this section we will first introduce the context and requirements of the migration of the GUI of RAD applications. We will then present an overview of the approach. Finally we will explain the benefits of the approach followed.

2.1 Migration requirements of the GUI of RAD applications

As it was stated in the introduction, the migration of a legacy application must deal with the data access, the business logic and the GUI. In this paper


CALLS

Call Identifier ID_TE Entity Code ENTITY_TEXTBO Call Type CALL_TYPE1_COMBOI

Title TITLE_TEXTBOX

Third Parties | Activities | Modalities

Resolution Date RESOLUTION Publication Date PUBLICATION Correct. Date 1st CORRECT1_ 2nd CORRECT2_1

Doc. Available At DOC_TEXTBOX

Official Publication OFFICIAL_TEXTBOX

More Info INFO_TEXTBOX Count Hours ✓

Third Party Convener

Name	Participation Type
NAME_TEXTBOX	PARTICIPATION_TEXTBOX

Web Spreading

Active ✓ Call Type CALL_TYPE2_COMBOBOX

Fig. 1 Example window.

we focus on the automation of reverse engineering tasks related to GUIs built with a RAD environment. One of the key aspects involved in the migration of the GUI is the discovery of the layout, that is, obtaining a representation that reflects how the graphical panels and widgets are arranged on the screen and displayed to the user, i.e. the design of the interface. Discovering the layout of a GUI is important if subsequent reengineering activities are to be accomplished, and our aim is to tackle the reengineering of applications created with RAD environments (from here on referred as *RAD applications*). Figure 1 shows an example window created with Oracle Forms, which is a well-known RAD environment.

Modern GUI technologies frequently provide developers with predefined layouts with which to create interfaces, such as *GridLayout* or *BorderLayout* in the case of Java Swing. However, the GUI of the RAD applications does not have an explicit notion of layout, and widgets are dragged from a palette and placed into a particular position (which is sometimes almost arbitrary) on a canvas. RAD applications therefore only provide a *coordinate-based layout.* The migration of RAD applications to 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.

Various works have dealt with the reengineering of legacy applications, some of which migrate the user interface as part of the process. A typical example is the migration of legacy desktop applications to Web platforms (Chen et al, 2003; Bandelloni et al, 2005). However, the quality of the target layout of the user interface tends to be poor, since these proposals only map widgets between source and target technologies and replicate the original layout in a simple and straightforward manner. Hence they do not actually recover the layout of the application, thus hindering subsequent reenginering steps. Next, we briefly comment on three cases in which an explicit recovery and representation of the layout would enable reengineering activities to be performed:

- **Layout-preserving migration**. Application users are sometimes averse to change, and some migration projects have therefore 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 to generate a mock application which tracks user input in order to generate test cases (Memon et al, 2003).
- **Reengineering for GUI adaptation**. Migrating to a new GUI technology requires taking 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 (Lutteroth, 2008). An explicit representation of the GUI would enable this analysis, and would also permit refactoring.

In this way, we propose a solution that can be used to extract a high level representation of the layout of the original application. In contrast to previous works, our solution is devised to be used not only for a specific technology but reused with little effort for all the source technologies that share the characteristics of RAD environments. We offer a concrete solution that takes advantage of the benefits of MDE, that is, as part of the solution we include the algorithms (in the form of model transformations) and data structures (metamodels) needed to tackle the problem. Our solution is driven by the following requirements:

- **Explicit GUI information**. A high-level representation of the GUI must be discovered, i.e. metadata concerning the GUI. It must be possible to analyze and automatically transform this metadata.
- **Modularity and automation**. Owing to the wide semantic gap between RAD environments and current technologies, it would be desirable to split the reengineering process into simpler stages to make it maintainable. We are also interested in automating it as much as possible.
- **Source and target independence**. The reengineering process should be easy to reuse with different RAD technologies (source independence). Furthermore, it must be extensible, so that new target platforms can be added without changing the reverse engineering and restructuring stages (target independence).

2.2 Overview of the MDE solution

In order to fulfill the elicited requirements we have addressed the reverse engineering process by applying MDE techniques, that is, metamodelling and model transformations (Clark et al, 2004). A **metamodel** describes the structure of **models**, which are used to represent an aspect of a system at a particular abstraction level. In our case, GUI metadata will be explicitly expressed by means of models, and metamodels will define the structure of these metadata. Several metamodels have been defined to represent a source GUI and the metadata gathered during the reverse engineering process.

Model transformations allow the conversion of models at different levels of abstraction to be automated, by establishing mappings between metamodels. In our case, we have automated the reengineering process by means of a transformation chain, that is, a sequence of model transformations performed in order to split the transformation process into several stages since the process entails bridging a wide semantic gap.

Figure 2 shows the transformation chain devised. The process is split into the three typical stages of a reengineering process (reverse engineering, restructuring and forward engineering).

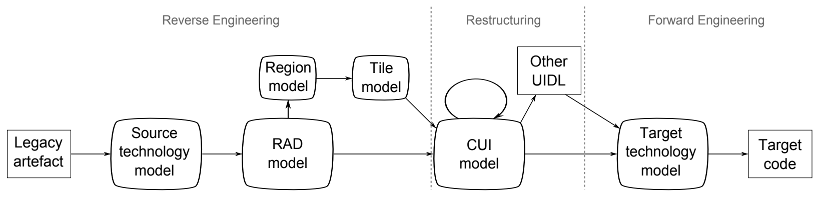


Fig. 2 Model-based architecture used to migrate GUIs built with a RAD.

The reverse engineering stage of a model-driven reengineering process starts by injecting models from the source artefacts. This requires the implementation of a bridge between technical spaces (Bézivin and Kurtev, 2005), particularly a bridge between the technical space of the source artefacts (e.g. *grammarware* or XML) and the MDE technical space (a.k.a. *modelware*).

In our architecture, a model-based representation of the legacy GUI is obtained in the form of a **Source Technology model**. This model conforms to a metamodel that depends on the specific RAD technology.

Afterwards the Source Technology Model is transformed into a **RAD model**. This model conforms to the so called *RAD* metamodel which generalises concepts common to GUIs built with a RAD environment. It is a kind of normalisation model that is intended to make the rest of the reverse engineering process independent of the source technology.

The principal and most complex part of the reverse engineering process (which is the main focus of this work) is the layout abstraction step which takes a RAD model as input, discovers the implicit layout of the GUI and returns a **Concrete User Interface (CUI) model** as output. According to the terminology of the Reference Framework given in (Limbourg and Vanderdonckt, 2004), a CUI model is a technology-independent representation of a GUI. This model provides a representation of the GUI at a higher-level of abstraction than the RAD model.

The CUI model is the result of the reverse engineering process, and makes restructuring and forward engineering possible. Restructuring allows the GUI to be adapted and redesigned. This can be done manually, semi-automatically or automatically. On the other hand, forward engineering allows new software artefacts to be generated. For instance, our CUI model could be mapped (this can be considered as restructuring) into another User Interface Description Language (UIDL) such as UsiXML (Limbourg and Vanderdonckt, 2004) in order to take advantage of the existing tools.

Finally, the CUI or UIDL representing the original or modified user interface can be moved to a different technology, thus obtaining a **Target Technology model** (e.g. a Java Server Faces model) from which it can automatically generate the final user interface code.

2.3 Benefits of the approach

The proposed architecture satisfies the previously stated requirements. Firstly, explicit information about the layout of the graphical elements in the user interface is condensed in CUI models, thus allowing automatic restructuring and forward engineering processes and tools to be applied to these models.

Secondly, resolving the layout abstraction by means of a transformation chain allows the problem solution (algorithms) to be split into smaller modules (transformations) which can be developed and evolved independently, since the metamodels act like the contracts of the modules. The transformations can be chained and executed sequentially to achieve automation, signifying that the most complex part of the process (the generation of the CUI models from the RAD models) is performed automatically. It can be observed that the application of the MDE principles results in a more maintainable solution.

Lastly, as we have a RAD model as an input of the reverse engineering process and a CUI model as an output, we achieve source and target platform independence. Thus, only the corresponding injector plus the transformation to derive RAD models are required to support a new RAD technology. Note that the reverse engineering algorithms can be applied independently of the source and target technologies. Reusability and extensibility are thus also promoted in our approach.

The contributions of this work are therefore: i) a set of algorithms with which to reverse engineer the layout of RAD GUIs, which are applicable to coordinate-based GUIs in general, ii) the metamodels (i.e. data structures) to represent the information gathered in the reverse engineering, iii) a concrete solution implemented as an MDE framework in order to reengineer the GUI of RAD applications.

3 Reverse engineering metamodels

In this section we describe the RAD and CUI metamodels that were presented in the previous section. Contrary to Source and Target Technology models, RAD and CUI models are independent of a concrete technology.

RAD metamodel

The commonalities of the GUIs built with RAD environments are described by means of a metamodel denominated as RAD. This is a generic metamodel which allows the GUI of any source RAD application to be expressed in terms of features which are typically provided by RAD environments, such as widgets positioned with coordinates (which form an implicit layout) and a hierarchy of common widgets.

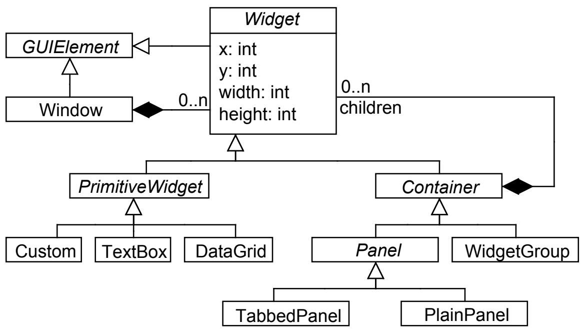


Fig. 3 Excerpt of the RAD metamodel.

The design of the RAD metamodel has been driven by the identification of common features of RAD-based GUIs. An excerpt of the metamodel is shown in Figure 3. Some of these features and their representation in the metamodel are outlined as follows:

– **Implicit layout**. The position of GUI elements (e.g. widgets) is stated by means of coordinates that are relative to the main window or another container. The size of a widget is also given explicitly by the designer. This means that, for example, when a window is resized the widgets are not resized or rearranged accordingly. In the metamodel this is conveyed by the *Widget* metaclass which has *x* and *y* attributes (a coordinate), and an explicit *width* and *height*.
– **Clustering elements**. There are special widgets which are intended to group and/or highlight semantically-related widgets. These widgets are represented as subtypes of *Container* in the metamodel. In particular, we distinguish between *Panel*s that are elements that arrange a window in

parts (in some RAD environments they can also be reused between windows), and *WidgetGroup*s that are used to highlight a set of widgets in close proximity, frequently by means of a border.
- **Overlapping**. Widgets are often loosely contained in their container, that is, they are overlapped with the container instead of having explicit containment relationships. A container could also be overlapped with another container. This means that a *Container* may not have any widget in the *children* reference, although there may be some widgets that would (visually) be expected to be contained. For example, the *CardLabel* element in Figure 6 is not actually contained in the *PaymentFrame* element, as can be seen in the GUI tree in Figure 7, but is simply overlapped with the region occupied by *PaymentFrame*.
- **Standard widgets**. RAD environments share a common set of standard widgets, such as text boxes, buttons, combo boxes, tables, and so forth. They are represented in the metamodel with metaclasses inheriting from *PrimitiveWidget*.
- **Technology-dependent widgets**. Source technology-dependent widgets (e.g. an ActiveX control) cannot be represented in the RAD metamodel (which is technology-independent), and cannot therefore be part of the subsequent reverse engineering. We propose two alternatives to deal with this issue: i) the metamodel provides a special widget (*Custom*), that allows the reverse engineering process to deal with them, and developers are in charge of giving them a proper meaning in a later reengineering stage, ii) some specific widgets can be emulated by one or more standard widgets from the metamodel. For instance, an Oracle Forms multirecord is a group of primitive widgets (e.g. text boxes) arranged in a tabular form, which can be mapped into a *DataGrid*. This mapping is typically carried out in the RAD normalisation stage.

