# GUI visual aspect migration: a framework agnostic solution

Benoît Verhaeghe, Nicolas Anquetil, Anne Etien, Stéphane Ducasse, Abderrahmane Seriai, Mustapha Derras

**Noname manuscript No.**
(will be inserted by the editor)

# GUI visual aspect migration: A framework agnostic solution

**Benoît Verhaeghe[1,2,3] · Nicolas Anquetil[3,2] · Anne Etien[3,2] · Stéphane Ducasse[2,3] · Abderrahmane Seriai[1] · Mustapha Derras[1]**

Received: date / Accepted: date

**Abstract** With the generalization of mobile devices and Web applications, GUI frameworks evolve at a fast pace: JavaFX replaced Swing, Angular 8 replaced Angular 1.4 which had replaced GWT (Google Web Toolkit). This situation forces organizations to migrate their applications to modern frameworks regularly so they do not become obsolete. There has been research in the past on automatic GUI migration. However, and concurrently, large organisations' applications use many different technologies. For example, the IT company with which we are working, Berger-Levrault, wishes to migrate applications written in generic programming language (Java/GWT), proprietary "4th generation" languages (VisualBasic 6, PowerBuilder), or markup languages (Silverlight). Furthermore, one must expect that in a few years time,

B. Verhaeghe
E-mail: benoit.verhaeghe@berger-levrault.com

N. Anquetil
E-mail: nicolas.anquetil@inria.fr

A. Etien
E-mail: anne.etien@inria.fr

S. Duscasse
E-mail: stephane.ducasse@inria.fr

A. Seriai
E-mail: abderrahmane.seriai@berger-levrault.com

M. Derras
E-mail: mustapha.derras@berger-levrault.com

[1]Berger-Levrault, Montpellier, France
[2]RMod team, INRIA Lille Nord Europe, Villeneuve d'Ascq, France
[3]Université de Lille, CNRS, Inria, Centrale Lille, UMR 9189 – CRIStAL, France

new frameworks will appear and new migrations will be required. Thus, there is a need for a language-agnostic migration approach allowing one to migrate various legacy GUI to the latest technologies. None of the existing solutions allow to deal with such a variety of GUI framework. They also typically focus on a subpart of the migration (*i.e.* how to extract a specific GUI framework) ignoring the re-engineering/forward-engineering part of the migration (which is straightforward for a single technology). This makes it difficult to adapt these solutions to other GUI frameworks. We propose an approach to migrate the GUI part of applications. It is based on meta-models to represent the visual element structure and layout. We detail how to create both the GUI extractors and generators, with guidelines to support new markup and programming languages. We evaluate our approach by implementing three extractors and generators for web-based or desktop-based user interfaces defined with generic programming languages (Java, Pharo, TypeScript) or markup languages (XML, HTML). We comment case studies on five applications, opened and closed source, of different sizes. The implementations of our generic approach detect 99% of the widgets and identify (*i.e.* determine the type of the widget) 97% of them. We give examples of the migrated GUIs, both successful and not.

**Keywords** Graphical User Interface · Visual Part · Model-Driven Engineering · Migration

## 1 Introduction

On the one hand, old Graphical User Interface (GUI) frameworks are not supported anymore: the last major version of GWT was in 2009. On the other hand, recent GUI frameworks evolve fast: two major versions of Angular[1], three major versions of React.js[2], four versions of Vue.js[3], and three versions of Ember.js[4] were released in 2018.

Some companies invested massively to create complex applications in old frameworks. It is the case of Berger-Levrault with applications totaling more than 500 web pages. It makes it impossible to modernize the application by re-developing it from scratch. Nowadays, the company needs to migrate applications written in generic programming languages (Java/GWT), proprietary "4th generation" languages (VisualBasic 6, PowerBuilder, WebDev), or markup languages (Silverlight). Thus, we need to provide a generic approach that allows companies to migrate the GUI visual aspect among several GUI frameworks. It is also clear that, in the near future, new frameworks will appear and new migrations will be required. So, detailing how to adapt the GUI migration approach to different GUI frameworks is mandatory.

[1] https://angular.io/

[2] https://reactjs.org/

[3] https://vuejs.org/

[4] https://emberjs.com/

Tools and approaches have been proposed to support GUI migration (Fleurey et al., 2007; Joorabchi and Mesbah, 2012; Samir et al., 2007; Shah and Tilevich, 2011; Włodarski et al., 2019). However, the authors did not apply their migration approaches on multiple GUI frameworks, and specifically on web-based GUIs. Whereas authors detailed migration high-level steps (*e.g.* extraction and generation) or focused on migration specific aspects inherited from their work context, none explained how to adapt their approach and meta-model to several migration projects.

So, to enable the migration of several GUI using a variety of frameworks, we need to: (1) define a framework-agnostic approach, (2) design a specializable meta-model to support the differences between frameworks (*e.g.*, different widgets) and (3) detail how to apply the approach and the meta-model on different kind of languages (*e.g.*, markup or programming languages).

In this paper, we present an approach to migrate GUIs of web-based or desktop-based software systems, defined with GUI framework using markup (*e.g.* XML, HTML) or programming languages (*e.g.* Swing, Spec). The approach comes with meta-models which represent the GUIs structure and layout. We detail the steps to extract GUIs and generate the target applications. Note that this paper presents an extension of previous work (Dutriez et al., 2019; Verhaeghe et al., 2019) where only GUI structure, and not the layout, was considered for two migration projects (from GWT to Angular, and from Spec to Spec2), without a detailed framework-agnostic approach.

To validate this approach, we developed Casino, a tool that can migrate several GUI frameworks to others. We detail implementations of our approach to extract GUIs defined with a programming language (Java/GWT, Pharo/Spec), and with a markup language (Java/GXT) and to generate GUIs defined with a programming language (Pharo/Seaside and Pharo/Spec2), and with a markup language (TypeScript/Angular). Note that other combinations are possible, for instance, we also worked on plain HTML, Java Swing, and Silverlight extractor, and on an Aurelia generator[5]. We released the source code of the existing implementations and future ones in GitHub[6]. Then, we validate our approach on two industrial and three open-source projects. Our approach detects 99% of the widgets (*i.e.* "this element is a widget of the GUI") and identifies 97% (*i.e.* "this widget is a button") of them.

Our approach migrates the visual aspect of the original applications. Behavioral aspects (treatments occurring when the end-user interacts with the GUI) are out of the scope of this paper. They will be presented in future publications.

The contributions of the paper are:

- a detailed approach to migrate application visual parts independently of the GUI implementation language;
- meta-models to represent GUIs structure and layout;
- a tool that implements our approach; and

[5] https://aurelia.io/

[6] https://github.com/badetitou/Casino

– implementations of GUI frameworks migration.

In Section 2, we review the literature on GUI migration. In Section 3, we present our generic migration approach. In Section 4, we detail our GUI meta-models. In Section 5, we present implementations of our approach for programming and markup languages. In Section 6, we present the results produced by applying our approach to five real applications. In Section 7, we discuss our results. In Section 8, we conclude and present future work.

## 2 State of the art

We will first rapidly mention recent work on GUI generation using Artificial Intelligence (from screenshot examples). This is the case of Beltramelli (2017); Chen et al. (2018); Moran et al. (2018). These approaches rely on a huge dataset of screenshot examples (14,382 screenshots for (Moran et al., 2018) and 10,804 for (Chen et al., 2018)), to train the model. Thus, Beltramelli (2017) warns that the approach "is not, in any way, intended, nor able to generate code in a real-world context" and "both the source code and the datasets are provided to foster future research [...] and are not designed for end-users"[7]. Consequently, we rule out Artificial Intelligence as a possible approach given the current state of the art.

We identified various publications related to GUI migration using meta-model (Fleurey et al., 2007; Samir et al., 2007). Section 2.1 details existing related work. Section 2.2 presents the approaches proposed to migrate GUI application. Section 2.3 describes the user interface meta-models found in the literature.

### 2.1 Related work

We are interested in a generic GUI migration approach able to handle multiple source and target frameworks. Thus we are interested in whether the proposed solution can (i) import GUI from markup languages (*e.g.* HTML), (ii) import GUI from programming languages (*e.g.* Swing); (iii) import from binary source (*e.g.* Oracle form); (iv) handle multiple languages; (v) export GUI to markup language; (vi) export GUI to programming language. We will not consider the case of exporting to binary framework as no modern GUI framework use this approach anymore.

Prior research makes valuable contributions: GUI internal representation (models, see Section 2.3) and/or migration "process" (see Section 2.2). However, there are rarely enough details provided to generalize the approaches to other languages/frameworks.

Table 1 summarizes the related work considered. First line presents our need for the extraction, exportation, and multi framework support. Note that extract binary is part of our need but will not be detailed in this article.

[7] https://github.com/tonybeltramelli/pix2code#disclaimer

Table 1: Related work

| | **extract markup language** | **extract programming language** | **extract binary** | **multiple migration** | **export markup language** | **export programming language** |
|---|---|---|---|---|---|---|
| *Our needs* | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| Hayakawa et al. (2012) | ✓ | | | ✓ | ✓ | |
| Mesbah and van Deursen (2007) | ✓ | | | | ✓ | |
| Bragagnolo et al. (2020a) | | | ✓ | | ✓ | |
| Garcés et al. (2017) | | | ✓ | | ✓ | |
| Sánchez Ramón et al. (2014) | | | ✓ | | | ✓ |
| Fleurey et al. (2007) | | ✓ | | | ✓ | |
| Samir et al. (2007) | | ✓ | | | ✓ | |

Most of past research considered migrating from markup languages: Garcés et al. (2017) (partial XML to JEE), (Hayakawa et al., 2012) (multiple markup languages), (Mesbah and van Deursen, 2007) (multi-page web application to Single Page Application (SPA) using Ajax). Migrating from markup languages is easier because the language is simple to parse (*e.g.* there are numerous parsers for HTML or XML), detecting the GUI elements (widgets) is straightforward (*e.g.*, a tag <button>), and the structure of the interface is well described by the DOM.

