Migrating GWT to Angular 6 using MDE

Benoît Verhaeghe, Nicolas Anquetil, Stéphane Ducasse, Abderrahmane Seriai, Laurent Deruelle, Mustapha Derras

Migrating GWT to Angular 6 using MDE

Benoît Verhaeghe[1,2,3], Nicolas Anquetil[1,3], Stéphane Ducasse[3,1], Abderrahmane Seriai[2], Laurent Deruelle[2], Mustapha Derras[2]
[1]Univ. Lille, CNRS, Centrale Lille, Inria, UMR 9189 – CRIStAL, 59650 Villeneuve d'Ascq, France
{firstname.lastname}@univ-lille.fr
[2]Berger-Levrault, France
{firstname.lastname}@berger-levrault.com
[3]RMod team, INRIA Lille Nord Europe, Villeneuve d'Ascq, France
{firstname.lastname}@inria.fr

Abstract

In the context of a collaboration with Berger-Levrault, a major IT company, we are working on the migration of a GWT application to Angular. We focus on the GUI aspect of this migration which, even if both are web frameworks, is made difficult because they use different programming languages (Java for one, Typescript for the other) and different organization schemas (*e.g.* different XML files). Moreover, the new application must mimic closely the visual aspect of the old one so that the users of the application are not disturbed. We propose an approach in three steps that uses a meta-model to represent the GUI at a high abstraction level. We evaluated this approach on an application comprising 470 Java (GWT) classes representing 56 screens. We are able to model all the web pages of the application and 93% of the widgets they contain, and we successfully migrated (*i.e.*, the result is visually equal to the original) 26 out of 39 pages (66%). We give examples of the migrated pages, both successful and not.

1 Introduction

During the evolution of an application, it is sometimes necessary to migrate its implementation to a different programming language and/or Graphical User Interface (GUI) framework [2, 27]. Web GUI frameworks in particular evolve at a fast pace. For example, in 2018 there were two major versions of Angular, three major versions of React.js, four versions of Vue.js, and three versions of Ember.js. This forces companies to update their software systems regularly to avoid being stuck in old technologies.

Our work takes place in collaboration with Berger-Levrault, a major IT company selling Web applications developed in GWT. However, GWT is no longer being updated with only one major release since 2015. As a consequence, Berger-Levrault decided to migrate its GUI to Angular 6.

The company developes 8 applications using GWT. Each application has more than 1.5 MLOC and represents more than 400 web pages. The applications are built from more than 45 kind of widgets and 29 kind of attributes. The company estimated the migration for one application to 4 000 man-days. So, migrating automatically the visual part of an application would already be a useful step for the modernization of the company's applications. Because of fast evolution of GUI framework, the company also needs a reusable solution for the migration to the next programming language.

There are many published papers on supporting GUI migration (*e.g.* [11, 22, 24]). None of them discuss the case of GUI migration of web-based applications.

We present an approach to help developers migrate the GUI of their web-based software systems. This approach includes a GUI meta-model, a strategy to generate the model, and how to create the target GUI. To validate this approach, we developed a tool which migrates Java GWT applications to Angular. Then, we validated our approach on an industrial project that is used to present all the widgets and their usage. It is composed of 470 Java classes and 56 web pages. Our approach imported correctly 93% of the widgets and 100% of the pages. Since not all the existing widgets are re-implemented in Angular, we tried to migrate 39 pages and were successful (same visual appearance) for

In: A. Editor, B. Coeditor (eds.): Proceedings of the XYZ Workshop, Location, Country, DD-MMM-YYYY, published at http://ceur-ws.org

26 of them (66%).

First, in Section 2, we review the literature on GUI meta-modeling. We describe the context of our project in Section 3. In Section 4, we describe our migration approach. We present our implementation in Section 5. Section 6 describes the experiment we made to validate our approach. In Section 7, we present our results. Finally, in Section 8 and Section 9 we discuss the results obtained with our tool and future work.

2 State of the Art

Section 2.1 presents the techniques used to migrate an application. In Section 2.2, we describe the user interface meta-models found in the literature.

2.1 Existing migration strategies

To define a migration strategy, we identified research work related to application migration. Some of the proposed approaches do not perform a full migration, but only a part of it. Also, there are numerous publications dealing with programming language migration. We do not, however, consider them if they do not explicitly discuss the GUI migration. This is the case, for example, with the work of Brant *et al.* [2] that reports on a large Delphi to C# migration.