As stated previously, a RAD model is derived from a Source Technology model by means of a model-to-model transformation. Given that the RAD metamodel does not establish tight restrictions regarding the arrangement of widgets, defining this model-to-model transformation in order to translate Source Technology metamodel concepts into RAD metamodel concepts is normally straightforward.

CUI metamodel

In our solution, CUI models conform to the metamodel shown in Figure 4, in which the layout is explicitly modelled with compositions of high-level concepts which are present in most GUI frameworks, such as flows of elements, grids, and so forth.

In this metamodel, a *View* represents a software artefact that displays the part of the GUI that a user sees at a particular moment, such as a desktop application window. From here on, the term *View* will be used to refer to the metaclass and the term *view* will be used as a general concept. *View*s are

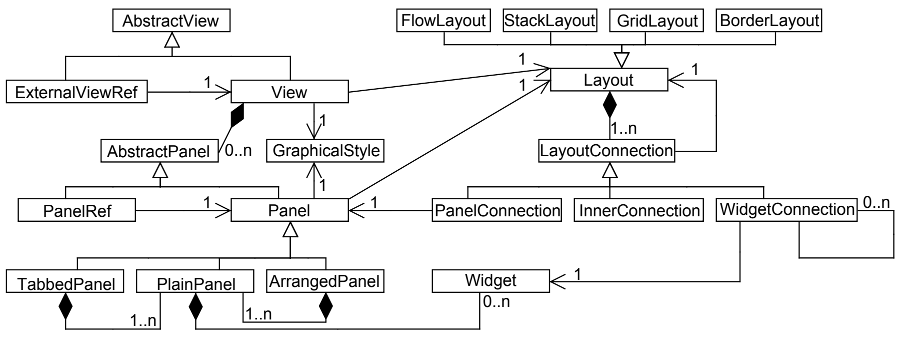


Fig. 4 Simplified CUI metamodel.

composed of *AbstractPanel*s (i.e. *Panel*s and *PanelRef*s), which are reusable parts of the GUI, in such a way that a panel could be used in several views. *Panel*s can contain subpanels or widgets. Views and panels have a graphical style (that defines the font type and background colour, for example) and a layout that describes how the subpanels or widgets are arranged. The layout is expressed in terms of hierarchies of high-level arrangements (e.g. *FlowLayout*, *StackLayout*, etc.), and has connections (*LayoutConnection*) that indicate which subpanels (*PanelConnection*) or widgets (*WidgetConnection*) are arranged according to it. *InnerConnection*s do not refer to any panel or widget and are used to create a layout tree structure. A *WidgetConnection* can be related to other *WidgetConnection*s, which is used to express dependencies between widgets (e.g. associate a text field and a label).

It is worth noting that the metamodel supports the separation between three concepts: the panel as a reusable part of a view, its graphical style and the layout of the subpanels or widgets that it contains. This metamodel also covers some other aspects of a GUI, which are not within the scope of this paper, but which are of interest when reengineering, such as support for internationalisation.

4 Layout reverse engineering

In this section we shall describe the algorithms involved in the layout reverse engineering process, along with the metamodels that support them. We shall focus on the main part of the reverse engineering process, which is the layout abstraction step (step from RAD to CUI in Figure 2)[1]. This section therefore explains the technology-independent part of the reverse engineering process, whereas the obtaining of the RAD models will be explained for a particular RAD technology in Section 5.

The wide semantic gap that exists between the RAD and the CUI metamodels, means that the transformation requires two additional steps which are

[1] A case study GUI taken from a real application that illustrates the reverse engineering process in detail can be found at `www.modelum.es/guizmo`.

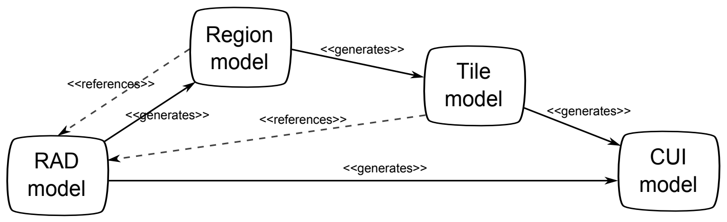


Fig. 5 Model architecture used to obtain the CUI model from the RAD model.

used to gather some information implicitly expressed in the RAD model. Figure 5 shows how this process works, where *generates* indicates the models that are required to generate a target model, and *references* indicates the models that contain references to previous models. In a first stage, a Region model is obtained, and a model-to-model transformation is then applied to this model in order to generate a Tile model. Both models annotate a RAD model with additional layout information that is useful to derive a CUI model.

We shall first outline the challenges faced when reverse engineering the layout in GUIs built with a RAD environment. The two stages that generate the Region and Tile models will then be explained in sections 4.2 and 4.3 respectively, while section 4.4 will show how we rely on the gathered information to derive a CUI model. The example view in Figure 6 and its partial GUI tree shown in Figure 7 are used as a running example in this section.

4.1 Challenges in layout reverse engineering

In RAD environments the layout is implicitly defined by the position of the elements, which are expressed in terms of coordinates. Our aim is to capture the visual arrangement of the elements in such a way that both replicating the layout and redesigning it for a different technology is easy. Transforming an implicit, coordinate-based layout (represented by the metamodel in Figure 3) into an explicit, high-level layout (represented by the metamodel in Figure 4) poses the following challenges.

1. **Region identification**. A view can be seen as a composition of parts or regions (perhaps implicit) which provides the widgets of the view with a structure. Reverse engineering the structure of a view by identifying regions is necessary for layout redesign. In the example we can make out three regions in the window. Region *R2* contains the widgets that are surrounded by the *PaymentFrame* frame, region *R1* is composed of the widgets above the frame, and region *R3* includes the widgets below the frame (note that *R1* and *R3* are implicit).
2. **Explicit containment**. As explained in Section 3, in some cases elements are not actually contained in a container, but are overlapped. In the example, *PaymentFrame* surrounds *CardLabel*, *CardCombo*, *DiscountLabel* and *DiscountCheck* (the checkbox next to *DiscountLabel*), but these widgets are

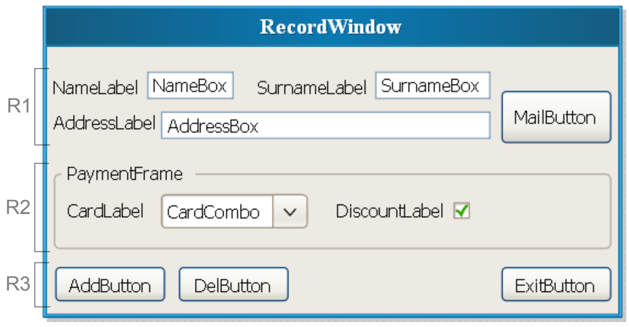


Fig. 6 Example view for entering personal information. Widgets are placed with explicit coordinates.

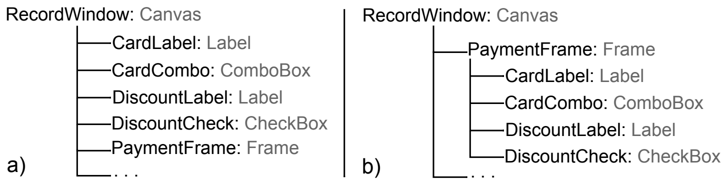


Fig. 7 (a) Fragment of the original GUI tree. (b) The expected GUI tree.

only visually contained in the frame, that is, their parent element in the model is not *PaymentFrame*, but rather *RecordWindow* (see Figure 7). The region identification commented on above must also be taken into account. Matching the containment hierarchy and the visual structure of the layout greatly simplifies the reverse engineering and restructuring algorithms, and it is thus necessary to establish explicit containment relationships.

3. **Widget structure recognition**. While region identification aims to recognize those parts of which the view is structured, widget structure recognition is focused on how widgets that are spatially-close to each other are arranged. For example, the widgets inside the *PaymentFrame* form a line. Widgets are often not perfectly aligned, so heuristics are needed. To continue with the example, *NameLabel*, *NameBox*, *SurnameLabel* and *SurnameBox* could form a line, but it is not clear whether *MailButton* would be considered as a component of this line.
4. **Coordinate abstraction**. As already mentioned, a coordinate-based positioning system is not desirable, and thus an alternative means to represent relationships between elements is needed. For example, it would be desirable to know that *NameLabel* is above *AddressLabel* and on the left of *NameBox*.
5. **Hole detection**. The term *hole* refers to an area of a remarkable size that does not contain widgets but is surrounded by them. In the example view, there is a hole between *DelButton* and *ExitButton*. It is necessary to capture layout holes if a similar layout is to be reproduced in a different technology.

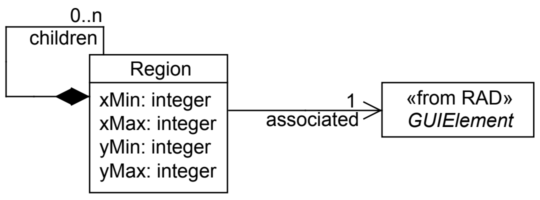


Fig. 8 Region metamodel.

The following subsections show the algorithms that deal with these issues.

4.2 Detecting regions and containers

This stage is intended to tackle issues 1 and 2 commented on above (namely, region identification and explicit containment). Here, a Region model is automatically derived from a RAD model.

A **Region model** is a model that annotates a RAD model in order to make visual containment relationships between widgets explicit. A Region model represents a tree of regions that conforms to the metamodel shown in Figure 8. It has a unique metaclass called *Region*, which has the two pairs of coordinates that define a rectangular area, and the *children* reference to the sub-regions contained in it. Note that *Region* elements are annotations for the *GUIElement*s of a RAD model. Region models have three main features: i) each *GUIElement* is associated with a *Region* defined by two pairs of coordinates, ii) *Container*s and *PrimitiveWidget*s must not exist at the same level (i.e. a region that annotates a *Container* cannot be a sibling of a region that annotates a *PrimitiveWidget*), and iii) overlapped regions are not permitted.

PrimitiveWidgets are prevented from being at the same level as *Containers* as a means to structure the GUI in a uniform manner, given that the windows are divided into parts which are disjointed and complementary. Each window therefore contains several separate regions (which can in turn contain more regions or widgets), and each widget belongs to a unique region. The goal of this design decision is twofold. On the one hand, we believe that conceptually a UI is composed of related parts like a puzzle in such a way that there are no widgets outside of a part. On the other hand, it makes the structure of the UI uniform and simplifies the later algorithms.

A precondition of the algorithm used to create the regions is that the border of a *Container* must never cross the border of another *Container*. Our framework has a previous phase that checks whether frame border overlapping occurs. If this occurs, then the reverse engineering process is stopped and a message is shown to the developper so he can fix the GUI manually (although, in our experience this situation rarely arises).

In the algorithm we distinguish between three types of regions: *widget regions*, *base regions* and *extra regions*. A *widget region* is a region associated with a widget. The term *base region* is used to refer to a region that is as-

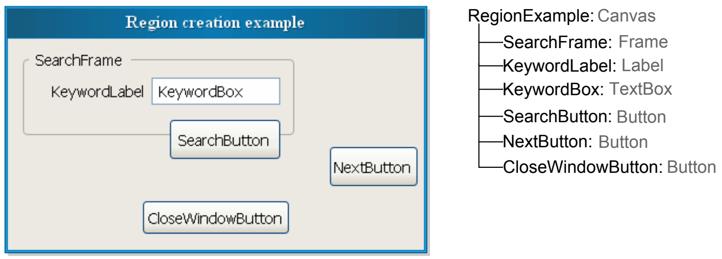


Fig. 9 Left: example window for the region detection. Right: the logical structure of the widgets.