Sánchez Ramón et al. (2014) and Garcés et al. (2017) considered the case of Oracle Forms, a framework that we classify as binary source since there is no textual representation of the GUI (or an incomplete XML representation Garcés et al. (2017, again)). Bragagnolo et al. (2020a) also worked on GUI extraction based on binary sources with the migration of Visual Basic applications. Sánchez Ramón et al. (2014) consider migrating to Java Swing (programming language), Garcés et al. (2017) to JEE application (markup language because the GUI is defined in HTML files), and Bragagnolo et al. (2020a) to Angular (markup language). The publications focus on the extraction part of the process since Rapid Application Development frameworks have specific problems to get access to a representation of the GUI. We will come back on this specific issue in Section 7. Their GUI meta-models are valuable (see 2.3) as well as their generic process (see 2.2), however, there are not enough details to generalize them to other languages.

Fleurey et al. (2007) and Samir et al. (2007) are the only ones who consider the extraction of GUI based on programming language. The first one from

Coolgen generated code [8] and the second one from Java Swing code. Even if none of the publications detail how to adapt the approach to extract other programming language GUI, they give hints on the general approach such as how to map source and target widgets. The two publications migrate to markup language based GUI: Fleurey et al. (2007) migrate to J2EE, and Samir et al. (2007) to Ajax Web with XUL[9]. We note that the XUL format has been discontinued.

Thus, none of the presented projects deal with the multi-framework migration defined with markup and programming languages. And, only Fleurey et al. (2007) and Samir et al. (2007) present the extraction of GUI defined using a programming language.

### 2.2 Migration approach

Fleurey et al. (2007); Garcés et al. (2017) and Sánchez Ramón et al. (2014) developed tools that semi-automatically migrate GUI. All of them use the following four steps migration "process":

1. Generation of a model of the original application.
2. Transformation of this model into a pivot model that includes data structures, actions, user interface, and navigation.
3. Transformation of the pivot GUI model into a target framework model.
4. Generation of the target source code.

Hayakawa et al. (2012) proposed a similar approach in two steps: (a) reverse engineering similar to the extraction of a pivot model, and (b) generation of the new application similar to the generation of target source code. They can reduce the number of steps in part because they worked only with GUI defined with markup languages that are easier to parse and generate.

In addition to those main steps, Mesbah and van Deursen (2007) added a behavior extraction step and a behavior generation step. This step allows them to migrate the navigation flow and the behavior of the User Interface.

The proposed approaches provide good results in all case studies. The generic "process" in four steps seems consensual enough and we will base our solution on it. However, none of the authors had to adapt their respective approach to multiple sources or target frameworks. As a result they focused only on parts of the migration problem (Sánchez Ramón et al., 2014) which makes their approaches difficult to adapt to other migration projects.

On the other hand, some (*e.g.* Mesbah and van Deursen (2007)) considered issues that are out of the scope of this paper. For instance, they also focus on migrating the behavioral, non GUI related, part of the code. Even if the behavior is out of the scope of visual aspect migration, it is represented in the literature GUI meta-models as *events* to enable future work focused on this aspect.

[8] Coolgen: https://en.wikipedia.org/wiki/CA_Gen

[9] https://developer.mozilla.org/en-US/docs/Archive/Mozilla/XUL

Most approaches use the horseshoe process (Kazman et al., 1998) based on meta-models. In the following, we discuss some of the proposed GUI meta-models.

All researchers used a hierarchical representation of the GUI in the form of a Domain Object Model (DOM)(Brambilla and Fraternali, 2014; Fleurey et al., 2007; Gotti and Mbarki, 2016; Joorabchi and Mesbah, 2012; Memon et al., 2003; Mesbah et al., 2012; Samir et al., 2007; Shah and Tilevich, 2011). Each node in the DOM tree represents a *widget* of the user interface. Thus, this representation is not controversial and representing the DOM appears as a good solution to represent the GUI skeleton.

In addition to the DOM and the widgets, some authors added *attributes* (Garcés et al., 2017; Gotti and Mbarki, 2016; Joorabchi and Mesbah, 2012; Memon et al., 2003; Samir et al., 2007; Shah and Tilevich, 2011) and *events* (Fleurey et al., 2007; Garcés et al., 2017; Joorabchi and Mesbah, 2012; Mesbah et al., 2012; Samir et al., 2007). Attributes are used to customize the visual aspect of the widgets, and the events enable the representation of the navigation inside the GUI. Thus, attributes and events are important to have a detailed GUI meta-model.

Gotti and Mbarki (2016) propose an approach to extract a detailed GUI representation from Java Swing code. To do so, and additionally to the DOM and the attributes, they identify different kinds of known widgets such as Button, Label, Panel, *etc.* With the specific widgets concepts, they can better map each widget of the source code to their GUI model.

Sánchez Ramón et al. (2016) also define kinds of widgets such as Panel, TextBox, DataGrid. They also define a special kind of widget named "custom" to support widgets that might not be recognized by their GUI extraction approach.

To represent correctly the visual aspect of a GUI, a layout representation is also necessary (Rodríguez-Echeverría et al., 2011; Sánchez Ramón et al., 2016; Verhaeghe et al., 2019). There are three identified layout managers in the literature: hardcoded, hierarchical, and constraint-based:

- **Hardcoded layout.** Sánchez Ramón et al. (2016) define for each widget its position with absolute coordinates on the screen. It is used in old GUI frameworks.
- **Hierarchical layout** consists of subdividing the available space of the screen into panels. Then the panels are responsible for placing their children in the dedicated space (Hasselknippe and Li, 2017). Sánchez Ramón et al. (2014) proposed a layout meta-model that supports hierarchical layouts. Zeidler et al. (2012) claims that the grid-bag layout, which is a hierarchical layout, is the most prominent and that it is supported by almost all available GUI toolkits.
- **Constraint-based layout** also uses a hierarchical structure but it uses constraints to place the widgets, for example: "place this button on the

right of this text". Lutteroth et al. (2008) presented the Auckland Layout Model which is an implementation of a constraint-based layout.

The literature presents several GUI meta-models composed of a structure meta-model and a layout meta-model. Authors have also proposed modifications to adapt their meta-model to their specific context. Only Sánchez Ramón et al. (2016) present the concept of custom to deal with unknown widgets of RAD frameworks. However, to the best of our knowledge, no study presents how those meta-models can be specialized for other contexts. Thus, we need to define a specializable GUI meta-model that will be tunable.

## 3 Framework-agnostic migration approach

The GUI migration problem has already been studied in different migration projects. However, there is no detailed framework-agnostic approach to migrate GUI visual aspect. In the following, we detail our approach and how to use it with markup and programming languages. Note that we do not consider the extraction of GUI defined with binary file and it will be part of our future work.

In Section 3.1, we give an overview of our approach, then we detail the extraction step for different kind of GUI frameworks (Section 3.2), and the generation step (Section 3.3).

### 3.1 Our Migration Process

We designed a three-step approach for GUI migrations. Each step is divided into tunable sub-steps to enable multi-framework support. Examples of sub-step adjustments are illustrated with concrete cases in the following sections.

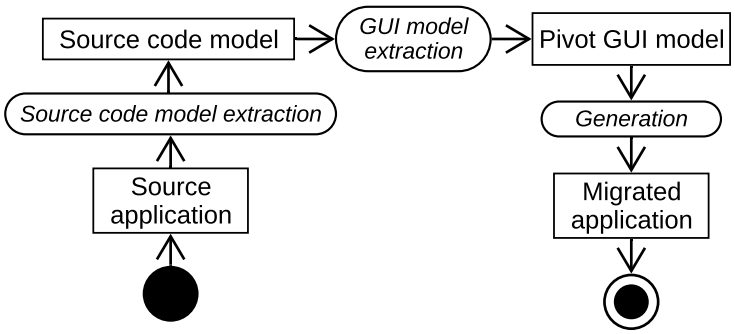



Fig. 1: Our GUI migration process

The process represented in Figure 1, is divided into the three following steps:

*Source code model extraction.* We build a model that represents the source code of the original application. To do so, one needs a source language parser and its meta-model. The source language can be a programming language or any other markup languages like XML or HTML. One can also extract a model from a binary file format.

*GUI model extraction.* We analyze the source code model to identify the *Visual part* elements. We build a mapping between the widgets of the source framework and the widgets of the GUI meta-model. Finally, we create a GUI model from this mapping.

*Generation.* We re-create the GUI in the target language. First, we have to define how the user interface is implemented in the target framework. Then we have to define a mapping between the widgets of the GUI model and their implementations in the target framework. We show examples of generators in Section 5.2.

Our approach is similar to the four steps approach used in the literature. But, we merged the last two steps considering the generation of the target model and code more trivial.

In the following, we present a concrete case of migration from Java/GWT project to Angular. First, we *extract a java model* from the project source code. Then, we *extract a GUI model* from the Java model. It produces a model of the GUI of the source application, *i.e.*, the widgets and their position. Finally, we *generate* the Angular application from the GUI model. It produces a runnable version of the application including only the GUI elements in the target language.

## 3.2 Extraction

For this step, we must extract the widgets, their attributes, and their layout and link them together. We must also be able to deal with custom widgets defined by the developers and used in their applications. This step depends on the GUI framework used. There are some "configuration" sub-steps to be performed to support a new GUI framework. In the following, we describe first the sub-steps applied (once) for each new GUI framework. Then we describe the sub-steps applied for each application of an already known framework.