We identified three techniques to create a representation of the GUI: static, dynamic, or hybrid.

Static. The static strategy consists in analyzing the source code and extracting information from it. It does not execute the code of the analyzed application.

Cloutier *et al.* [3] analyzed directly the HTML, CSS, and JavaScript files. The analysis builds a syntax tree of the source code of the website and extracts the widgets from the HTML files. The work consists mainly in the identification of links between code source elements of the program (JavaScript classes, HTML tags, etc.). The work presented does not tackle the full migration of the GUI.

Lelli *et al.* [13], Silva *et al.* [25] and Staiger [26] used tools that analyze source code of desktop applications. The tools look for widget definition in the source code, then they analyze the methods that invoked or are invoked by the widgets and identify the relationships between widgets and their visual properties.

Sánchez Ramán *et al.* [23] and Garcés *et al.* [8] developed approaches to extract the GUI of Oracle Forms applications. With this framework, developers define the user interfaces in external files where the position of each widget is specified. Their approaches consist in the creation of the hierarchy of widgets from their position. However, the case studies are simple with only few forms or labels. The page layout is also simple because all the elements are displayed below one another.

The static strategy allows one to analyze an application without having to execute it or even compile it. Apart from the classical problem of showing all the potential facts rather than only the real one, another limitation appears for example, with a client/server application, when a part of the graphical interface depends on the result of a request to a server.

Dynamic. The dynamic strategy consists in the analysis of the graphical interfaces of an application while it is running. It explores the application state by performing all the actions on the user interface of the software system and extracting the widgets and their information.

Memon *et al.* [15], Samir *et al.* [22], Shah and Tilevich [24] and Morgado *et al.* [20] developed tools that implement a dynamic strategy. However, the solutions proposed are only available for desktop rather than Web applications.

The dynamic analysis allows one to explore all the windows of an application and to gather detailed information about them. However, automatically running an application to methodically capture all its screens might be a difficult task depending on the technology used. Also, if a request is done to build a GUI, the dynamic analysis does not detect this information which may be essential for a full representation of a GUI.

Hybrid. The hybrid strategy tries to combine the advantages of the static and dynamic analyses.

Gotti and Mbarki [9] used a hybrid strategy to analyze Java applications. First they create a model from a static analysis of the source code. The static analysis finds the widgets and attributes of a user interface and how they are structured. Then, the dynamic analysis executes all the possible actions linked to a widget and analyze if a modification occurs on the interface.

Despite the usage of both static and dynamic analysis, the hybrid strategy does not solve the request problem inherent to client/server applications. It also has the same issues as the dynamic analysis of running automatically an application and capturing its screens.

Fleurey *et al.* [7] and Garcés *et al.* [8] worked on full migration of software systems. They developed a tool that semi-automatically performs the migration. To do so, they used the horseshoe process (Kazman *et al.* [12]). The migration is divided into the following four steps:

1. Generation of the model of the original application.
2. Transformation of this model into a pivot model. This includes data structure, actions and algorithms, user interface, and navigation.
3. Transformation of the pivot model into a target language model.
4. Generation of the target source code.

None of the authors considered the migration from web GUI to web GUI. Also, none had the constraint of keeping similar layout except Sánchez Ramán *et al.* [23]; however, they worked on Oracle Forms applications which are very

different from a web GUI. As a consequence, their work is not directly applicable to our case study.

2.2 User Interface representation

In the previous section, many abstract representations of a GUI are used. We looked to the proposed representations and compared them. We now present the two GUI meta-models defined by the OMG. The Knowledge Discovery Metamodel (KDM) allows one to represent any kind of application. The Interaction Flow Modeling Language (IFML) is specialized in applications with a GUI. Section 2.2.2 presents other representations described in the literature and compare them to the ones of the OMG.

2.2.1 OMG standards

The OMG defines the KDM standard to support the evolution of software. The standard defined a meta-model to represent a piece of software at a high level of abstraction. It includes a UI package which represents the components and behavior of a GUI.