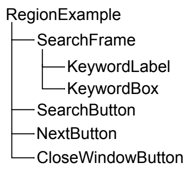


Fig. 10 Structure of the regions after step 2 for the example in Figure 9.

sociated with a RAD container. *Extra regions* are artificial regions which are created to contain widgets that are not included in a *base region*. Note that *base regions* and *extra regions* will contain subregions, unlike *widget regions*. We will explain the region detection algorithm with the ad-hoc example window in Figure 9. The algorithm used to create the Region model (Algorithm 1) is summarised in the following steps:

Algorithm 1 Region creation algorithm.

```
 1: for all $window$ do
 2:     $r_0 \leftarrow createRegion(window)$
 3:     for all $w \in getWidgets(window)$ do        ▷ Gets PrimitiveWidgets and Containers
 4:         $r_1 \leftarrow createRegion(w)$
 5:         $addChild(r_0, r_1)$
 6:     end for
 7:
 8:     for all $r_1, r_2 \in children(r_0)$ do
 9:         if $r_1 \neq r_2 \wedge contains(r_1, r_2)$ then
10:             if $\nexists r_3 \neq r_2 \neq r_0.(contains(r_3, r_2)) \vee$
                        $\forall r_3 \neq r_2 \neq r_0.(contains(r_3, r_2) \rightarrow contains(r_3, r_1))$ then
11:                 $addChild(r_1, r_2)$
12:             end if
13:         end if
14:     end for
15:
16:     $createExtraRegions(r_0)$
17: end for
```

1. Create a region for every *GUIElement* (lines 2 to 6). r_0 is a base region associated with the window, which is the root of the region tree. r_1 is a

(widget or base) region associated with w, which can be a primitive widget or a container. $add(r_0, r_1)$ means that r_1 is set as a child of r_0. The area of a new region is derived from the (x, y) coordinates, the width and the height of the *GUIElement*. For example, in Figure 9, a base region is created for each one of the containers (the *RegionExample* window and *SearchFrame*), and a widget region is created for each primitive widget (*KeywordLabel*, *KeywordBox*, *SearchButton*, *NextButton*, *CloseWindowButton*).

2. Create a tree structure by nesting the regions according to the visual containment relationships (lines 8 to 14). The expression $contains(r_1, r_2)$ is true if the coordinates of r_2 are inside the rectangle defined by the coordinates of r_1. For each pair of regions, r_1 and r_2, we make r_2 a child of r_1 if r_1 contains r_2 and one of the following conditions is true: i) there is not a different region r_3 containing r_2 (r_2 is a direct child of r_1), ii) there is another region r_3 containing r_2 but it also contains r_1 (r_2 is a direct child of r_1 which in turn is a direct child of r_3). The evolution of the example window after this step can be seen in Figure 10: *SearchFrame* now contains *KeywordLabel* and *KeywordBox*. At the end of this step, there can be widget regions which are siblings of base regions in the Region model. Following with the example we can see that *SearchButton*, *NextButton* *CloseWindowButton* are siblings of *SearchFrame*.
3. Create extra regions to prevent *PrimitiveWidget*s from being at the same level (siblings) as the *Container*s. The algorithm iterates once over every widget region that is a sibling of either a base region or an extra region (at the beginning there are only base regions). For each widget region we have three possible cases:
   - *Case A*: the widget is not partly contained in any existing base or extra region (i.e. the widget does not cross the bounds of any base or extra region), so a new extra region is therefore created for the widget region. The new region takes the maximum area available without interfering with the other regions. Following with the example, we assume that we have already dealt with *KeywordLabel* and *KeywordBox*, and now is the turn of *CloseWindowButton*. As this widget is not contained in the unique base region *R1* (see left part of Figure 11), a new extra region *R2* is created for it (right part of Figure 11).
   - *Case B*: the widget region is partly contained in a base region. In this case the size of that base region is increased to enable it to cover the area occupied by the widget region, and the widget is added to it. Augmenting the size of the base region may cause that the base region overlaps some extra regions, and the overlapped extra regions are therefore shrunk to avoid the overlapping. Continuing with the example, let us make the algorithm iterate over *SearchButton* which is partly contained in the region *R1* associated with *SearchFrame* (left part of Figure 12), so we augment the base region to fully contain *SearchButton* (right part of Figure 12). This implies that the region *R2* is shrunk. If a widget is partly contained in more than one sibling base region, then the widget is included in only one base region, and in this case the

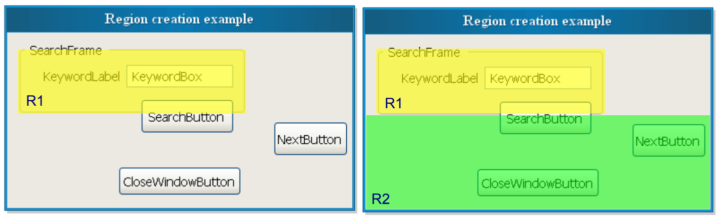


Fig. 11 Case A. Left: example window with a base region *R1*. Right: a new extra region *R2* created to contain *CloseWindowButton*.

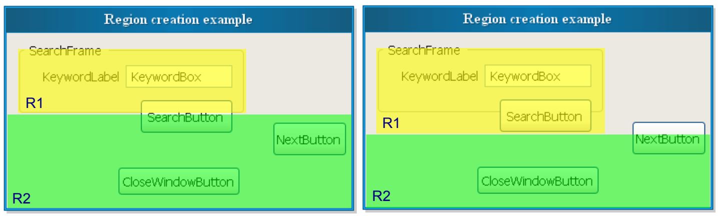


Fig. 12 Case B. Left: example window with a base region *R1* and an extra region *R2*. Right: the base region *R1* is augmented to include *SearchButton* completely and the extra region *R2* is diminished.

widget is shrunk to fit into that base region. We have not found this case yet in practice.

– *Case C*: the widget is partly contained in an extra region. It is necessary to reduce the extra region that partly contains the widget so that the widget no longer enters its area anymore. In addition, a new extra region to contain the widget is created. Going back to the example (see Figure 13), the algorithm iterates over *NextButton*. As the widget crosses the bounds of the extra region *R2*, this region is resized to exclude the widget. Hence, a new extra region *R3* is created to contain the new widget without interfering with any of the already created regions.

The cases are evaluated in the following order: case B, case C, case A. Note that in the example we have iterated over the widgets in a way that it facilitates the explanation of the cases, though other orders are also possible. The different orders will end up in regions that may differ in their coordinates but that group the widgets in the same way.

4.3 Uncovering relative positions

The objective of this second stage is to make the layout independent of the coordinate-based system. This deals with issues 3, 4 and 5 mentioned previously (namely, widget structure recognition, coordinate abstraction and hole detection). The input of this stage is a Region model, and a Tile model is automatically generated.

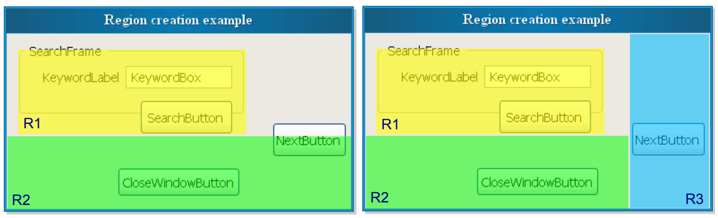


Fig. 13 Case C. Left: example window with a base region *R1* and an extra region *R2*. Right: a new extra region *R3* is created to contain *NextButton*, and the region *R2* is diminished.

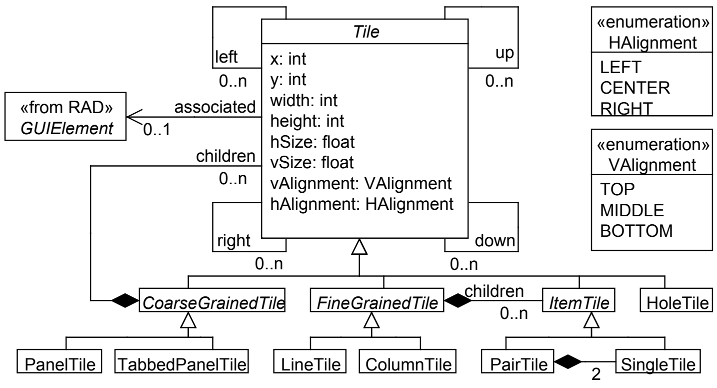
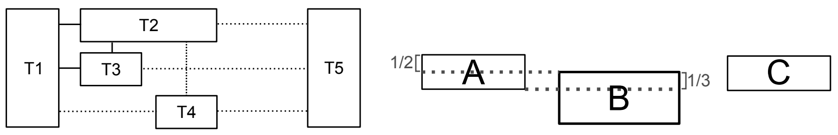


Fig. 14 Tile metamodel.

Tile models are mainly focused on representing how widgets and containers are arranged, in terms of relative positions among them. We define a **tile** as a part of a view with spatial relationships with other neighbouring parts. For example, a certain tile could have another tile above it and a different tile below. This positioning system is useful for the later identification of high-level layout patterns, as will be shown in Section 4.4. Tile models also refine Region models by identifying sub-structures inside regions, such as groups of widgets that form a line.

The Tile metamodel is shown in Figure 14. The main concept is that of *Tile*. Every *Tile* is associated with the *GUIElement* from which it originated, if one exists (i.e. some tiles originated from extra regions). Such references to the RAD model are propagated from the Region model. There are four zero-to-many relationships between tiles, which are used to relate the tiles spatially, namely *right, left, up, down.* We use *hSize* and *vSize* to measure the percentage of the width and height that is taken up by that tile in the view with regard to the width and height of the container tile. Tiles also include information about the area they take up by means of *x*, *y*, *width*, *height.* A tile can also be aligned with regard to its container tile, and *hAlignment* and *vAlignment* are used for this purpose. We distinguish four kinds of tiles:

- **Coarse-grained tiles**: these tiles arrange a view in parts which can be visually distinguished. Each tile represents a block of related widgets which



(a) Adjacency example (b) Horizontal intersection value example

Fig. 15 Relative positions between tiles.

are in the same area and are likely to contain widgets to perform system actions (e.g. the bottom buttons in Figure 6), or data concerning a topic such as "payment details" in the middle part of Figure 6. All base and extra regions are mapped to this kind of tile. For instance, in Figure 6 the regions *R1*, *R2*, and *R3* are mapped to *PanelTile*s.

– **Fine-grained tiles**: these tiles arrange a set of widgets that are spatially close and have a certain spatial structure, such as a horizontal line (*LineTile*) or a vertical column (*ColumnTile*). Fine-grained tiles are aggregated inside coarse-grained tiles. To continue with the example, *NameLabel*, *NameBox*, *SurnameLabel* and *SurnameBox* are all mapped together to a *LineTile*.
– **Item tiles**: they are associated with single widgets (*SingleTile*) and pairs (*PairTile*) of related widgets such as a text box (e.g. *NameBox*) and its associated label (e.g. *NameLabel*). *Item tiles* are contained in *Fine-grained tiles*.
– **Hole tiles**: these tiles represent a portion of the view of notable size which has no widgets, such as the space between *DelButton* and *ExitButton* in Figure 6.

Next, we establish some of the concepts which allow spatial relationships between tiles to be detected. Figure 15 is used to illustrate these concepts. All the following concepts are defined over tiles, but since we have the (X,Y) coordinates, the width and height of both regions and tiles, the concepts are applicable to regions too.