### *3.2.1 New framework extraction*

To support a new source framework (see Table 2) we define the following sub-steps:

1. **Map source framework to pivot model**, where we define dictionaries mapping known widgets, attributes, layouts to their counterpart in our

Table 2: Framework agnostic extraction approach - support new framework

| source | Map source to pivot | Indentify containment rules | Indentify custom widgets rules | Indentify root widgets rules |
|---|---|---|---|---|
| Markup language | Tag to widget concept (Garcés et al., 2017; Hayakawa et al., 2012; Sánchez Ramón et al., 2014) | Use DOM (Memon et al., 2003; Samir et al., 2007) | Unknown tag | Configuration file (Garcés et al., 2017; Memon et al., 2003; Mesbah and van Deursen, 2007) |
| Programming language | Source class and factory (Samir et al., 2007) | Method invocation | Unknown subclasses | Specific set of widgets (Memon et al., 2003; Rodríguez-Echeverría et al., 2011) |

In parenthesis: related work for the step.

pivot model (presented in Section 4). For programming languages, we map a widget class to our pivot counterpart. For example, in Swing, JButton maps to our pivot `Button` widget. For markup language, we map a tag to a widget concept. For example, in HTML, the label attribute maps to our pivot `Label` attribute

2. Then, we need to **identify containment rules** in the source code, containment links between widgets/layout and their children attributes/widgets. In the case of programming language, such containment links may come from specific method calls as add(...) or setWidget(...) on a container widget. For markup language, the containment is already defined in the DOM
3. **Identify custom widgets rules** specifies how to identify application specific widgets that are not part of the *source to pivot* dictionary map (first step). For programming languages, it corresponds to unknown subclasses of the generic widget class. For example, the rule for GWT is to look for all subclasses of the Widget class; in Angular, one looks for all component.ts files. In case of markup language, we look for unknown tags. Such custom widgets can typically not be translated automatically but need to be identified. Thus, the generator flags them in the generated GUI for developers to take actions (either to migrate them manually, or update the *source to pivot* map).
4. Finally, **identify root widgets rules** specifies how root widgets will be recognized. Root widgets are the entry points in the GUI of the application; they correspond to windows in a desktop application or pages in a web application. For markup language, roots are defined in a configuration file, whereas in programming languages, they are identifiable as a specific set of widget (*i.e.* `JWindow` in Java Swing). We will see later that they are important to create a hierarchical representation (a DOM) of a GUI. Note that we only care about custom widgets and not custom attributes or

layout. We consider that it is not possible to define custom attributes (that would apply to already known widgets) or custom layouts. Such new attributes/layouts can only come as part of new custom widgets.

Table 3: Substeps to migrate an application using a known framework

| **Source** | **Identify Custom Widget** | **Create widgets instances** | **Detect Composition** | **Perform layout additional substep** |
|---|---|---|---|---|
| Markup language | | Browse markup file and create widgets when encounter recognized tags | Children in the GUI are children in the DOM (Memon et al., 2003; Samir et al., 2007) | — |
| Programming language | Apply Custom widgets identification rules | Look for source class instantiation or call to factories (Rodríguez-Echeverría et al., 2011; Silva et al., 2010) | Look for call to pre-determined methods | Use symbolic execution to resolve precise position of widget (Silva et al., 2010) |

In parenthesis: related work for the step.

### *3.2.2 New application extraction*

Once our approach is "configured" to support a GUI framework, migrating an application for this framework consists in four sub-steps (see Table 3).

1. First, **identify custom widgets** based on the rules for the framework (*custom widget rules*). For both markup and programming languages, it corresponds to applying the defined custom identification rules. The new widgets are added, on the fly, to the framework *source to pivot* dictionary and mapped to a special pivot `Custom Widget`. Each *instance* of an unknown widget is mapped to a different instance of `Custom Widget`. No effort is made to group together various instances of the same unknown widget.
2. Second, in the **create widget instances** sub-step, the *source to pivot* dictionary for the framework is used to identify all instances of known widgets, attributes, and layouts. In case of markup language, we visit the markup source file and create widgets corresponding to recognized tags. For programming languages, we look for widget class instantiation. It can occur by calling the widget constructor (*i.e.* in java: `new`) or through a factory defined by developers or the source framework.

3. Third, **detect composition**, each instance of widget, attribute, and layout is linked to its parent widget following the *identified containment rules* of the framework. In case of markup language, the composition is already defined in the DOM (Document Object Model) so children in the source DOM are children in the pivot DOM. For programming languages, we look for call to methods defined in the *containment rules* (*i.e.* `add(...)`, `setWidget(...)`). This results in a DOM that includes the widgets, their attributes, and also the layouts.
4. Finally, **performing a layout additional** sub-step is often necessary to improve the computation of widgets layout (typically with grid layouts). For example, widgets could be positioned one relative to the other, or some computation might be required to get the row and column values in a grid layout. We did not report this sub-step for markup languages since the DOM building is sufficient to get a good representation of the layout. However, for programming languages, this sub-step is necessary and might require symbolic execution to resolve the position of widgets in the GUI.

### 3.3 Generation

Table 4: Framework agnostic generation approach - support new framework

| | **Identify target framework environment** | **Map pivot model to target framework** |
|---|---|---|
| Markup | Written in one or multiple files | Widget concept to tag |
| Programming | Defined in one method or multiple methods defined by the target framework | Widget concept to widget instantiation method |

There is no study nor detailed explanation on how to export GUI into the target language. The basic approach consists in visiting the DOM of the pivot model, generating appropriate code. However, the generated code may be split into different files, or one pivot entity may produce several target entities, or several pivot entities may be grouped in only one target entity. As for the extraction we first present how to "configure" our approach for a new target GUI framework, then we discuss actual generation for an application in a known framework.

Two sub-steps are needed to support a new framework (see Table 4): **identify target framework environment** and **map pivot model to target framework**.

1. The first sub-step, **identify target framework environment**, defines where the code will be generated to be supported by the target GUI framework. For instance, for markup languages, the GUI can be fully defined in

a file (*.html*) or in multiple files (angular component). In the case of programming languages, some frameworks force the user to define the GUI in a specific method or the GUI can be defined at any place in the code (*Java Swing*).

2. In the second sub-step, **map pivot model to target framework**, we define a mapping between widgets, attributes, and layouts to their target framework counterpart. For markup languages, it corresponds to the target tag, and for programming languages, the way to instantiate the widget or set the attribute (*i.e.* calling the constructor or a factory).

Table 5: Framework agnostic generation approach - support new application

| | **Identify target application environment** | **Generate Code** |
|---|---|---|
| Markup | Configuration information (URL for web application data access) | Visit the GUI Model DOM and generate for each widget/attribute/layout/ its tag counterpart |
| Programming | | Generate the GUI code using setter, DOM builder methods, and constructor |

Then, two other sub-steps are needed to support the migration of a new application (see Table 5): **identify target application environment** and **generate code**.

1. **Identify target application environment** is identical for programming and markup languages. It consists of discovering the configuration needed by the application. For example, it includes the endpoints URL to access data.
2. The second sub-step, **generate code**, defines how the code is exported. This sub-step can be generalized at the "support new framework" level but must be performed for each application. For instance, for markup languages, it is possible to visit the GUI model DOM and generate for each widget its target language counterpart with its attributes. In the case of programming language, it calls methods that instantiate the widgets (*e.g.* call to the constructor, call to a factory, *etc*) and the methods used to build the DOM (*e.g.* `add()`, `setWidget()`) to generate the target GUI.

## 4 Specializable GUI meta-model

Our approach uses a pivot GUI meta-model (Fleurey et al., 2007; Garcés et al., 2017). This meta-model is defined as the composition of several meta-models, each representing different elements of the visual part. We define three meta-models: (1) Core (Section 4.1) represents the structure of a user interface,

(2) Layout (Section 4.2) represents the visual disposition of user interface's widgets, and (3) Widgets (Section 4.3) represents the possible widget types. Finally, in Section 4.4, we present how the meta-model can be specialized and the importance of custom widgets.

## 4.1 Core meta-model

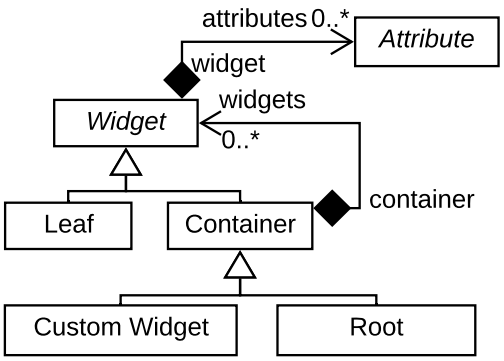



Fig. 2: Core meta-model

To represent the user interfaces of desktop or web-based applications, we designed the meta-model presented in Figure 2. The core represents the structure of a user interface. Developers can then tune the meta-model by adding new entities to fit their requirements such as a complex parametrizable table component. It also introduces the concept of custom widget that will be used in Section 4.4.

`Widget` is a graphical resource. It can be refined as `Leaf` or `Container`.
`Container` is a composite of `Widgets`.
`Leaf` is a basic widget that can not contain another widget. For example, a text input.
`Root` represents the main container of a graphical interface. It is either a window of a desktop application or a web page. The `Root` is a kind of `Container`.
`Attribute` represents a widget property and can change its behavior. For example, a *button* may have a *text* attribute.
`Custom Widgets` is a kind of `Container` that represents an unknown widget in our meta-models. During the migration process, it represents a detected but not recognized widget.