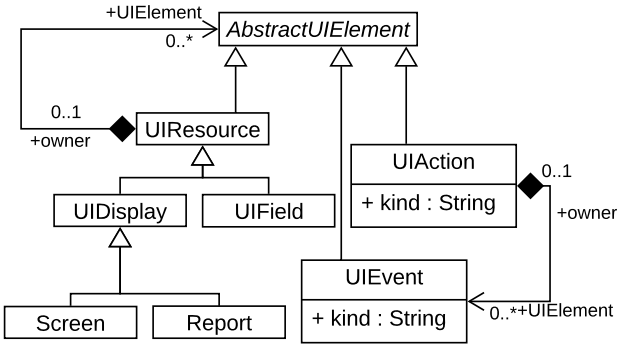


Figure 1: KDM - **UIResources** Class Diagram

Figure 1 represents the core of the UI part called UIResources Class Diagram. The main entity is **UIResource**. It can be refined as **UIDisplay** or **UIField**. **UIDisplay** corresponds to the physical support on which the interface will be displayed, *e.g.* a computer screen, a printed report, *etc.* **UIField** corresponds to a user interface widget, *e.g.* a form, a text field, a panel, *etc.* The composition between **UIResource** and **AbstractUIElement** is used to define the DOM (Document Object Model). Each **UIResource** can contain another one to represent a widget that contains another widget.

A **UIResource** can have, through composition, an UIAction to represent the behavior of the user interface.

The aim of IFML [1] is to provide tools to describe the visible parts of an application, with the components and the containers, the state of the components, the logic of the application and the binding between data and GUI.

Figure 2 represents the meta-model of the visual part of an application. The visible elements of the GUI are called

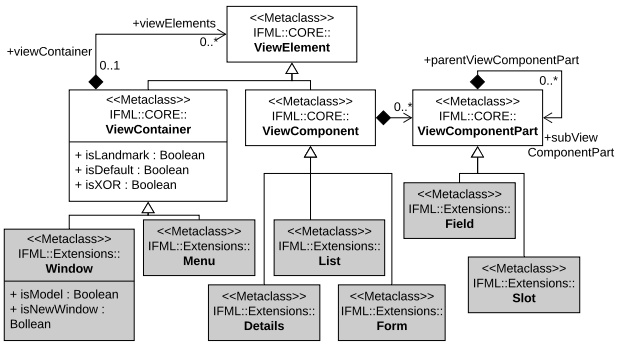


Figure 2: IFML - View Elements

ViewElement. A **ViewElement** can be refined as a **ViewContainer** or a **ViewComponent**.

A **ViewContainer** represents a container of other **ViewContainer**s or **ViewComponent**s. For example, it can be a window, an HTML page, a panel *etc.* The composition between **ViewContainer** and **ViewElement** is used to define the DOM.

A **ViewComponent** corresponds to a widget which displays its content, *e.g.* a form, a data grid, an image gallery *etc.* It can be linked to multiple **ViewComponentPart**. A **ViewComponentPart** represents an element of a **ViewComponent**. For example, an input field inside a form, a text which is displayed inside a data grid, or an image element of a gallery.

2.2.2 GUI meta-models

Other GUI meta-models have been proposed in the literature. We compare them to the OMG standards.

All the meta-models use the Composite pattern to represent the DOM of a GUI and define a kind of **UIResource** entity to represent a graphical element of the interface.

Gotti and Mbarki [9] and Sánchez Ramán *et al.* [23] proposed a meta-model inspired by the KDM models. The meta-model has the main entities defined by KDM. Both authors added the Attribute entity to the meta-model. They also define different kinds of widgets such as Button, Label, Panel, *etc.*

Fleurey *et al.* [7] did not explicitly describe the GUI meta-model, but we extracted information from their navigation model. They have at least two elements in their UI model that represent a **Window** and a **GraphicElement**. The Window corresponds to the Display entity of the KDM model. And because the **GraphicElement** and the **Window** are not linked, we can suppose that the **GraphicElement** is a UIRessource. The **GraphicElement** has an **Event**.

Morgado *et al.* [20] used a UI meta-model but did not describe it. We only know that the UI is represented as a tree which is similar to the DOM.

The UI meta-model of Garcés *et al.* [8] differs a lot from the previous ones. There are the attributes, the events, and

the screen but the notion of widget is present as a field which displays data of a table. They also used an **Event** entity to represent the interaction of the user with the user interface. The **Event** entity corresponds to the **Action** and the **Event** entities of the KDM model.