We will define *adjacency* as a criterion with which to decide whether two tiles of the same kind are spatially related (for example, that a coarse-grained tile T1 is on the left of another coarse-grained tile T2). Our definition of adjacency is based on the concept of *sharing*. A pair of tiles is **vertically sharing** if the intersection of the projections of both tiles on the X axis is not empty, i.e. the x-range of both tiles is overlapped. Likewise, a pair of tiles is **horizontally sharing** if the intersection of the projections of both tiles on the Y axis is not empty, i.e. the y-range of both tiles is overlapped. As is observed in Figure 15(a), *T2* and *T3* are vertically sharing, and T2 and T4 are also vertically sharing, but T3 and T4 are not.

The introduced definitions of sharing are too strict because they consider that overlapped projections always reflect horizontal lines or vertical columns. For instance, in Figure 15(b) *A*, *B* and *C* may (or may not) form a line, because they are not perfectly aligned. This can be addressed by modifying the sharing definition to be more tolerant, and we introduce the *intersection value* with this aim. We define the **vertical intersection value** as the percentage of width that a pair of tiles have in common. It is calculated as the intersection of the x-ranges of the pair of tiles divided by the minimum width of both tiles. Similarly we define the **horizontal intersection value** between a pair of tiles as the percentage of height that a pair of tiles have in common, which is calculated as the intersection of the y-ranges of the pair of tiles divided by the minimum height of both tiles. Figure 15(b) shows how this function is applied. The percentage of the height that tile A has in common with tile B regarding tile A is 0.5, while the value is 0.33 as regards tile B. The result is therefore the maximum value, that is 0.5. Note that a pair of tiles that are horizontally sharing always have a positive horizontal intersection value (similarly with the vertically sharing). The sharing can be redefined (for horizontal sharing as well as vertical sharing) as follows: a pair of tiles are sharing if the intersection value is greater than a threshold which represents the tolerance level, currently set to 0.5.

Based on the concept of sharing, we can now define *adjacency.* A tile t_1 is **vertically adjacent** to another tile t_2 if and only if both tiles are vertically sharing and there is no tile t_3 between t_1 and t_2. Likewise, a tile t_1 is **horizontally adjacent** to another tile t_2 if and only if both tiles are horizontally sharing and there is no tile t_3 between t_1 and t_2. There is a precondition that the tiles t_1 and t_2 must not be overlapped (in our case this is enforced by the RAD). To continue with the example in Figure 15(a), we can see that *T2* and *T3* are vertically adjacent, and *T1* and *T4* are horizontally adjacent, among others.

The *up, down, left, right* relationships of the tiles are defined based on the adjacency as follows. For a tile t_1 it is true that $t_1.right = \{t_2\}$ and $t_2.left = \{t_1\}$ if t_1 and t_2 are horizontally adjacent and t_2 is to the right of t_1. The *down, left, right* relationships are defined in a similar way. Note that when one type of relationship is established for a tile, the opposite type is also set. In the example shown in Figure 15(a), we have the following relationships for T1, T2 and T4:

$$\begin{aligned}
&T1.right = \{T2, T3, T4\}\\
&T2.left = \{T1\}; T2.right = \{T5\}; T2.down = \{T3, T4\}\\
&T4.up = \{T2\}; T4.right = \{T5\}; T4.left = \{T1\}
\end{aligned}$$

As can be seen, $T2.down = \{T3, T4\}$. However, there is a blank space between T2 and T4 that is not captured with the concept of adjacency. Thus, there is some layout information that is lost due to blank spaces being ignored.

In order to tackle this issue, the first step is to set a criterion with which to decide whether two vertically/horizontally adjacent tiles are not sufficiently close, but there is a significant blank space between them. We define that a pair of widgets is **horizontally close** if the percentage of the horizontal distance between the pair, with regard to the container width is smaller than a particular value. A pair of widgets is **vertically close** if the percentage of the vertical distance between the pair, with regard to the container height is smaller than a particular value. It is currently set at 20%. In the example shown in Figure 15(a), when using this criterion we have that *T2* and *T3* are adjacent and close, whereas *T2* and *T4* are adjacent but not close.

When a blank space is detected, two complementary approaches are used to represent it. The first one is to specify that some tiles are aligned with regard to the container tile. To continue with the example, *T1*, *T2* and *T3* are aligned on the left, *T5* is aligned on the right, and *T4* is aligned in the bottom-center. There can be several adjacent tiles with the same alignment, which does not mean that all these tiles have to be attached to the bounds of the container. For instance, *T1* and *T2* are both aligned to the left but actually *T2* is on the right of *T1*. The alignment solution has the disadvantage that there may be blank spaces that are not represented.

The second approach is to include *HoleTile*s which represent blank spaces in the layout, thus signifying that an arbitrary distance between tiles must be maintained. These kind of tiles have dimensions that are specified as a proportion between the empty space and the width or height of the container. Since they are not exclusive solutions, both have been implemented in order to facilitate the obtaining of an accurate high-level layout in the CUI model. The *hAlignment* and *vAlignment* attributes were introduced for the first alternative and the *HoleTile* metaclass for the second one.

Next, the algorithm that takes a Region model and generates a Tile model is presented. Some auxiliary functions are not explained, but their names denote what they do. The algorithm is organised in four phases: i) creating the tiles, ii) establishing *up, down, left, right* relationships between tiles, iii) setting the spatial alignment of the tiles with regard to the container tiles, and iv) creating hole tiles to represent blank spaces. Each phase will be explained separately.

Phase 1. The first phase (see Algorithm 2) generates tiles based on regions. The algorithm traverses the Region model recursively from the root region. It has two parameters: i) the container region (base or extra region) to be traversed, and ii) the parent tile which will contain the created tiles. For each container region (the parameter) to which the procedure *CreateTiles* is applied, it creates a coarse-grained tile (line 5), and for each widget region that is a child of the parameter region, it creates an item tile (lines 10 to 13).

Fine-grained tiles are generated for the content of container regions which include widget regions (lines 7 to 15). This is done by using a clustering algorithm that is applied to the container region in order to identify structures of widget regions (line 24). The clustering algorithm makes a first attempt to group widgets in horizontal lines or columns (horizontal lines have priority over columns) based on the vertical/horizontal sharing. As it has already been said, a pair of regions are sharing if their intersection value is higher than a threshold (set by default at 0.5), and will therefore be classified in the same group. In cases it happens that some widget regions have such a big height

Algorithm 2 Tile creation algorithm. Phase 1: Mapping and clustering.

```
 1: $root \leftarrow getRootRegion()$
 2: $createTiles(root, \emptyset)$
 3:
 4: procedure CREATETILES($region, parentTile$)
 5:     $coarseTile \leftarrow createCoarseGrainedTile(region)$
 6:     if $containsWidgetRegions(region)$ then            ▷ All children are widget regions
 7:         $groups \leftarrow clusterWidgets(region)$
 8:         for all $group \in groups$ do
 9:             $fineTile \leftarrow createFineGrainedTile(group)$
10:             for all $itemRegion \in group$ do
11:                 $itemTile \leftarrow createItemTile(itemRegion)$
12:                 $add(fineTile, itemTile)$
13:             end for
14:             $add(coarseTile, fineTile)$
15:         end for
16:     else                                              ▷ All children are container regions
17:         for all $childRegion \in children(region)$ do
18:             $createTiles(childRegion, coarseTile)$
19:         end for
20:     end if
21:     $addChild(parentTile, coarseTile)$
22: end procedure
23:
24: function CLUSTERWIDGETS($region$)                       ▷ Clustering algorithm
25:     $G \leftarrow detectGroups(children(region))$        ▷ Uses horizontal/vertical sharing
26:     for all $G_1, G_2 \in G.(G_1 \cap G_2 \neq \emptyset)$ do
27:         $G_{new} \leftarrow G_1 \cap G_2$
28:         $remove(G_1, G_{new})$
29:         $remove(G_2, G_{new})$
30:         $add(G, G_{new})$
31:     end for
32:     for all $G_1 \in G$ do
33:         if $\exists r_1, r_2 \in G_1.(areAdjacent(r_1, r_2) \wedge notClose(r_1, r_2))$ then
                                                         ▷ Uses horizontally/vertically close
34:             $splitGroup(G_1)$
35:         end if
36:     end for
37:     return $G$
38: end function
```

that they are horizontally close to widget regions in more than one line, that is, they can belong to different lines of widgets (e.g. tile *T1* in Figure 15(a)). Similarly, some widget regions may be so wide that they are vertically close to widget regions in more than one column. In order to avoid this, we create new groups for those regions that are classified in more than one group (lines 26–31). Finally, we check that adjacent regions inside the groups are vertically/horizontally close, and if this is not the case, then the group is split (lines 32–36).

Phase 2. The second phase of the tile creation algorithm establishes the *up, down, left, right* relationships between adjacent tiles. For each ordered pair of tiles (t_1, t_2) which are children of the same coarse or fine-grained tile, $t_1.up \leftarrow$

t_2 and $t_2.down \leftarrow t_1$ if: i) they are vertically adjacent, ii) they are vertically close and iii) t_2 is above t_1. The *left, right* sets are obtained in the same manner.

Algorithm 3 Tile creation algorithm. Phase 3: Alignment.

```
 1: for all $t_0 \in \mathcal{T}_{coarse} \cup \mathcal{T}_{fine}$ do        ▷ $t_0$ is a coarse-grained or fine-grained tile
 2:     $HAlignedSeq = \{\}$
 3:     $OrderedTiles \leftarrow topologicalSort(children(t_0))$
                                  ▷ Topological sort from up to down and left to right
 4:     for all $t_1 \in OrderedTiles$ do
 5:         /* For simplicity we are only considering the horizontal alignment */
 6:         $add(HAlignedSeq, t_1)$
 7:         if $t_1.right = \{\}$ then
 8:             $xMinPercent \leftarrow first(HAlignedSeq).x/t_0.width$
 9:             $xMaxPercent \leftarrow$
                    $(last(HAlignedSeq).x + last(HAlignedSeq).width)/t_0.width$
10:             if $xMinPercent \leq Lower\_threshold$ then
11:                 for all $t_2 \in HAlignedSeq$ do $t_2.hAlignment \leftarrow LEFT$
12:             else if $xMaxPercent \geq Upper\_Threshold$ then
13:                 for all $t_2 \in HAlignedSeq$ do $t_2.hAlignment \leftarrow RIGHT$
14:             else
15:                 for all $t_2 \in HAlignedSeq$ do $t_2.hAlignment \leftarrow CENTER$
16:             end if
17:             $HAlignedSeq = \{\}$
18:         end if
19:     end for
20: end for
```

Phase 3. The third phase (see Algorithm 3) is in charge of aligning tiles with regard to their container tile. The idea behind this algorithm is based on the following two principles: i) if a tile is very close to the boundaries of its container tile, then the tile is aligned with regard to them, and ii) if several tiles are next to each other, then all of them have the same alignment. For instance, let us assume that in Figure 15(a) the tiles *T1*, *T2*, *T4* and *T5* are very close to the boundaries of the container tile. Therefore *T1* is aligned to the left because it is close to the left boundary, and *T2* and *T3* are aligned to the left because they are on the right of *T1* which is aligned to the left.

In the algorithm the tiles are iterated in a topological order (lines 4–19), which is computed from the directed graph that results from taking into account only the *right* and *down* relations of the tiles. We add each tile to the current alignment group (line 6) and when there are no more adjacent close tiles on the right (line 7), then we assign an alignment type to each one of the tiles in the group (lines 8 to 18). If the most-left tile of the group (the first tile) is close to the left boundary, the alignment is *LEFT* (line 11). If the most-right tile of the group (the last tile) is close to the right boundary, the alignment is *RIGHT* (line 13). If none of the previous cases is applicable, then the alignment is set to *CENTER*.