The DOM, massively used in the literature (see Section 2.3), is represented with the relation between `Container` and `Widget`. To represent the widget visual disposition, we introduced a layout meta-model (see Section 4.2) that represents the DOM with additional information such as how children are visually disposed inside their parent.

## 4.2 Layout meta-model

To represent the layout of a graphical user interface, we design a dedicated meta-model. It allows one to have a better representation of the visual disposition of the graphical components of the user interface.

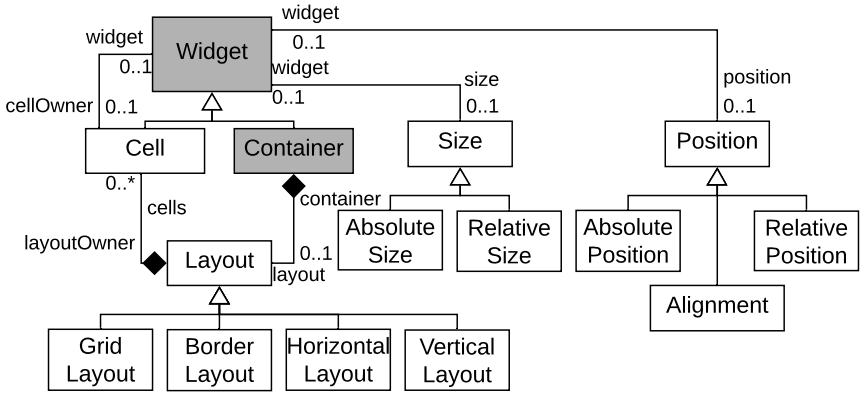



Fig. 3: Layout meta-model (grey concepts are shared with the core meta-model)

Figure 3 represents our layout meta-model. The entities `Widget` and `Container` are part of the core meta-model presented in Section 4.1. The layout meta-model adds four main entities to the core meta-model.

`Size` describes the height and width of a widget. The size of a widget can be absolute or relative. The `AbsoluteSize` is expressed in pixels. The `RelativeSize` is expressed as a percentage of its container.

`Position` describes the position of a widget in the user interface. It can be absolute, relative, or defined by alignment properties. The `AbsolutePosition` represents the coordinates of a widget in the user interface. The `RelativePosition` represents the coordinates of a widget in its container. The `Alignment` defines how to position a widget inside its container. It can be in the *top*, *bottom*, *right*, *left* or *center* of its container or a combination of them.

`Layout` represents rules to position the children of one container. Any `Container` of the core meta-model can have one layout. A `Layout` can be refined as a `Grid Layout`; a `Border Layout`; a `Horizontal Layout` and a `Vertical Layout`. We currently support these layouts because they are most frequently used in our context. However, many other hierarchical layout managers exist (Lutteroth et al., 2008; Sánchez Ramón et al., 2016; Zeidler et al., 2012) and one can extend the layout meta-model to support them.

`Cell` A `Layout` can contain multiple `Cell`s. Then, each `Cell` contains one widget. Thus, the layout dispose the widgets using the `Cell`. It allows one to have fine control of the final GUI layout.

A hardcoded layout can be modeled as is with the `AbsolutePosition` entity. But it is best to migrate it to a hierarchical layout as proposed by Sánchez Ramón et al. (2016).

Note that some `Containers` do not have a `Layout`. For instance, a `<select>` in HTML has multiple `<option>`, thus, it is considered as a `Container` but does not have `Layout`.

### 4.3 Widget meta-model

The core meta-model presented in Section 4.1 allows one to represent the GUI structure. However, to migrate a GUI we map widgets of the source framework to widgets of the pivot meta-model. This mapping is made through the widget meta-model.

The widget meta-model describes the most common user interface widgets. It currently contains all the entities described in the W3School website[10] such as `Button`, `Label`, and `OrderedList`. However, the website only presents the widgets of the HTML standard. Our widget meta-model is composed of 61 widgets and 31 attributes. An excerpt of the widget meta-model is presented in Figure 4.

This meta-model can be extended with other widgets to fit the needs of a specific migration. It was the case for the GUI extraction of Berger-Levrault applications where developers have developed a few specific widgets. In fact, the widget meta-model includes the already known widgets, whereas the `Custom widget` represents the unknown widgets.

### 4.4 Custom widget and specialization

One of the major problem when considering multi GUI framework migration is the ability to handle widgets that might not be present in the GUI pivot model. Indeed, the mapping of widgets between frameworks is not one to one (Gerdes Jr, 2009; Sánchez Ramón et al., 2014; Sánchez Ramón et al., 2016;

[10] https://www.w3schools.com/html/default.asp

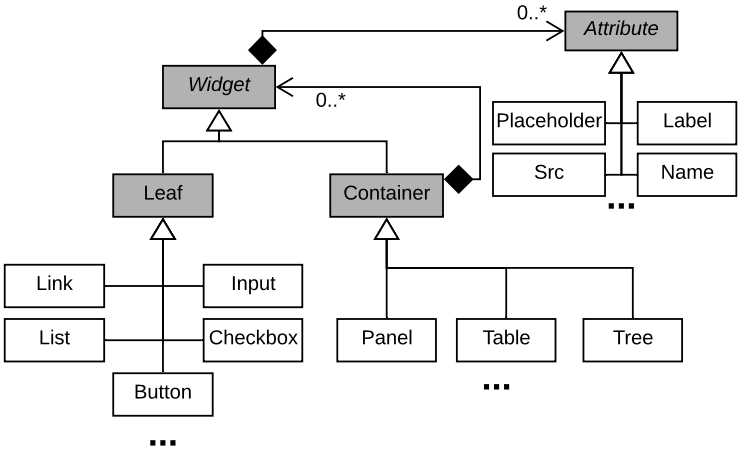



Fig. 4: Excerpt of the widget meta-model (grey concepts are shared with the core meta-model)

Shah and Tilevich, 2011) and third-party libraries may offer new widgets. For example, AWT is an old GUI framework and some of its widgets do not have a counterpart in Angular. Note that in the case of migration between standard web-application (such as applied by (Hayakawa et al., 2012)), the problem is less important because most of the widgets are also standard (*i.e.* `div`, `span`, `input`, *etc.*).

To tackle such a problem, we use the concept of `Custom Widget` (Sánchez Ramón et al., 2014; Sánchez Ramón et al., 2016). During the extraction step, when an unknown widget type is encountered, the extractor creates a `Custom Widget`. Then, it tries to extract the DOM of the `Custom Widget` as a simple container. During the generation, `Custom Widget`s are generated as containers with a comment in the generated code to warn developers and give them additional information: name of the source widget, attributes (if identified), possible children widgets (if identified), location in the source code. With this information, one can add a new widget to the pivot meta-model and update the mapping of the known widgets (mapping source to pivot).

Indeed, we designed our approach and meta-model to be specializable. Thus, better than using `Custom Widget`, one can create specific widgets and use them with our approach. It is the choice we made for the framework of Berger-Levrault (see Section 6.1). It also enables our approach to be iterative: one performs the migration, our tool identifies `Custom Widget`s, then the developers extend our meta-model, and iterates. Note that, for better migration results such a widget should also be created (programmed) in the target framework. If the developers felt the need to create them in the source framework, there is a good chance that the same need applies to the target framework.

## 5 Implementation

To validate our approach we implemented extractors and generators for GUI frameworks that use programming and markup languages. In Section 5.1, we detail the implementation of the steps to extract the GUI part of an application. In Section 5.2, we detail the implementations to generate the target GUI.

### 5.1 Extractors

We implemented three extractors for three different frameworks: The first, BLCore, Java-based GWT, is web-based and the interface is defined in the programming language Java; the second, XML-based GXT[11] is web-based and the interface is defined in a markup language (XML); and the third, Pharo-based Spec (Fabry and Ducasse, 2017), is desktop-based and the interface is defined in another programming language (Pharo). In this section, we discuss separately the four sub-steps to support a new framework (see Section 3.2.1) and the four sub-steps to support a new application (see Section 3.2.2) with concrete examples from the extractors.

#### *5.1.1 Handling a new Framework*

As presented in Section 3.2.1, to support a new framework one needs to:

- map the source framework to the pivot model,
- identify the containment rules,
- identify the custom widgets rules,
- and identify the root widgets rules.

```
<gxt:Window ui:field="window">
  <container:VerticalLayoutContainer>
    <form:FieldLabel text="{i18n.user}">
      <form:widget>
        <form:TextField ui:field="login"/>
      </form:widget>
    </form:FieldLabel>
  </container:VerticalLayoutContainer>
  <gxt:button>
    <button:TextButton text="{i18n.login}"/>
  </gxt:button>
</gxt:Window>
```

Fig. 5: Snippet of an GXT login view in XML

[11] GXT, https://www.sencha.com/products/gxt/, is an extension of GWT

**Map source framework to pivot model.** The basic approach to define a mapping between source and pivot meta-model is simply to create a dictionary. For GXT extractor, we map an XML tag or attribute (source meta-model) to its corresponding pivot widget, attribute, or layout. For example line 2 of Figure 5, the container:VerticalLayoutContainer source tag maps to the pivot widget Panel containing a Vertical pivot layout. Note that, line 5, the ui:field source attribute does not map to anything in the pivot meta-model by itself but is used to map the textField with its counterpart in the Java source code where the behavior of the GUI is described.

For the frameworks based on programming languages (GWT and Spec), we map a source class or method to a pivot widget, attribute, or layout. We need to consider also methods because some widgets may be constructed through factories. Also, an attribute of a source widget might be assigned with a setter method, in this case, the source method (setter) maps to a pivot attribute. For the GUI extraction, we do not consider getters because they do not tell us anything on the value to assign to the attributes and therefore they do not point to anything that we could generate to get the same rendering.