Phase 4. The last phase of the algorithm identifies significant blank spaces in the view, and creates hole tiles for them. For each pair of tiles that are children of a coarse or fine-grained tile, if the tiles are adjacent and are not close, then we create a hole tile. This new hole tile is placed between t_1, t_2 and the *up, down, left, right* relationships of both tiles are modified. These properties are also initialised for the hole tile according to its relative positioning regarding the t_1 and t_2 tiles. Finally the new hole tile is added to the parent tile.

4.4 High-level layout

At this stage, information about the relationships among elements of the GUI has been gathered. However, it is interesting to take a further step forward in the way in which the layout is represented in the Tile model to make it more similar to the layout managers provided by modern GUI frameworks. To this end, the CUI metamodel introduced in section 3 defines explicit high-level layouts such as grids (*GridLayout*) or stacks of elements (*StackLayout*). For example, if we had a sequence of tiles sorted vertically (each tile below another one), we would explicitly "mark" those tiles as forming a stack layout. The layout types which we use are included in common GUI frameworks such as Java Swing, as well as in diagram editors and other domains (Jacobs et al, 2003; Li and Kurata, 2005).

CUI models are generated from RAD models by using the information provided by the Tile model, in the form of annotations. The algorithm that creates CUI models from Tile models is split into three phases:

1. *Create the structure tree.* The RAD widgets are mapped to CUI widgets, and the tree structure of the widgets of the CUI model is created according to the containment relationships detected in the Region identification stage. With this aim, the tile model is traversed in a recursive manner, and the following actions are performed according to the tile type: if the tile is a coarse-grained tile, it creates a *Panel*, adds it to its container *View* or *Panel*, and continues with the tile children; if the tile is a fine-grained tile, it simply navigates its children; if it is a single tile, it creates a widget for it and adds it to the container *Panel*.
2. *Create the layout tree.* In order to get the high-level layout tree, the Tile model is traversed recursively. For each coarse-grained tile we apply several fitness functions on its children and the layout type whose fitness function returns the greatest value is selected. The fitness functions return a number between 0 and 1 that represents the estimated percentage of tiles that fit the layout out of the tiles in the group. A new layout of the selected type is created by applying a heuristic associated with the layout type. In the case of fine-grained tiles, *LineTiles* are directly mapped to *FlowLayouts*, and *ColumnTiles* are directly mapped to *StackLayouts*.
3. *Link both trees.* It links the GUI and layout trees, by selecting the layout for each container and the container of each child connection of each layout.

The tree structure of the layout tree in step 2 is achieved by means of the *LayoutConnection*s. Each new layout that is created is nested in the parent *LayoutConnection*. Depending on the type of children tiles, different *LayoutConnections* will be created: *PanelConnection* if the child is a coarse-grained tile (it is associated with a *Panel* in the step 3), *InnerConnection* if the child is a fine-grained tile, and *WidgetConnection* if the child is a item tile (it will be associated with a *Widget* in the step 3). Then, the same process is applied for each children coarse or fine-grained tile with the *LayoutConnection* as a parameter.

As can be noticed from step 2, we have a set of layout types and each of them has an associated heuristic and a fitness function. The heuristics select a starting tile and navigates its *left*, *right*, *up*, *down* references in an attempt to discover whether related tiles form a high-level layout. In general, several alternative layouts can be found to obtain a similar GUI from a visualisation point of view. In order to decide which layout best fits a group of tiles, the fitness functions are calculated for the group, and the layout heuristic whose fitness function is maximum is applied. It may happen that two or more functions return the highest values. In this case, the best layout is selected according to the following priority criterion: *FlowLayout*, *StackLayout*, *GridLayout*, *BorderLayout*, *VHLayout*, *HVLayout*. Next we will detail each type of layout, as well as the heuristics and fitness functions.

FlowLayout and StackLayout

A *FlowLayout* is a set of tiles arranged in a row (horizontal line). Similarly, a *StackLayout* is a set of tiles arranged in a column (vertical line). The tiles are contiguous, i.e. there cannot be a big separation between a pair of tiles. If the layout defines some kind of alignment (*horizontalAlignment* and *verticalAlignment*), all the widgets to which the layout is applied are aligned in that way.

The heuristic for the *FlowLayout* takes the top-left tile and navigates the tiles to the right until there are no more tiles. When there are several tiles to the right of a tile, only the uppermost tile is selected. For the StackLayout, the heuristic starts with the top-left tile and navigates to the bottom until there are no more tiles. When there are several tiles below a tile, only the leftmost is selected. As it has already been said, these heuristics are only applied to the content of coarse-grained tiles, since fine-grained tiles are directly mapped.

The fitness function for the *FlowLayout* obtains the percentage of tiles that can be navigated from left to right (starting with the most top-left tile). The fitness function for the *StackLayout* obtains the percentage of tiles that can be navigated from top to bottom (starting with the most top-left tile). In these functions *HoleTiles* are considered to be tiles that have not been navigated and then they reduce the fitness value.

Let us focus on the Figure 6 to show some layout examples. We can find a *FlowLayout* in the region *R2* composed of *CardLabel*, *CardCombo*, *DiscountLabel* and *DiscountCheck*. A *StackLayout* is formed by the three regions *R1*, *R2*, *R3*. In the region *R3* we could see a non-perfect match of the *FlowLayout* heuristic. Assuming that *AddButton* and *DelButton* form a fine-grained tile and *ExitButton* forms another fine-grained tile, the fitness function would return 0.66. This value is caused by the hole that exists between both tiles (2 fine-grained tiles / 2 fine-grained tiles + 1 hole).

GridLayout

This is a set of tiles arranged in a grid of n rows $\times$ m columns. The number of rows and columns may be different, but all the rows (and columns) must have the same number of tiles.

In this case the heuristic starts with the top-left tile and navigates the group of tiles from left to right and from top to bottom in a tabular way.

The fitness function returns the percentage of tiles that can be matched by a rectangular grid. It starts with the top-left tile and counts the number of tiles of the biggest grid possible. *HoleTiles* are not counted (they reduce the fitness value).

When some tiles fit a *FlowLayout* or *StackLayout*, then they also fit a *GridLayout*. For this reason, *FlowLayout* and *StackLayout* have a higher priority than *GridLayout*. There are no *GridLayouts* in the example introduced in Figure 6.

BorderLayout

This layout divides the container into five parts: left, right, top, bottom and center. The heuristic selects at most one tile for each one of the five given parts as follows. A tile t will be: in the top part if $t.vAlignment = TOP$, in the left part if $t.hAlignment = LEFT$, in the center part if $t.hAlignment = CENTER$, in the right part if $t.hAlignment = RIGHT$, and in the bottom part if $t.vAlignment = BOTTOM$. In addition, for a tile to match a part it must keep some relations with the rest of the tiles (e.g. the left tile must be below the top tile, on the left of the center tile, and above the bottom tile).

The fitness function evaluates the tiles that can fit any of the five areas predefined by a *BorderLayout*. If there is more than one tile that can fit one part, these "excess" tiles are penalised. In contrast to other layouts, a *HoleTile* is not penalised but permitted. Note that because the *FlowLayout* and *StackLayout* have a higher priority, a *BorderLayout* with emtpy parts (i.e. *HoleTiles*) that matches *FlowLayout* or *StackLayout* will be never selected. For instance, in Figure 6, the regions *R1*, *R2* and *R3* could be considered as a *BorderLayout* with top, center and bottom parts, but they are detected as a *StackLayout*.

In Figure 6, we can find an example of *BorderLayout* in the region *R3*. In that region, the widgets *AddButton* and *DelButton* are grouped in a fine-grained tile and *ExitButton* is another fine-grained tile. Thus, *AddButton* and *DelButton* are the left part and *ExitButton* is the right part of the *BorderLayout* (there are only two parts). In this case the fitness function associated with the BorderLayout returns 1 (i.e. the maximum value), so we can see that the hole has not been penalised.

HVLayout and VHLayout

An *HVLayout* is a *FlowLayout* composed of *StackLayouts*. A *VHLayout* is a *StackLayout* composed of *FlowLayouts*. An *HVLayout* can have a different number of elements in each column while in a *GridLayout* all the columns must have the same number of rows. Similarly *VHLayout* is not restricted to have the same number of elements in the lines (rows) as in a *GridLayout*.

The *HVLayout* heuristic obtains the group of tiles that have no upper tiles. From the top-left tile it navigates the tiles from the top to the bottom until there are no more tiles below, and it thus obtains the first column. The tile from the upper tiles that is next to the top-left tile is then selected and navigated to the bottom until it obtains a second column which will be to the right of the first column. This process is repeated while new columns on the right of existing ones can be found. The heuristic penalises *HoleTiles*. The heuristic for *VHLayout* is similar to the *HVLayout* heuristic but in this case it searches for rows until there are no more rows below the previous one.

VHLayout and *HVLayout* are more general than the others and may fit in most cases, in fact *VHLayout* is the most common layout found in RAD applications. On the other hand they are less specific and do not capture the visual design as well as other layouts such as *GridLayout* and *BorderLayout*. Because of this, *GridLayout* and *BorderLayout* have a higher priority than *VHLayout* and *HVLayout*, but a lower priority than *FlowLayout* and *StackLayout* since the latter are more specific.

In the example window in Figure 6, we can see a *VHLayout* in region *R1*, where there are two lines of widgets.

Unknown

If the maximum value returned by all the fitness functions is below a certain threshold (it has been set to 0.65, which means that equals or more than 65% of the elements in the group must fit the layout), then an *UnknownLayout* is created, which is a special layout that indicates that the layout of the group must be determined by the developer.

5 Implementation

This section briefly presents the tools involved in and some of the implementation details of our GUI reengineering framework, focusing particularly on Oracle Forms as the legacy technology.

The framework has been implemented on top of the Eclipse platform, and is based on the Eclipse Modeling Framework (*EMF*) (Eclipse, 2003). The metamodelling language that has been selected to represent the models and

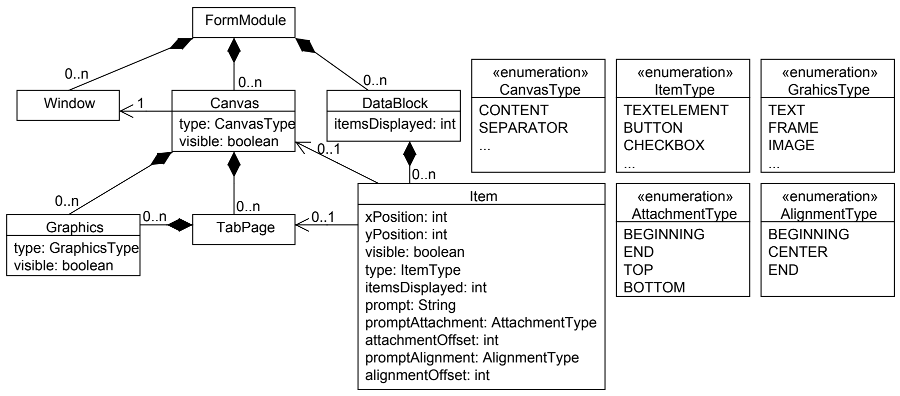


Fig. 16 Excerpt of the Oracle Forms metamodel.

metamodels is Ecore. The workflow of the reengineering process is defined and managed by a task management tool called Rake (Rake, 2012), a sort of Make for Ruby.

5.1 Injection

Let us first consider the injection step (from legacy artefacts to Source Technology models) shown in Figure 2. An injector is required for every RAD technology for which we want to migrate applications. It is worth noting that this step is particularly dependent on the source artefact format and the export facilities of the RAD environment. Some environments such as Delphi and Visual Basic use plain text files to store the GUI specification. Oracle Forms, however, uses a binary format (FMB files), but there is an export facility that generates XML files conforming to an XML schema which is available in the Oracle Developer Suite.

In our case, we have built an injector for Oracle Forms on the basis of the aforementioned XML schema. This has been done by using *EMF* which, given an XML schema, automatically generates a metamodel and the injector that takes XML files and creates models conforming to this metamodel. The Oracle Forms metamodel automatically derived by EMF mirrors the structure of the XML schema provided by Oracle, as is shown in Figure 16.

The following points summarise the structure of this metamodel.

– A *form* (*FormModule*) in Oracle Forms is a set of *Windows* with its related business logic expressed in PL/SQL triggers. The code is extracted from the XML files in a separate process, which is not within the scope of this paper.
– A *Window* can show one or several *Canvas*es, which are the panels on which the widgets are displayed. There is a special type of *Canvas* called *SEPARATOR* which can contain *TabPage*s.

– A *Canvas* is a surface that is used to display *Graphic*s and *Item*s. *Graphic*s are graphic decorators such as fixed text (*TEXT*), or graphical frames (*FRAME*). *Item*s are widgets such as buttons or text fields, which are distinguished by the *type* property. Contrary to what might be expected, *Canvas*es contain *Graphic*s but not *Item*s. *Item*s are contained in *DataBlock*s and they are associated to zero or one *Canvas*es. An *Item* will be displayed if it is associated with a *Canvas*, its *width* and *height* are greater than zero, and it has *visible* set to *true*.
– A *DataBlock* is a logical group of widgets that are often associated with columns of the same table in the database.
– The coordinates of *Item*s and *Graphic*s are relative to the *Canvas* that displays them, whereas *Canvas*es are located with absolute coordinates. In the metamodel there are no explicit relationships to specify whether a graphical frame contains other widgets or graphical frames, or whether two canvases are overlapped.
– Moreover, the *itemsDisplayed* property of *Item* indicates the number of instances of the same kind of widget that are shown. This feature is referred to as *multi-record items*, and can be regarded as a form of data grid.
– It is possible to specify a *prompt* for an *Item*, i.e. a text that is associated with the *Item*. The coordinates of the *prompt* elements are defined with regard to the associated *Item*. The *prompt* can therefore be on the left (*promptAttachment=BEGINNING*), on the right (*promptAttachment=END*), above (*promptAttachment=TOP*), or below (*promptAttachment=BOTTOM*) the *Item*, and it can also be aligned to the left (*promptAlignment=BEGINNING*), in the middle (*promptAlignment=CENTER*) or to the right (*promptAlignment=END*) of the *Item*.

5.2 Mapping Oracle Forms to RAD models

Once the source artefacts related to the GUI have been injected into a model, the latter must be transformed into a RAD model that represents the same GUI but is independent of the source technology (step from Source Technology models to RAD models in Figure 2). The RAD model can be considered as a normalised form of the source artefacts.

The Forms to RAD transformation is, in general, fairly straightforward, since there is a direct mapping between the source technology metamodel elements and the RAD metamodel elements. This mapping is summarised in Table 1.

However, Oracle Forms has some specific features which are not found in other RAD applications such as Delphi or Visual Basic. We shall now discuss two specific features that hinder the Forms-to-RAD transformation, which are prompts and multi-record items.

In some RAD environments, there is a kind of widget that is frequently called *Label* which is a piece of static text that can be placed anywhere in a window. In contrast, Oracle Forms includes a similar widget, but it also offers

Forms	RAD
Window	Window
Canvas (type=CONTENT)	PlainPanel
Canvas (type=SEPARATOR)	TabbedPanel
TabPage	PlainPanel
Graphics (type=TEXT)	Label
Graphics (type=FRAME)	WidgetGroup
Item (type=TEXTELEMENT)	TextBox
Item (type=BUTTON)	Button
Item (prompt)	Label
Item (itemsDisplayed >1)	Table

Table 1 Forms to RAD mappings.

another possibility, which is to use a *Prompt* element which is associated with a widget. The location of the *Prompt* can be expressed with regard to different reference points, which always refer to the associated widget. Specifically, the *Prompt* can be above, below, on the left or on the right of the widget and aligned to the beginning, the middle, or the end of the widget, which results in twelve possibilities. In order to calculate its coordinates it is necessary to obtain the width or height of the text of the *Prompt*, which is not easy since it depends on both the font type and the font size. Moreover, Forms by default does not express coordinates in pixels, but in proprietary measures. This implies that *Prompt* coordinates can contain a small error owing to the width/height calculation and the conversion between measures. In our case, we did not find a coordinate error greater than 8 pixels.

Another specific feature is multi-record widgets, that is, a widget that is replicated a number of times. For example, let us assume a window that must show some aspects of people's personal data. In this case, it will be necessary to have some text fields to display the name, surname and other data, and one multi-record text field could therefore be used for the name, another for the surname and so on. In current GUI technology we use data tables for this purpose. Since multi-record widgets can be scattered on the canvas and show different kinds of information, there is the challenge of deciding when certain multi-record widgets must be in the same table (i.e., each multi-record widget is a column of the table). The criterion used to group widgets in tables is the following: we group multi-record widgets that are closer than a fixed value where no non-multi-record widget exists between them. Moreover, when buttons that belong to the same datablock as multi-record widgets exist, they are also included in the table. Since this is a heuristic to group widgets in tables, developers might need to modify the RAD models in order to correctly rearrange widgets in tables.

Finally, the transformation from the source technology to RAD performs some clean up tasks. In particular, it marks the elements that are not visible and checks that widgets do not overlap. A GUI frequently includes non-visible widgets which are intended to store values that are used in transactions, so they never appear in the interface. The elements that are not visible are marked in the RAD model, so the layout algorithms ignore them. Overlapped widgets are sometimes found in applications. Developers can use overlapping to show different information with different widgets that are displayed and hidden by means of programming. Since this is not a good practice and widget overlapping hinders layout detection algorithms, overlapping is detected, and developers must fix this to ensure that the rest of the process continues properly.

5.3 Reverse engineering

With regard to the reverse engineering stage of Figure 2, the algorithms presented in Section 4 have been implemented as a chain of model-to-model transformations. To this end we have chosen the RubyTL (Cuadrado et al, 2006) language. RubyTL is a rule-based model-to-model transformation language embedded in Ruby which is integrated in the AGE environment (Cuadrado and Molina, 2007). It provides powerful query facilities, in addition to a modularity mechanism, called phasing, that has facilitated the implementation and modularisation of the solution (Cuadrado and Molina, 2009).

5.4 Forward engineering

Restructuring and forward engineering tasks are made possible with the CUI model obtained in the reverse engineering step. As part of our prototype we have implemented a generator from CUI models to Java Swing, using the Textplate code-generation language integrated into the AGE environment.

The transformation is relatively straightforward, since the CUI model represents the layout information explicitly. This also allows the original legacy GUI to be recreated using features that are only available in the target technology. For instance, the proportion of the space that is occupied by a widget or layout is used to generate resizable windows that preserve the original proportions. When a window is resized, its content (i.e. the widgets) must be resized accordingly. However we do not wish to resize all the widgets, but only those widgets that can contain or display more information if they are resized. Therefore, some widgets will be resized while others will maintain a fixed size. Widgets such as *JLabels* and *JButtons* will have a fixed size, which will be their preferred size. Widgets such as *JTextField* or *JTable* will have a variable size, which will depend on the size of the window.

It is interesting to note that further advantage can be taken of the information expressed explicitly in the CUI in order to generate a GUI in the most suitable manner for a target technology. For example, in our CUI it is possible to represent some relationships among widgets (think, for example, of a widget and its associated label) by means of the association between *WidgetConnections* in the CUI metamodel presented in Figure 4. This information can be used to define gaps between pairs of widgets, and the gap between related widgets can be made narrower than the gap between unrelated widgets.

6 Evaluation

In order to evaluate our approach and demonstrate its applicability we have applied our prototype to the GUI of two real applications in two different domains created by two different companies.

The case study A is a business management application which is intended to be used to manage the research projects and grants that are assigned to the research groups of a Spanish university. It is composed of 107 windows, which indicates a medium-high complexity. The application was developed by different developers of the same company and the conventions concerning the style of the forms were not particularly strict, signifying that there is a variety of form styles.

The case study B is a business management application targeted at being used by a department of the Regional Government to deal with budgets, income sources, expenses, investment projects and human resources. The application consisting of 57 windows has a medium complexity. Though this application was also programmed by different people, the windows follow a more strict style (imposed by the company) than in the case study A.

Both applications were developed in Oracle Forms 6 and both applications needed to be migrated to the Java platform.

6.1 Methodology

When recovering the visual appearance of a window, it frequently occurs that different layouts applied to the same widgets could result in a window with a similar visual appearance. The evaluation of our approach cannot therefore be accomplished by simply comparing the layout produced by our tool with an expected layout visually. Instead, the following steps are performed for each window:

1. The original window is manually analysed by a member of our team (different to the developer of the tool) in such a way that certain data concerning the following criteria are registered:
   – *Window parts.* Identify the parts of the window and register the relationships among them. A *part* is defined as a group of widgets that form a distinguished area of the window. A part is a set of close widgets which:
      – is visually highlighted by means, for example, of a surrounding frame or is enclosed in a coloured rectangle.
      – is distant to other groups of widgets.
      – has other parts around it to which the widgets do not belong.

     Note that in our approach, parts are normally represented with coarse-grained tiles, although this is irrelevant to the person in charge of performing the evaluation.
   – *Relationships among widgets.* The structure of the widgets within each part is identified: the position of every widget regarding the others and

the part, and the alignment which exists among them and with regard to the part.

The rationale for identifying parts is to have a layout-independent notion of the coarse-grained structure of the windows, while the relationships among the widgets are related to the fine-grained structure of the window.

2. The complete reverse engineering transformation chain is executed for the given window to obtain a CUI model. This CUI model is used to execute an additional generation step in order to obtain a Java Swing GUI, which uses the layout discovered.
3. The GUI generated is now assessed by the same person and using the same criteria as in step 1, and is compared with the data gathered from the original window. The CUI model is also analysed in order to avoid that mistakes in the Java Swing generator could mislead the evaluation, since in some cases the generated GUI had layout mistakes because of bugs in the Swing template. Two main metrics are obtained in the evaluation process for each window:
   – *Parts laid out OK.* For a part 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, also taking into account their alignment.

The criteria used to assess both the original and the generated windows are obviously subjective. In order to reduce the inconsistencies between the results, the evaluation of all the windows has been performed by the same person. 15% of the windows (with a range of complexity) were also evaluated by a second member of our team to check whether his evaluation matches to a great extent with the one carried out by the main evaluator. This aims at ensuring that the main evaluator has not introduced a strong systematic bias.

6.2 Evaluation results

The results of the evaluation of the two case studies are summarised in Table 2 and Table 3. In order to show the scalability of our approach we have classified the windows used in the evaluation into three groups, according to the number of widgets involved. As can be observed, there are a large number of small windows in both cases (63.55% for the case study A and 66.67% for the case study B) which are used as dialogs, for example, to perform searches based on certain criteria. Almost 20% of the windows in the case study A are large (an average of 86 widgets/window), and commonly use tabbed panels to arrange the widgets (an average of 4.24 canvases/window). In contrast, the case study B contains more medium-size windows (28.07%) and a few large windows (5.26%).

Figures 17 and 18 show the dispersion of the success rate (as a percentage) of our approach when identifying parts for both case studies, and Figures

	Large (>60)	Medium (20 - 60)	Small (<20)	Total
Total amount of windows	21	18	68	107
Windows of each type (out of the total)	19.63%	16.82%	63.55%	100%
Total canvases	89	19	69	177
Canvas/window average	4.24	1.06	1.01	1.65
Widgets/window average	86.00	36.43	8.10	28.15
Parts/window average	10.18	3.14	1.70	3.61
Parts laid out OK	83.24%	98.06%	100.00%	96.38%
Widgets laid out OK	87.14%	85.61%	88.10%	87.50%

Table 2 Evaluation results for the case study A.