There are other possible mappings, for example a GWT source widget DynamicFieldSetPanel maps to a pivot widget FieldSet and its boolean attribute dynamicFieldSet. The presence of an attribute may also be conditioned to the instantiation of a widget. For example, instantiating a Button widget with a string parameter (new Button("OK")), will set its text attribute. Another case is that of a source attribute mapping to two pivot attributes. For example the width source attribute maps to either the absoluteWidth or relativeWidth pivot attributes depending on whether the value assigned to it is in pixel (*e.g.* "250px") or in percentage (*e.g.* "50%"). This requires symbolic execution and cannot always be achieved (see Section 5.1.2).

**Identify containment link.** This might be very simple. For example in GXT, we just use the DOM of the XML file describing the interface since it is already structured as a containment tree. For frameworks based on programming languages, the links between a widget and its attributes are easily set when the attributes are identified. The containment links between widgets and subwidgets, or widgets and layouts, are identified through the use of a small set of specific methods: ownerWidget.add(<*widget*>) for Spec and GWT, and also ownerWidget.setWidget(..., ..., <*widget*>) for GWT. For these methods, we specify that the parent widget is the receiver, and the child widget the argument.

**Identify custom widget.** In programming language based frameworks, we typically look for new classes inheriting from the most abstract widget[12] in the source meta-model: Widget class for GWT, and ComposablePresenter for Spec.

In GXT, custom widgets are either unknown tags in a GUI description file (named xxx.ui.xml), or a GUI description file not listed as an application root page in the configuration file. For example, in the snippet of Figure 5, we expect to know all tags within <gxt:Window>. Other tags may appear outside

[12] We remind the reader that we consider there can be no custom attribute or layout, see Section 3.2

it, but they are used for configuration and do not impact the visual aspect of the GUI.

**Identify root widget.** In GWT and GXT, we browse the model of the XML configuration file describing the application where all root pages are listed. Note that this is the same file in both cases as GXT is just an extension of GWT.

In Spec, the notion of root widget is fuzzier, it relies on the idea that any widget, however simple or complex, can be opened as a window or included in another widget. Therefore, in this case, we have to treat each root widget of the application separately which means we rely on the user to tell us what a root widget is.

### *5.1.2 Handling a new Application*

As presented in Section 3.2, to support a new application in a known framework one needs to:

- actually identify the custom widget types,
- create all widget instances,
- detect the composition of widgets, and
- perform an optional additional sub-step for layout.

**Identify custom widget types.** We apply the identification rule defined for the given framework. For example, as described above, in GWT we look for all new class descendants of Widget.

**Create widget instances.** In GXT identifying source widgets instances, attributes and layout is achieved by browsing the model of the XML file from top to bottom, creating pivot widgets, attributes, or layouts as we encounter tags (these tags were identified in the Mapping Definition sub-step in the previous section). The composition of the widgets and/or their attributes is easily extracted from the DOM of the source XML file. The same goes for layouts.

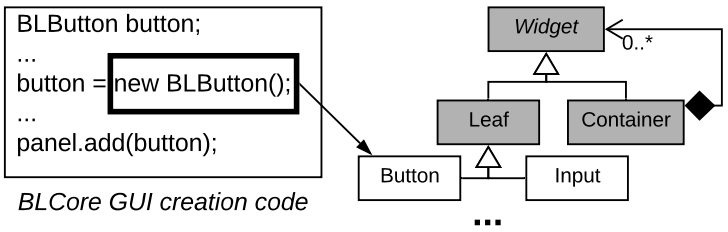



Fig. 6: Example of Widgets Identification

Identifying widgets, attributes, layout instances for the two frameworks based on a programming language is more complex.

If a source widget is identified by a class, we just look for instantiations (new) of this class (see Figure 6). If a source widget is identified by a method (*e.g.* in a Factory), we look for calls to this method. In both cases, we need to retain the instance created for later analysis. Typically the instance is assigned to a variable, and we retain this variable. For example in Figure 6 the Button instance is assigned to the variable button. Since we are working on a model of the source code and not the source code as a textual artifact, this means we have the variable as an entity in the source model and we can easily find every place where it is accessed. In Figure 7, there are two widget instantiations: LinkLabel (which is a known widget in this case), line 4; and Label (another known widget), line 6. The first instance is assigned to the lblPg variable, line 4.

```
1  class SPMetier1 extends AbstractSimplePageMetier {
2    @Override
3    public void buildPageUi(Object object) {
4      LinkLabel lblPg = new LinkLabel("Next");
5      lblPg.setEnabled(methodCall());
6      content.add(new Label("<Business content>"));
7      content.add(lblPg);
8      super.setBuild(true);
9    }
10 }
```

Fig. 7: User interface creation in Java GWT

When looking for attributes in the source model, we search for known setter messages sent to variables containing widget instances. Again these setter messages were identified for the framework as indicators of attributes. In Figure 7, on line 5, the lblPg variable receives the setEnabled message that maps to the disable pivot attribute. Note that, again, the value of setEnabled must be interpreted to give the correct value to the pivot attribute. In the example, we have to execute the method methodCall to resolve the boolean value. This is one of the most difficult and least reliable computations we perform as will be seen in the results of experiments given in Section 6.3. But we still achieved a worst-case of 67% of attributes correctly detected.

**Detect widget composition.** For the DOM building, there are two examples of calls to the add(*<widget>*) in Figure 7, on line 6 and 7. In each case, we already identified the variable to which these messages are sent (content) and the children widgets that are passed as argument.

Note that the line 8 of Figure 7 is not used as it does not involve any known variable or method. It actually does not impact the interface built.

In Spec, the process is similar except that widget creation and containment links are defined in different places. Still, they also rely on the use of variables containing the widgets created and used later to establish the containment links. The work may be a little easier as the creation of the containment

links is separated in a well-defined method which makes it a bit similar to a declarative specification.

Note that children widgets are rarely added directly in their parent widget. Because our DOM also contains the layout, widgets are added to `cell`s that are put in `layout`s, themselves children of the parent `widget`.

**Perfom additional layout computation.** To get an accurate representation of the layout, one needs to compute the position of each cell inside its parent layout. The basic case is to be able to recover the order in which widgets are added into their parent layout. As an example, in a VerticalFlowLayout or HorizontalFlowLayout, the order in which widgets are added controls their position one relative to the other, see for example Figure 7 (lines 6 and 7) where two widgets are added consecutively to their parent.

Spec having a limited number of simple layouts (VerticalFlowLayout and HorizontalFlowLayout), it falls within this easy case.

```
1 int row = 0;
2 Grid grid = new Grid();
3 grid.setWidget(0, 0, new Label("name:"));
4 grid.setWidget(row++, 1, new Button());
5 grid.setWidget(++row, 1, new Label(""));
6 grid.setWidget(grid.getRowCount(), 0, new Label());
7 grid.getFlexCellFormatter().setWidth(0, 1, "50%");
```

Fig. 8: Complex layout creation in Java GWT

More complex layouts, like the Grid layout, allow cells to occupy (span) several positions in the grid or to be inserted in any position of the grid. For GUI based on markup languages (*e.g.* GXT, HTML), position computation is still relatively easy as the information is hardcoded in the source. For GUI based on programming languages (*e.g.* Java GWT), the position or span might be the result of computations at execution time and therefore more difficult to extract. For example see lines 3 to 6 in Figure 8.

We try to solve some of these cases by resorting to symbolic computation. The same ideas were used in other GUI extraction tools (Silva et al., 2010). Symbolic execution involves identifying the semantics of all functions/operators that can be used in the source code. Here, in the example of Figure 8, one needs to know the Pre/Post Increment/Decrement operators (line 4 and 5) as well as the getRowCount, getColumnCount, getCellCount methods, and the assignment operation (line 1). Concretely for our examples, we only need to implement the pre and post-increment operators (x++ ; ++x), and the getRowCount and getCellCount methods. Using symbolic computation, we can resolve the successive values of "row" in Figure 8.

5.2 Generators

Once we have a GUI model, it is possible to generate the GUI in the target framework. As for the extractors, we implemented three generators for three different frameworks: The first, Angular, is web-based and the interface is defined in a markup language (HTML), the second, Seaside, is web-based and the interface is defined in a programming language (Pharo), and the third, Spec2, is desktop-based and the interface is defined in a programming language (Pharo).

In this section, we discuss the steps to generate the target code with concrete examples from the generators:

**Identify target framework environment.** Before working on the concrete code generation, it is essential to discover the architecture required by the target GUI framework. The simplest solution is to read the documentation of the target framework to understand the good practices when developing an application. We also define how GUI dependencies are imported. In Angular, the GUI code is defined inside an HTML file. However, it is also necessary to create several configuration files (*e.g.* module, CSS, route). All those files are taken under consideration at this sub-step and are necessary to create the target GUI. In Spec2 and Seaside, the GUI definition code has to be written in a specific method.

**Map pivot model to target framework.** Building the dictionary that maps pivot model concepts to target framework is similar to the one used for the extraction. In the case of Angular, we map pivot widgets to Angular tags (*i.e.* a Button corresponds to `<input type="button"/>`). In the case of Spec2, we had to determine the method used to instantiate a widget. For instance, the main widgets (*i.e.* button, label) should be instantiated using a factory pattern whereas the traditional call to a constructor method is favored for less frequent or more complex widgets. Finally, the Seaside framework uses invocations to factory methods to build the widgets. Note that, as for the extraction, a concept in our pivot model can correspond to multiple widgets in the target framework.