	Large (>60)	Medium (20 - 60)	Small (<20)	Total
Total amount of windows	3	16	38	57
Windows of each type (out of the total)	5.26%	28.07%	66.67%	100%
Total canvases	6	24	38	68
Canvas/window average	2.00	1.50	1.00	1.19
Widgets/window average	65.33	37.31	7.03	18.60
Parts/window average	5.00	4.31	2.42	3.09
Parts laid out OK	89.00%	94.75%	100.00%	97.95%
Widgets laid out OK	95.87%	89.98%	97.80%	95.51%

Table 3 Evaluation results for the case study B.

19 and 20 represent the dispersion of the success rate when placing widgets. The plots also include a regression curve which expresses the tendency of the percentages when the number of elements increases. Various conclusions can be drawn from these results.

In general, the accuracy of the coarse-grained layout detection (parts) is 100% when there are few widgets and it drops when the number of widgets increases. In the case study A there are a few outliers (below an accuracy of 40%) that correspond to a special kind of layout that we have not considered (this will be commented on in the description of the non-regular layout detection in Section 6.3). In the case study B we have a high success rate (almost 98%) because the windows are better structured and the visible parts are normally surrounded by borders (frames). The errors in this case are mainly due to frames which have been emulated by forming a rectangle with four single graphical lines. This feature is not supported at present, which leads to unidentified regions. In both case studies, the recognition of parts is higher than 80% in the majority of occasions, which could be considered as an acceptable rate.

With regard to the fine-grained layout detection (widgets laid out properly), in the case study A there are several windows whose accuracy is below 80%, particularly those with less than 20 widgets. We have observed that they normally correspond to dialog windows, in which the developers place many widgets very close together in order to make the most of the available space in the dialog. We have also observed that, in the case study application, buttons are sometimes situated in a particular place simply because there is some free space there. In these cases, a small refactoring of the generated layout will lead to a cleaner GUI. In the case study B we can see that most of the windows have a certain error rate that stems from the fact that some widgets are miss-

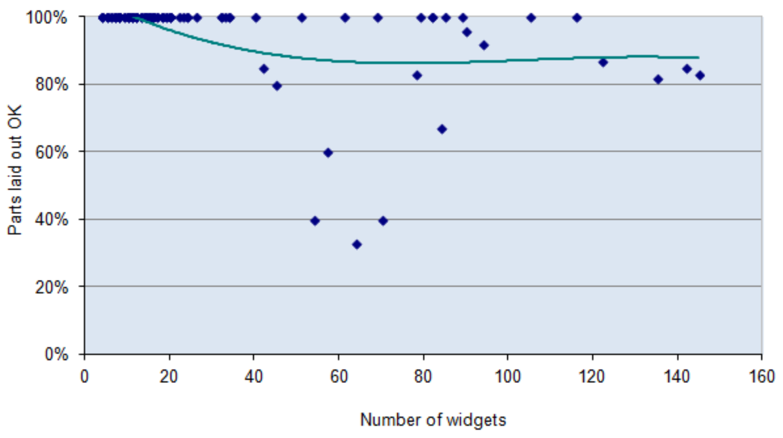


Fig. 17 Scatter plot that represents the accuracy of part detection for the case study A.

ing because they are not well-recognised in the current implementation (it is a problem of detecting the widgets to generate the RAD model). In some of the windows, mainly in the medium-size windows, we have also a slightly higher error rate since the layouts obtained do not properly reflect the holes detected (the *unidentified holes* problem will be explained in Section 6.3).

The plots show that there is not a considerable variation in the success rate neither for detecting regions nor for placing widgets in the window when the number of widgets increases, so it seems that our approach is scalable for reasonably large GUIs (in the case study A it has been applied to windows with 160 widgets per window). The rationale behind this result is that large GUIs are typically arranged in parts, either using explicit markup elements (e.g., frames) or using implicit separators such as blank spaces.

Comparing both case studies, the best results are obtained in the case study B, mainly because the windows follow a more strict style than in the case study A. The success rate of the layout of the parts in case study B is higher (6% better for large windows) mainly because the parts are surrounded by frames. In general, our approach works better when explicit markup elements are used. The improvement of the widget layout in the case study B (8% better for large windows) is because the widgets are not scattered but conform to more or less common layouts. Considering both measures together, we can claim that our approach has an acceptable accuracy rate. It is important to note that in any case, the CUI model obtained after the discovery process can be edited either to fix errors or to refactor the GUI.

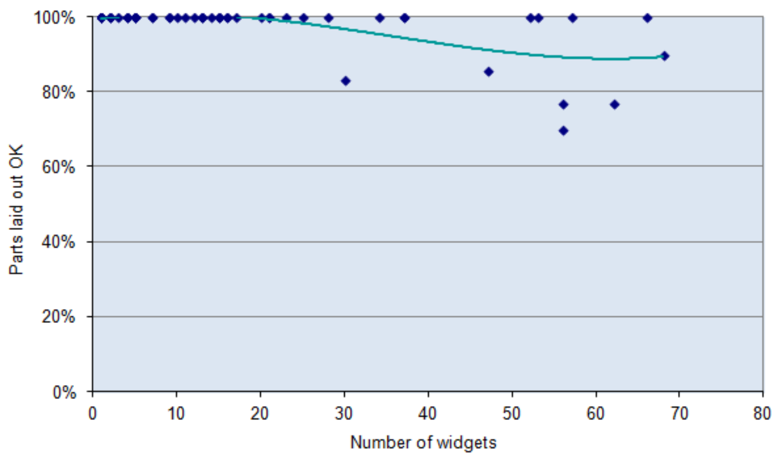


Fig. 18 Scatter plot that represents the accuracy of part detection for the case study B.

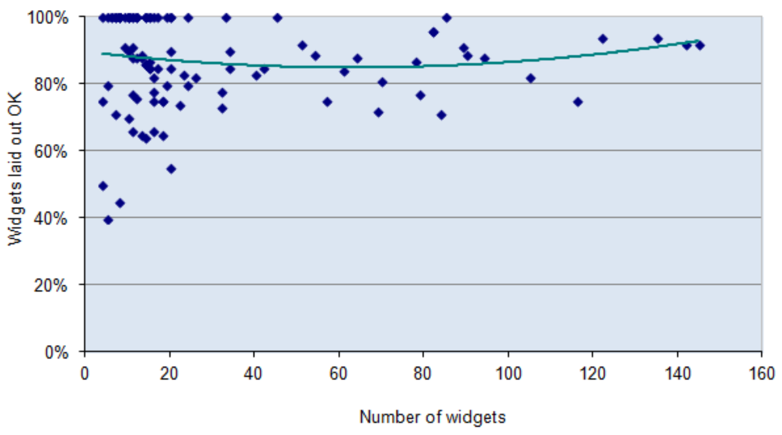


Fig. 19 Scatter plot that represents the accuracy of widget placement for the case study A.

6.3 Limitations of the approach

This evaluation has allowed us to identify a set of limitations that are not currently dealt with by our approach, and which may lead to inaccuracies in the layout recovery process.

Missing parts identification. Parts that are not explicitly limited by frames, panels or the boundaries of the window itself are not identified as regions or coarse-grained tiles. For example, in Figure 21(a) it is possible to visually identify two parts in the window because of the distance

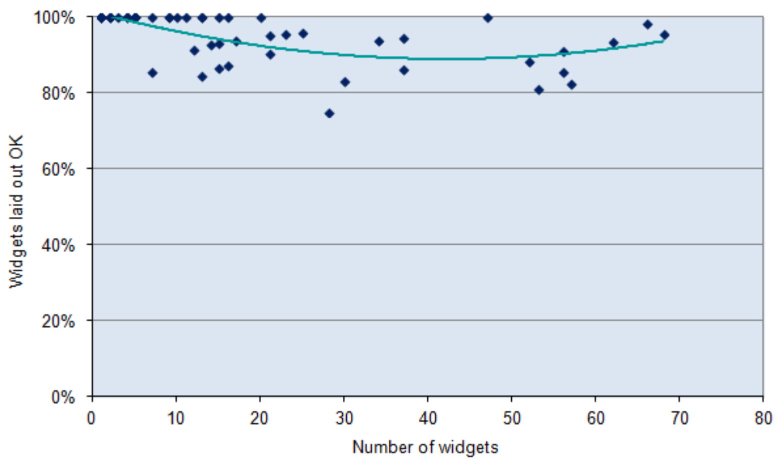
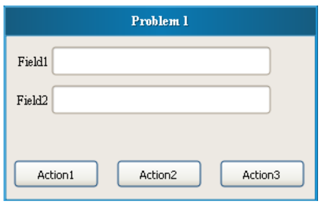


Fig. 20 Scatter plot that represents the accuracy of widget placement for the case study B.

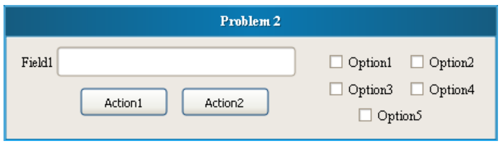
between the elements. The first is the upper part which is composed of the labels and text boxes, and the second is the lower part which is composed of buttons. In this case, our algorithm will create just one coarse-grained tile, made up of three line tiles. However, this inaccuracy does not always make the final layout incorrect, since the content of the region could be laid out correctly, as is the case of the example.

Non-regular layout detection. It sometimes occurs that there are widget arrangements that do not have a regular structure, and they cannot therefore be easily represented with a layout system. For example, the window in Figure 21(b) shows a layout that is not properly detected. There are two main problems: (1) our algorithms identify only one part, so they are not able to split the window into one part for the input and the buttons, and another part for the checkboxes, and (2) "Option 5"is not aligned with any column. In this case, an algorithm that is focused on small groups of widgets and creates a composite layout might attain better results.

Widget alignment with other widgets. Our approach uses a Tile model to arrange structures of widgets in terms of tiles that can be nested and it is possible to specify that the tiles inside a tile are aligned to the left, right, top or bottom. Nevertheless, in some cases we have found that widgets are aligned with regard to other widgets (rather than with regard to their parent container). In order to implement this feature which was not considered in the design of the solution, it is necessary to align the tiles with regard to their sibling tiles. In our evaluation, this situation occurred in a small percentage of windows, but in some cases the current implementation generated a good layout because the sibling tiles that are aligned are both aligned with regard to the same parent tile.



(a) Missing part identification problem



(b) Non-regular layout detection problem

Fig. 21 Some examples of the limitations of the approach.

Unidentified holes. Hole recognition is addressed in the algorithm that generates the Tile model, and depends on the parameters that specify the minimum distance between tiles (seen as a percentage of the width/height occupied by the tile with regard to its parent). If the parameters are set in such a way that a small distance is captured as a hole, there will be a lot of holes and the high-level layout algorithm (that generates the CUI) will not know how to deal with them. We therefore prefer to set the parameters in such a way that only notable holes are captured. This implies that some holes are not identified, but this hardly ever occurs.

On the whole we can state that our approach has an accuracy of 96% when laying out parts and an accuracy of 87% for widgets inside the parts. Simple layouts fit the arrangement of the widgets in most cases, especially the stack of flows layout. However, the problems mentioned above should be tackled if higher success rates are to be considered. These issues will be addressed in future work.

7 Related work

A great research effort has been devoted to GUI reverse engineering and a number of proposals have been published. In this section, these works are classifed in two large groups depending on whether the legacy platform is either web or desktop. Moreover, we have only considered those works that are focused on the structural aspect of a window (i.e. extracting the widgets or the layout). We have also included a third category of works which present some languages and metamodels for defining GUIs.

Desktop approaches

Our proposal is directly related to works that recover an explicit layout representation from a GUI where the layout is implicitly represented (i.e. windows with an absolute positioning of the widgets).

As far as we know, the approach presented in (Lutteroth, 2008) is the single research effort which has addressed the discovering of GUI layouts for desktop aplications. This author uses a mathematical model (the *Auckland Layout*

Model, ALM (Lutteroth et al, 2008)) which is based on linear programming, to represent the GUI layout with a high abstraction level. It defines an algorithm 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 differences between this approach and ours are the following: i) their approach is targeted at performing perfective maintenance of the GUIs, whereas we are concerned about migrating legacy applications, ii) they use a mathematical model (the ALM model) whereas we use a metamodel (MDE approach), iii) their approach does not deal with the problem of detecting the actual containment hierarchy that appears in RAD applications, iv) the ALM model defines constraints on the layout, whereas we represent the layout using a tree of high-level structures.

In (Staiger, 2007) the authors propose the use of static analysis on C/C++ code to detect the parts of the program which belong to the GUI, detect widgets and hierarchies they form, and show the event handlers connected to events of those widgets. This approach works on the source code of a programming language and its goal is to identify the code that is related to the widgets and the event handlers. By contrast, we directly focus on widget declarations to infer the layout.

Next we comment on some works that use dynamic analysis based on runtime traces. A fundamental difference with our work is that we perform static analysis on the source code that implements the GUI, and they use some kind of runtime information (dynamic analysis) such as the GUI containment hierarchy in memory. What is more, no high-level layout information is gathered, so restructuring is not possible. In (Stroulia et al, 2003), a discussion on how to migrate text interfaces by deriving state machines from screen snapshots is presented. The process is semi-automatic, and is assisted by users.

An approach with which to migrate Windows applications to Visual Basic .NET can be found in (Gerdes, 2009). Its aim is to replicate the GUI's look & feel by means of mapping runtime objects to .NET objects (explicit layout recovery is not tackled).

In (Dixon et al, 2011), the authors propose a pixel-based approach based on real-time interpretation of the GUI 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 of the interface implementation.

The work presented in (Memon et al, 2003) describes an approach that performs runtime reverse engineering to get: i) a model (called GUI forest) that represents the GUI structure, ii) event-flow graphs, and iii) an integration tree directly from the executable GUI. These models are extracted taking advantage of the information provided by layout managers (i.e., the visual containment relationship). The extracted information is then used to automatically generate test cases. Our approach could be used for similar purposes, but using static analysis instead, and it is applicable to GUIs based on absolute coordinates.

Web approaches

In (Maras et al, 2011) a semi-automatic method and a tool to extract and reuse web controls is presented. The developer interacts with controls to record the behaviour that will be required for it to work as a standalone control. In comparison with our case, we have the window code available and the widgets are much simpler than in the web context (where a widget is composed of some HTML code, CSS code and scripting code). So encapsulating and reusing widgets is not a concern, but simple mappings between technologies are enough.

In (Cai et al, 2003) an approach for extracting the web content structure based on the visual representation is proposed, which simulates how users understand web layout structure based on their visual perception. This work is related to the region detection phase of our process, but it cannot be applied to our case because it is tightly based on the nature of the HTML code, which is rather different than coordinated-based interfaces.

VAQUISTA (Vanderdonckt et al, 2001) is a tool which performs the reverse engineering of web pages into XIML (Puerta and Eisenstein, 2002) models according to flexible heuristics, and requires user interaction during the reverse engineering process. In this case, the source are web pages written in HTML4 which were laid out with tables, and the tool maps each table cell to a target element, so the table layout is replicated.

Some other related works propose the reengineering of web pages, particularly to adapt them to mobile devices. The following two works fall into this area. In (Chen et al, 2003) 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 (Bandelloni et al, 2005), 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 in 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. In contrast, as explained previously, in a GUI built with a RAD tool the layout is implicit in the widgets' coordinates, which requires analyzing the whole GUI.

Model-based approaches

The OMG's ADM initiative (, OMG) is aimed at standardising metamodels in order to represent the metadata commonly used in software system modernisation. The core of ADM is the KDM metamodel, which includes a package with which to represent the UI of an application. However, this package is too simple and must be extended to support a wide range of UIs (e.g. no widget or layout hierarchies are defined, but base metaclasses are intended to be extended). This package could be used as a CUI in our architecture, if provided with the proper extensions.

Finally, some User Interface Description Languages (UIDLs) have been proposed, such as USIXML (Limbourg and Vanderdonckt, 2004) or XIML (Puerta and Eisenstein, 2002), which allow platform-independent user interfaces to be modelled. Tools and generators have been developed for these languages. Our architecture could generate code for these languages, in order to reuse their generators. There are also some MDE frameworks to develop GUIs (for example (Meliá et al, 2008)), which could be seamlessly integrated into our architecture by writing the required model transformations.

8 Conclusions and future work

We have presented a model-driven approach to reverse engineer legacy GUIs whose layout is implicitly represented by widget positioning. The work has been focused on RAD legacy applications though it could be easily adapted to other environments. As a result, a framework for migrating GUIs implemented with RAD environments has been built. This framework has been evaluated with two real-world Oracle Forms applications by migrating a sum of 164 windows to Java Swing.

The results evidence that the solution devised can be used to infer the layout of the applications to a great extent. However, the algorithms fail to accurately detect the layout when the window is not arranged in parts (surrounded by a border) or the widgets are placed in such a way that their structure cannot be easily described by a common layout. In these cases, manual tuning of the CUI model would be required. Fortunately, the windows often follow some style patterns that contribute to make the GUI more comprehensible and usable, and for this reason our approach succeeds in most of the cases, as it has been shown.

Designing a modular solution (i.e., reusable and extensible) is a desirable quality but not an easy task. We have relied on the MDE principles to achieve it. Therefore, an important contribution of our work is the design of the transformation chain that obtains a new GUI from the legacy RAD GUI. This involves a series of metamodels (i.e. data structures) and transformations (i.e. algorithms) to shift from one representation to another.

Metamodels have been proved to be a useful formalism to represent the knowledge harvested in our reverse engineering process. In our case, the RAD and CUI metamodels are two key data structures of the migration approach which provide it with reusability as well as extensibility, given that little effort is required to change the source and/or target of the process (source and target independence). It is worth noting the originality of the RAD metamodel to get a normalised representation of the GUI of any RAD platform, and the CUI metamodel to represents a general GUI.

The reverse engineering algorithms have been implemented as model transformations based on the metamodels, which have allowed us to automate the reverse engineering process. This process has been split into several transformations of a relatively reduced complexity, therefore leading to a modularised solution. This work also permitted to try out the usage of model-to-model transformation languages to implement reverse engineering algorithms. Related to this, reverse engineering algorithms have required intensive use of model queries and imperative statements, which we have found that not all the languages are suitable to perform. Particularly RubyTL (the model transformation used) has been proved to be useful as it meets our needs.

As future work, we will improve the region detection by considering that a group of widgets which is visually apart from other groups can be a region. The widget recognition can be improved to deal with widgets that are scattered within a part or that are aligned with regard to other widgets. New layout types will also be included in the framework. We also intend to include some other aspects related to GUI migration in our architecture, such as event handling and navigation flows. We are additionally interested in exploring to what extent our architecture can be adapted to deal with the migration of web-based GUIs. Finally, new source and target platforms, and new restructuring algorithms will be considered.

References

Andrade LF, Gouveia J, Antunes M, El-Ramly M, Koutsoukos G (2006) Forms2net - migrating oracle forms to microsoft .net. In: GTTSE, pp 261–277

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

Bézivin J, Kurtev I (2005) Model-based technology integration with the technical space concept. In: In: Proceedings of the Metainformatics Symposium, Springer-Verlag, Springer-Verlag

Cai D, Yu S, Wen JR, Ma WY (2003) Vips: a vision-based page segmentation algorithm. Tech. rep., Microsoft Research

Chen Y, Ma WY, Zhang H (2003) Detecting web page structure for adaptive viewing on small form factor devices. In: WWW, pp 225–233

Chikofsky EJ, Cross JH (1990) Reverse engineering and design recovery: A taxonomy. IEEE Software 7(1):13–17

Clark T, Evans A, Sammut P, Willans J (2004) Applied Metamodelling - A Foundation for Language Driven Development, 2nd edn. Ceteva

Cuadrado JS, Molina JG (2007) Building domain-specific languages for model-driven development. IEEE Softw 24(5):48–55

Cuadrado JS, Molina JG (2009) Modularization of model transformations through a phasing mechanism. Software and System Modeling 8(3):325–345

Cuadrado JS, Molina JG, Menárguez M (2006) RubyTL: A practical, extensible transformation language. In: 2nd European Conference on Model-Driven Architecture, Springer, LNCS, vol 4066, pp 158–172

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

Eclipse (2003) Eclipse Modeling Framework Project (EMF). http://www.eclipse.org/modeling/emf/

Gerdes J Jr (2009) User interface migration of microsoft windows applications. J Softw Maint Evol 21(3):171–187

Harrison JV, Lim WM (1998) Automated reverse engineering of legacy 4gl information system applications using the itoc workbench. In: Proc. of the 10th Conference on Advanced Information Systems Engineering, CAiSE'98, pp 8–12

Jacobs C, Li W, Schrier E, Bargeron D, Salesin D (2003) Adaptive grid-based document layout. ACM Trans Graph 22(3):838–847

Li W, Kurata H (2005) A grid layout algorithm for automatic drawing of biochemical networks. Bioinformatics 21(9):2036–2042

Limbourg Q, Vanderdonckt J (2004) Usixml: A user interface description language supporting multiple levels of independence. In: ICWE Workshops, pp 325–338

Lutteroth C (2008) Automated reverse engineering of hard-coded gui layouts. In: Ninth Australasian User Interface Conference (AUIC 2008), ACS, vol 76, pp 65–73

Lutteroth C, Strandh R, Weber G (2008) Domain specific high-level constraints for user interface layout. Constraints 13(3):307–342

Maras J, Štula M, Carlson J (2011) Reusing web application user-interface controls. In: Proceedings of the 11th international conference on Web engineering, Springer-Verlag, ICWE'11, pp 228–242

Martin J (1991) Rapid application development. Macmillan Publishing Co., Inc.

Meliá S, Gómez J, Pérez S, Díaz O (2008) A model-driven development for gwt-based rich internet applications with ooh4ria. In: ICWE, IEEE, pp 13–23

Memon A, Banerjee I, Nagarajan A (2003) 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, p 260

(OMG) OMG (2007) Architecture-Driven Modernization (ADM). http://adm.omg.org/

Puerta A, Eisenstein J (2002) Ximl: a common representation for interaction data. In: IUI '02: Proceedings of the 7th international conference on Intelligent user interfaces, ACM, pp 214–215

Rake (2012) Rake website. http://www.rake.org/

Sánchez Ramón O, Sánchez Cuadrado J, García Molina J (2010) Model-driven reverse engineering of legacy graphical user interfaces. In: Proceedings of the IEEE/ACM international conference on Automated software engineering, ACM, ASE '10, pp 147–150

Staiger S (2007) Static analysis of programs with graphical user interface. In: Proceedings of the 11th European Conference on Software Maintenance and Reengineering, IEEE Computer Society, CSMR '07, pp 252–264

Stroulia E, El-Ramly M, Iglinski P, Sorenson P (2003) User interface reverse engineering in support of interface migration to the web. Automated Software Engg 10(3):271–301

Tilley SR, Smith DB (1995) Perspectives on legacy system reengineering. Tech. rep., Software Engineering Institute, Carnegie Mellon University

Vanderdonckt J, Bouillon L, Souchon N (2001) 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, p 241