**Identify target application environment.** This sub-step must be done for each application. It consists in determining the environment in which the target application will be executed. It configures information such as data access endpoint URL, or for web application, local URL that might be necessary to get images (*e.g.* `<img src='myLocalUrl/..' />`). The generator must use the information to create a runnable application in the target environment. For all our generators, the approach will complete the target GUI code with configuration files (or annotations For programming languages).

**Generate code.** The code generation consists of creating the target GUI. The main approach is to visit the pivot DOM and for each widget create its counterparts in the target application. In the case of Angular, the generation is eased by the fact that both the generated HTML file and the pivot DOM have the same structure. For Spec2 and Seaside, our generator creates into specific methods the widgets and their composition using constructors and a

pre-defined set of methods. For Spec2 specifically, the DOM must be defined in a method and the widget instantiated in another method.

## 6 Result

We perform the migration using our extractors and generators on five real projects. In Section 6.1, we present the case studies. Then, we discuss our validation metrics (Section 6.2) and present the result for the extraction step (Section 6.3) and the generation step (Section 6.4).

### 6.1 Case studies

To validate our approach, we migrated five applications.

Two migration cases, Kitchensink and PostOffice, use BLCore as a source framework and Angular as a target framework. BLCore is the custom GUI framework of Berger-Levrault that extends the GWT GUI framework with specific widgets. This framework consists of 763 classes in 169 packages. It also encourages some coding conventions. Angular is a modern GUI framework supported by Google based on TypeScript.

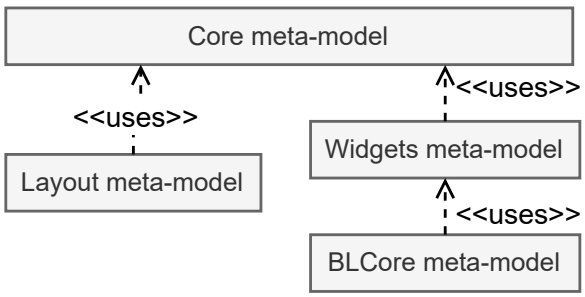



Fig. 9: BLCore - GUI meta-model

The GUI meta-model used for BLCore GUIs extraction, presented in Figure 9, is composed of: the layout meta-model uses core meta-model (Section 4, Figure 2), the widget (Figure 4) extends the core meta-model, and a BLCore meta-model extends the widget meta-model with all components created by Berger-Levrault.

Table 6 summarizes the different migrations, the language and framework source, and the target language and target framework.

Kitchensink is a closed-source application of Berger-Levrault. This software system, targetting developers, gathers inside a single application all the components available for building a user interface. It contains 470 Java classes

Table 6: Case study Description

| Project | Source Framework | GUI definition Type | Target Framework | GUI definition Type |
|---|---|---|---|---|
| Kitchensink | BLCore | programming | Angular | markup |
| PostOffice | BLCore | programming | Angular | markup |
| Traccar | GXT | markup | Seaside | programming |
| DBManager | Spec | programming | Spec2 | programming |
| SpecDB | Spec | programming | Spec2 | programming |

and 56 web pages. PostOffice is a software system used in French administration for the dispatch and digitalization of mails. It contains 3227 classes and 98 pages. Both are written using BLCore, a web-based GUI defined using programming language, and migrated to Angular, a web-based GUI defined using a markup language.

Two migration cases, DBManager and SpecDB, use Spec as a source framework and Spec2 as a target framework. SpecDB is part of the Spec widgets presentation package. It is used to show the different configurations of a Spec button. DBManager[13] provides a GUI to manage the connections between Pharo and databases. Its user interface is divided into multiple pages. Note that even if Spec2 is the next version of Spec, the framework has been completely re-written and so the migration corresponds to a full framework migration and not a GUI framework update.

The last one, Traccar[14], uses the GXT framework (a GWT extension) as a source framework and Seaside as a target framework Traccar is an open-source server and web client for various GPS tracking devices. It contains 649 classes and 28 pages.

Table 7: Application descriptions

| Source framework | Project | Widgets | Attributes | Roots |
|---|---|---|---|---|
| BLCore | Kitchensink | 238 | 156 | 6 *(out of 56)*[1] |
| | PostOffice | 724 | 1065 | 10 *(out of 98)*[1] |
| GXT | Traccar | 125 | 104 | 3 *(out of 28)*[1] |
| Spec | DBManager | 38 | 27 | 3 |
| | SpecDB | 15 | 21 | 1 |

[1] 10% sample

Table 7 summarizes the information about the different projects. For the Kitchensink, PostOffice, and Traccar projects, we take a sample of the root

[13] https://github.com/juliendelplanque/DBConnectionsManager

[14] https://github.com/traccar/traccar-web

pages for the validation. So we present the number of widgets, attributes, and roots of the sample. This choice is explained in the next section (Section 6.2).

### 6.2 Validation set-up

We divided the migration validation into two parts: the extraction validation and generation validation. The extraction validation consists of checking that our model includes the elements of the original application. It compares the DOM and the attributes of the GUI without considering the final visual aspect. The generation validation consists of visually comparing each original page with the exported one.

For the **extraction validation**, our solution validates that the GUI hierarchy is identified. For the GUI hierarchy, we check that all the widgets and attributes are detected and correctly identified. This validation consists of the following three metrics (Hayakawa et al., 2012; Joorabchi and Mesbah, 2012; Sánchez Ramón et al., 2014):

- The percentage of widgets correctly *detected* and created in our pivot model. It checks that the number of widgets in the source application is the same as in our pivot model.
- The percentage of widgets correctly *identified*, *i.e.* a button in the original framework corresponds to a button in the pivot model. For instance, a widget not identified is mapped to a `Custom Widget` or another widget (*e.g.* a button mapped to a panel).
- The percentage of widgets *assigned* to the correct container. It validates the DOM building.

Finally, the same metrics are used for the attributes.

We rely on manual validation to check all these metrics. Because the manual validation is tedious and error-prone for large applications, we take a sample of the pages of the Kitchensink, PostOffice, and Traccar applications. For each case, we consider a sample representing at least 10% of the application. So we randomly selected 6 pages out of 56 for Kitchensink, 10 pages out of 98 for PostOffice and 3 pages out of 28 for Traccar.

Table 8: Extraction from previous work

| Framework source | Project | Widget detected | Widget well assigned | Attribute detected | Attribute well assigned |
|---|---|---|---|---|---|
| BLCore | Kitchensink | 100% (238) | 89% (211) | NA[1] | NA[1] |
| Spec | DBManager | 100% (38) | 100% (38) | 92% (25) | 100% (25) |
| | SpecDB | 100% (15) | 100% (15) | 67% (14) | 100% (14) |
| average | | 100% | 96% | 80% | 100% |

[1] Value not given in previous work

The migration of the Kitchensink, DBManager, and SpecDB projects were also presented in previous work (Dutriez et al., 2019; Verhaeghe et al., 2019). However, those experiments did not consider the migration of the layout. Thus, their result might differ from the ones reported here. The results of the previous experiments are reported in Table 8 and discussed in the next section.

For the **generation validation**, work has been proposed to compare images (Cao et al., 2010; Moran et al., 2018). However, none is directly applicable to our migration cases. Indeed, to apply this strategy one must before deal with several challenges (Bragagnolo et al., 2020b):

- *Ajax-based architecture*: one should be able to browse the pages of an Ajax application for web application.
- *successive shifting*: we must deal with successive shifting of widget components when a target one is not perfectly visually equivalent to an original one.
- *dynamic content support*: for instance, some widgets, as a table, display information coming from an external server. The same data must be presented in the original and generated application.

Validation by image comparison is further discussed in Section 7.4.

We rely on a manual visual comparison of the pages. First, we check that the generated application is runnable. Then, we visually compare the application with the original one.

One could also think of using tests for the validation. However, this is not applicable, in our cases, because the applications we are migrating do not have such tests.

### 6.3 Extraction result

Table 9: Extraction results

| Framework source | Project | Widget detected | Widget identified | Widget well assigned | Attribute detected | Attribute identified | Attribute well assigned |
|---|---|---|---|---|---|---|---|
| BLCore | Kitchensink | 100% (238) | 94% (224) | 99% (236) | 77% (118) | 95% (112) | 100% (118) |
| | PostOffice | 99% (718) | 99% (712) | 96% (695) | 88% (940) | 98% (923) | 99% (937) |
| GXT | Traccar | 100% (125) | 99% (124) | 100% (125) | 81% (84) | 100% (84) | 100% (84) |
| Spec | DBManager | 100% (38) | 94% (36) | 100% (38) | 92% (25) | 100% (25) | 100% (25) |
| | SpecDB | 100% (15) | 100% (15) | 100% (15) | 67% (14) | 100% (14) | 100% (14) |
| Average | | 99% | 98% | 99% | 87% | 98% | 100% |

Table 9 summarizes the extraction results.

The tool detects 99% of all the widget instantiations for all the applications. It shows we have good heuristics to identify the widgets in the source applications.

The tool identifies correctly 97% of the widgets type. For the Traccar and DBManager, the tool misses widgets used in toolbars. This kind of widget is mainly used in desktop application and since our widgets meta-model comes from W3School which describes basic web components, we did not have them. For Kitchensink and PostOffice, the tool identifies 94% and 99% of the widgets. All the unidentified are widgets created by the company for its business. So they are mapped to `Custom Widget`. To solve this problem, one can extend our meta-model with the missing widgets. We also report that our implementation benefited from the detailed approach steps since it better extracts the GUI than the previous approaches. For instance, our new approach correctly assigned 99% of the widgets whereas the previous approaches dealt with only 96% of the widgets.

Except for the BLCore framework, all the widgets are well assigned to their container. With BLCore, the problem comes from the variety of ways to define widget containment.

Attributes are correctly detected at 87%. The best result appears in the DBManager application with 92%. Our approach reports better result than the 80% of attributes detected in previous experiments.

Attributes are harder to detect for two reasons: (1) GUI frameworks define default attributes for widgets, so we have to manually analyze those attributes and add them in our extractors, and (2) in programming languages, attributes can be declared in multiple ways: using a setter; a parameter in a constructor. This diversity forces us to analyze all the possibilities since we did not find a heuristic that will select all attributes definitions.

Finally, nearly all the detected attributes are well assigned to their container.

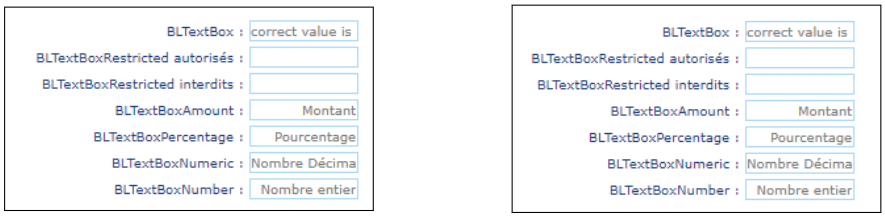
### 6.4 Generation result

We visually compared the visual aspect of the pages where the widgets are well identified. In the following, we present a comparison for three of the case studies presented in Section 6.1. Note that other comparisons for the Kitchensink migration are available in the documentation page of our project[15].

Figure 10 presents the visual differences for the `page` *Input box* of the Kitchensink application. On the left-hand side there is the original `page` and on the right-hand side the `page` after the migration. We can see that there are no differences between the two versions.

Figure 11 presents the visual differences for the User Setting page of Traccar. On the left-hand side, there is the original page, and on the right-hand side, the page after the migration to Seaside. There are more differences in

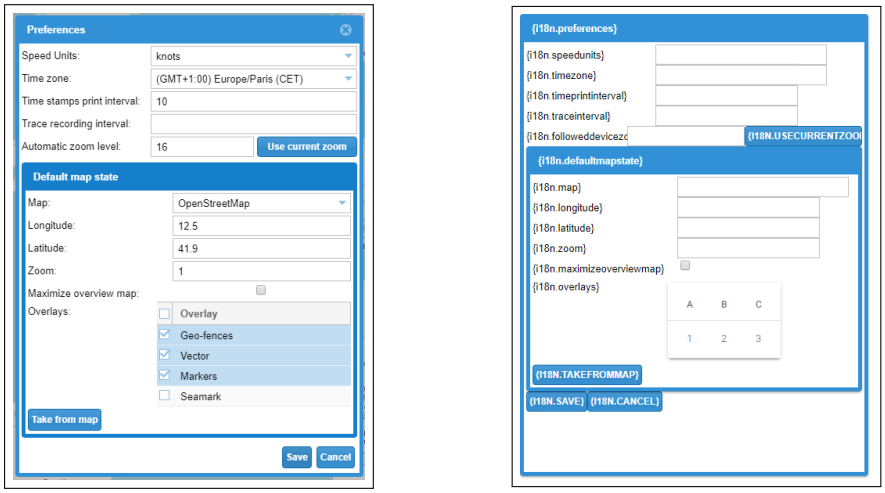
[15] https://badetitou.github.io/projects/Casino/Casino/#current-results




(a) GWT original

(b) Angular migration

Fig. 10: Visual comparison of a `Page` migration (Kitchensink)




(a) User Setting original

(b) User Setting migrated

Fig. 11: Visual comparison of the User Setting page (Traccar)

this example. The sizes of the input boxes are different, the text of the labels is replaced by the i18n notation, the "overlay" table has a complete different visual aspect, the checkbox is not centered, and the buttons at the bottom are not well placed. The table is not correctly migrated because the table with selection is a custom widget for the tool and we did not take the time to introduce it. It was then migrated from component it contains, here a simple table. The other differences come from application-specific CSS. Despite all these imprecisions, the structure and the layout of the page are respected.

Figure 12 presents the visual differences for the DBManager application. Again on the left-hand side, there is the original page, and on the right-hand side, the page after the migration. There are some differences in this example. The text of the buttons on the left of the image is not present, the buttons on the right of the image are enabled but are disabled in the original application and are not correctly placed. Apart from the last difference that comes from a developer hack, all the differences are due to attribute extraction problems.


(a) DBManager original

(b) DBManager migrated

Fig. 12: Visual comparison (DBManager)

Those problems come from missing attributes identification rules in our tool. Finally, the drop-box input has visual differences between the original and the generated application. It is due to different widget implementations.

The migration does not create the same visual aspect. We saw that there are few differences and the layouts are respected. Most of the imprecisions come from missing attributes or incorrect values. Those problems are easy to fix manually by developers.

We do not report major differences in the visual aspect of pages of the validation. Part of the success of the migration is due to the manual work we performed to map the GUI frameworks to our meta-models. We detail this manual effort requirement in Section 7.

## 7 Discussion

In this section, we discuss our approach and the validation set-up. In Section 7.1 we discuss the similar visual aspect constraint and animation migration. In Section 7.2, we present the case of none file-based GUI migration. In Section 7.3, we discuss the number of selected pages and how the selection has been made for the validation. In Section 7.4, we detail the difficulties encountered to perform an image comparison validation. In Section 7.5, we discuss the manual work required to implement our approach.

### 7.1 Similar visual aspect and Animations

Our approach allows one to migrate the visual aspect of its GUI among different GUI frameworks. As validation, we proposed to compare the visual aspect of the former GUI to the generated one. However, widgets do not have the same visual aspect in different GUI frameworks. For instance, an AWT button does not look like an AngularJS button.

It is possible to perform an expensive manual step to tune the visual aspect of all target framework widgets to mimic the former visual. Because of its cost, performing this step is not always desirable.

One the one hand, as depicted by Moore et al. (1994), “The resulting user interface should have the true look and feel of the new environment”. On the

other hand, for commercial software, keeping the same visual aspect increases the acceptance rate of client users (Sánchez Ramón et al., 2014).

Note that the same limitation exists for GUI animations. Animations are represented as attributes of the widgets. These GUI animations attributes exist in several GUI frameworks and it is possible to reproduce them in others. However, this task is expensive and one should rely on its migration context to determine if animation migration is mandatory. We did not work on animations migration because our source applications did not use animations. We believe that extra research work should be done on attributes to enable a good animations representation and migration.

### 7.2 Migrating none-file or binary file-based GUI

In this paper, we deal with the migration of GUI defined in programming languages or markup languages. However, some GUIs are defined using other formats. It is the case of GUIs that are defined in binary files.

For instance, GUI defined with RAD requires another approach. Bragagnolo et al. (2020a) have studied the extraction of the GUI for Access projects. In their case study, they had to perform the extraction of the GUI through the Access IDE which limited the extraction capabilities.

Thus, to migrate binary file-based GUI, the extraction part of the migration is different. The mapping to the GUI meta-model is similar to our approach once one has gained access to the source GUI representation.

Migration of binary file-based GUI is part of our industrial partner projects with a WebDeb to Angular migration project.

### 7.3 Sample selection

As depicted in the previous sections, we had to perform a manual validation. However, performing a manual validation on the 186 pages is too time-consuming.

Thus, we decided to perform the validation of a subset. To do so, we decided to selected 10% of the pages of the applications. This selection is crucial because it must not introduce bias in the validation. So, to avoid introducing bias, we decided to perform a random selection of the pages. However, it could still not fully represent the application.

So, to ensure the good representativity of the selected pages, we compare the total number of widgets and attributes in the application to the number of widgets and attributes of the selected pages. The selected pages include 9% of the total number of widgets and 13% of the total number of attributes. Thus, it appears that the 10% pages randomly selected are representative of the total application in terms of the number of elements.

Additionally to comparing the number of elements, we check that selected pages are of different sizes. As result, the pages have from one to hundreds of

widgets. Thus, the randomly selected page are of different sizes, and this helps ensure no bias in the validation.

### 7.4 Image comparison validation

To validate the proper generation of target GUI, we rely on a manual comparison of the pages visual aspect. We acknowledge the existence of projects that compare the visual aspect of two screenshots (Cao et al., 2010; Moran et al., 2018). Cao et al. (2010) classified pages by comparing their GUI, *e.g.* determined if a page come from a newspaper website or another one. Moran et al. (2018) detected differences between two versions of the same GUI after modification made by developers. Since the modifications are minimal and the GUI developed with the same framework, one can expect few differences between screenshots.

We performed a preliminary work (Bragagnolo et al., 2020b) to adapt image comparison to the validation of GUI migration. In this work, we reported several challenges that must be first solved.

First, for Ajax-based applications such as GWT and Angular, one should be able to browse the pages of the web application. Indeed, to automatically take screenshots of the pages, one must crawl all pages. However, pages are not directly accessible with their URL in some Ajax-based applications such as the applications using GWT in our context.

Second, we must deal with the successive shifting challenge. When migrating a widget from a source framework to a target framework, it might not have the same visual aspect. However, if a target widget has a size different than the original widget, it introduces a shift in the screenshot of a few pixels. Repeating this shift on several widgets in a page can lead to two completely different source and target pages, and so prevent the screenshot comparison.

Finally, to enable the image comparison in the migration context, one must before deal with dynamic content. Indeed, a table filled with data will not take the same space in a GUI as an empty table. However, in our context, we do not support the migration of backend connections. Thus, migrated tables are always empty and visual comparison can not be used.

Thus, automatic validation through visual comparison of page screenshots is not applicable in the current state of GUI visual aspect among multiple technologies.

### 7.5 Manual work

The migration of BLCore, GXT, and Spec requires two manual tasks (1) to map their widgets with the ones of our meta-model and (2) to identify how the DOM is built in each GUI framework.

Mapping the widget consists of analyzing the documentation of each GUI framework, retrieving the widgets and their attributes, and mapping them

to our meta-model. Whereas the second task is easy for markup languages, it requires more knowledge for programming languages analysis. Indeed, in programming languages based GUI, developers can define the DOM in several ways. So, we had to enumerate all the DOM building possibilities and integrate them into our extractors.

To reduce the required manual effort, one can perform a renovation of its software system before migrating. The renovation consists of improving the source code of the application. To do so, developers reduce the number of code smells and rewrite code to make it easier to manipulate. For example, one can decide to rewrite all widgets creation using the basic new method. By following this *Quality First*(Włodarski et al., 2019) rule, the extraction of the GUI is simpler, and the DOM is built using only one approach. Widgets are always instantiating using the constructor. And the number of custom widgets might be reduced in favor of standard ones.

## 8 Conclusion and Future work

In this paper, we exposed an approach and tool to migrate automatically the visual aspect of applications GUI. We detail the extraction and the generation steps of this approach for programming and markup GUI framework. We validated our approach by implementing it in a tool called Casino to perform the migration of Spec, BLCore, and GXT GUI frameworks to Angular, Seaside, and Spec2. Then we performed five real migrations on open-source and closed-source applications. We were able to extract correctly all pages of the applications and 97% of the widgets. And the migration results are visually equivalent.

Currently, only the visual part of the GUI is migrated. To improve the migration of an application user interface, we will enhance our meta-models and our tool to support the behavioral and the business parts.

We have validated our approach using manual validation techniques. While this gave us a good approximation of the results, using an automatic solution to validate the migration may provide us more information and help the developers during the process.

We are currently working with our industrial partner on making available extractors for PowerBuilder and Visual Basic 6 (which are not Object-Oriented programming language). We also plan to test our approach with mobile GUI with a subsidiary of our industrial partner. Thus, testing our approach on Android applications is part of our future work.

## 9 Declarations

**Funding** This work was supported by Berger-Levrault and Inria-Lille–Nord-Europe.

**Conflicts of interest/Competing interests** The authors have no relevant financial or non-financial interests to disclose.

**Availability of data and material** The source code of the Traccar case study is available at: https://github.com/traccar/traccar-web. The source code of the DBManager case study is available at: https://github.com/juliendelplanque/DBConnectionsManager. The source code of the SpecDB case study is available at: https://github.com/pharo-spec/Spec. The source code of Kitchensink as the source code of PostOffice are closed-source.

**Code availability** The source code of the Casino tool is available at: https://github.com/badetitou/Casino. Additional extractors and generators are provided at: https://badetitou.github.io/projects/Casino/#links.

## References


Beltramelli T (2017) pix2code: Generating code from a graphical user interface screenshot. arXiv preprint arXiv:170507962

Bragagnolo S, Anquetil N, Ducasse S, Abderrahmane S, Derras M (2020a) Analysing microsoft access projects: Building a model in a partially observable domain. In: International Conference on Software and Systems Reuse, ICSR2020

Bragagnolo S, Verhaeghe B, Seriai A, Derras M, Etien A (2020b) Challenges for layout validation: Lessons learned. In: International Conference on the Quality of Information and Communications Technology, QUATIC'2020, accepted

Brambilla M, Fraternali P (2014) Interaction flow modeling language: Model-driven UI engineering of web and mobile apps with IFML. Morgan Kaufmann

Cao J, Mao B, Luo J (2010) A segmentation method for web page analysis using shrinking and dividing. International Journal of Parallel, Emergent and Distributed Systems 25(2):93–104

Chen C, Su T, Meng G, Xing Z, Liu Y (2018) From ui design image to gui skeleton: A neural machine translator to bootstrap mobile gui implementation. In: Proceedings of the 40th International Conference on Software Engineering, Association for Computing Machinery, New York, NY, USA, ICSE '18, p 665–676, DOI 10.1145/3180155.3180240

Dutriez C, Verhaeghe B, Derras M (2019) Switching of GUI framework: the case from Spec to Spec 2. In: Proceedings of the 14th Edition of the International Workshop on Smalltalk Technologies, Cologne, Germany

Fabry J, Ducasse S (2017) The Spec UI Framework. Square Bracket Associates

Fleurey F, Breton E, Baudry B, Nicolas A, Jezéquel JM (2007) Model-Driven Engineering for Software Migration in a Large Industrial Context. In: Engels G, Opdyke B, Schmidt DC, Weil F (eds) Model Driven Engineering Languages and Systems, Springer Berlin Heidelberg, Berlin, Heidelberg, vol 4735, pp 482–497, DOI 10.1007/978-3-540-75209-7_33

Garcés K, Casallas R, Álvarez C, Sandoval E, Salamanca A, Viera F, Melo F, Soto JM (2017) White-box modernization of legacy applications: The

oracle forms case study. Computer Standards & Interfaces pp 110–122, DOI https://doi.org/10.1016/j.csi.2017.10.004

Gerdes Jr J (2009) User interface migration of microsoft windows applications. Journal of Software Maintenance and Evolution: Research and Practice 21(3):171–187

Gotti Z, Mbarki S (2016) Java swing modernization approach - complete abstract representation based on static and dynamic analysis:. In: Proceedings of the 11th International Joint Conference on Software Technologies, SCITEPRESS - Science and Technology Publications, pp 210–219, DOI 10.5220/0005986002100219

Hasselknippe KF, Li J (2017) A novel tool for automatic gui layout testing. In: 2017 24th Asia-Pacific Software Engineering Conference (APSEC), pp 695–700, DOI 10.1109/APSEC.2017.87

Hayakawa T, Hasegawa S, Yoshika S, Hikita T (2012) Maintaining web applications by translating among different RIA technologies. GSTF Journal on Computing p 7

Joorabchi ME, Mesbah A (2012) Reverse engineering iOS mobile applications. In: 2012 19th Working Conference on Reverse Engineering, IEEE, pp 177–186, DOI 10.1109/WCRE.2012.27

Kazman R, Woods S, Carriére S (1998) Requirements for integrating software architecture and reengineering models: Corum ii. In: Proceedings of WCRE '98, IEEE Computer Society, pp 154–163, iSBN: 0-8186-89-67-6

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

Memon A, Banerjee I, Nagarajan A (2003) GUI ripping: reverse engineering of graphical user interfaces for testing. In: Reverse Engineering, 2003. WCRE 2003. Proceedings. 10th Working Conference on, IEEE, pp 260–269, DOI 10.1109/WCRE.2003.1287256

Mesbah A, van Deursen A (2007) Migrating multi-page web applications to single-page ajax interfaces. In: Proceedings of the 11th European Conference on Software Maintenance and Reengineering, IEEE Computer Society, Washington, DC, USA, CSMR '07, pp 181–190, DOI 10.1109/CSMR.2007.33

Mesbah A, van Deursen A, Lenselink S (2012) Crawling ajax-based web applications through dynamic analysis of user interface state changes. ACM Transactions on the Web 6(1):1–30, DOI 10.1145/2109205.2109208

Moore, Rugaber, Seaver (1994) Knowledge-based user interface migration. In: Proceedings 1994 International Conference on Software Maintenance, IEEE Comput. Soc. Press, pp 72–79, DOI 10.1109/ICSM.1994.336788

Moran K, Watson C, Hoskins J, Purnell G, Poshyvanyk D (2018) Detecting and Summarizing GUI Changes in Evolving Mobile Apps. arXiv:180709440 [cs] ArXiv: 1807.09440

Rodríguez-Echeverría R, Conejero JM, Clemente PJ, Preciado JC, Sánchez-Figueroa F (2011) Modernization of legacy web applications into rich internet applications. In: International Conference on Web Engineering, Springer, pp 236–250

Samir H, Kamel A, Stroulia E (2007) Swing2script: Migration of Java-Swing applications to Ajax Web applications. In: 14th Working Conference on Reverse Engineering (WCRE 2007)

Sánchez Ramón O, Sánchez Cuadrado J, García Molina J (2014) Model-driven reverse engineering of legacy graphical user interfaces. Automated Software Engineering 21(2):147–186, DOI 10.1007/s10515-013-0130-2

Sánchez Ramón Ó, Sánchez Cuadrado J, García Molina J, Vanderdonckt J (2016) A layout inference algorithm for graphical user interfaces. Information and Software Technology 70:155–175

Shah E, Tilevich E (2011) Reverse-engineering user interfaces to facilitate porting to and across mobile devices and platforms. In: Proceedings of the compilation of the co-located workshops on DSM'11, TMC'11, AGERE! 2011, AOOPES'11, NEAT'11, \& VMIL'11, ACM, pp 255–260

Silva JaC, Silva CC, Goncalo RD, Saraiva Ja, Campos JC (2010) The GUISurfer tool: towards a language independent approach to reverse engineering GUI code. In: Proceedings of the 2Nd ACM SIGCHI Symposium on Engineering Interactive Computing Systems, ACM Press, pp 181–186, DOI 10.1145/1822018.1822045

Verhaeghe B, Etien A, Anquetil N, Seriai A, Deruelle L, Ducasse S, Derras M (2019) GUI migration using MDE from GWT to Angular 6: An industrial case. In: 2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER), Hangzhou, China

Włodarski L, Pereira B, Povazan I, Fabry J, Zaytsev V (2019) Qualify first! a large scale modernisation report. In: 2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER), IEEE, pp 569–573

Zeidler C, Müller J, Lutteroth C, Weber G (2012) Comparing the usability of grid-bag and constraint-based layouts. In: Proceedings of the 24th Australian Computer-Human Interaction Conference, ACM, pp 674–682