Memon *et al.* [15] represented a user interface with only two entities. A UI window which is composed of a set of widgets that can have attributes. Representing the DOM was not in the scope of their work. It is not possible to represent it with their meta-model.

Samir *et al.* [22] worked on the migration of Java-Swing applications to Ajax Web applications. They created a meta-model to represent the UI of the original application. This meta-model is stored in a XUL (XML-based User interface Language) file and represents the widgets with their attributes and the layout. Those widgets belong to a **Window** and can fire events when a UI input is performed. The input and the event correspond to the **Action** and the **Event** entities of the KDM model. The XUL format has been discontinued.

Shah and Tilevich [24] used a tree architecture to represent the UI. It allows them to model the DOM. The root of the tree is a **Frame**. It corresponds to the **UIDisplay** entity. The root contains components with their attributes.

Joorabchi and Mesbah [11] represented a user interface with a set of UI elements. Those elements correspond to the definition of a **UIField**. For each UI element, the authors' tool is able to handle the detection of multiple attributes and actions.

Memon [16] used a UI Model to represent the state of an application. A state is defined as the GUI's widgets and their properties.

Mesbah *et al.* [17] did not present directly their meta-model for the user interfaces. However, they explain that they use a DOM-tree representation to analyze different web pages. They also used the notion of events that can be fired. They used different instances of their UI meta-model to represent the web pages of the application. Those instances can be compared to multiple UIDisplay entities.

All the authors used the notion of widget that represents a visual entity of the user interface. Most of them have an entity attribute that represents a characteristic of a widget. Finally, the navigation links are represented with an action entity.

3 Context of the migration project

The goal of our work is to migrate the user interfaces from a given graphical interface framework to another. This is an industrial project, migrating web applications from GWT to Angular. The objective is to produce a running user interface in the target framework. We now present the conditions of the projects. In Section 3.1 we list some constraints that we must fulfill. In Section 3.2 we describe the main differences between GWT applications and Angular ones. In Section 3.3 we present a categorization of the front-end source code.

3.1 Constraints

From the previous works of Moore *et al.* [18] and Sánchez Ramán *et al.* [23], we identify the following constraints for our case study:

- *From GWT to Angular*. In the context of the collaboration with Berger-Levrault, our migration approach must work with Java GWT as source language and TypeScript Angular as target one.
- *Approach adaptability*. Our approach should be as adaptable as possible to different contexts. For example, it can be used with different source and target languages. This constraint includes the *Source and target independence* and the *Modularity* constraints.
- *Keep visual aspect*. The migration must keep the visual aspect of the target application as close as possible to the original. This constraint includes the *Layout-preserving migration* which it is in opposition to the *GUI Quality improvement*.
- *Code quality conservation*. As a relaxed *Code Quality improvement* constraint, our approach should produce code that looks familiar to the developers of the source application. As far as possible, the target code should keep the same structure, identifiers and comments. However, we will see in the next section that there are strong differences in GWT and Angular organization schemas.
- *Automatic*. An automatic solution makes the approach more accessible. It would be easier to use an automatic approach on large system [18]. This constraint corresponds to the *Automation* constraint of the literature.

3.2 Comparison of GWT and Angular

In this project, the source language and the target language impose two different organization schemas. Their differences are syntactic and semantic.

GWT is a framework that allows developers to write a web application in Java. The GUI code is compiled to HTML, CSS and JavaScript code. Angular is a front-end web application platform that allows developers to write a web application with the TypeScript language. It is used to create Single-Page Applications[1].

Table 1 summarizes the differences between GWT applications and Angular ones, concerning: web page definition, their style and the configuration files. Before explaining these three differences, we note one major similarity:

[1]Single-Page Applications (SPAs) are Web apps that load a single HTML page and dynamically update that page as the user interacts with the app.

Table 1: Comparison of GWT and Angular organization schema

	GWT	Angular
Web page definition	One Java class	One TypeScript file and one HTML file
Main visual aspect of the application	One CSS file	One CSS file
Specific visual aspect of a web page	Included in the Java file	One optional CSS file
Number of configuration files	One file	Two files

both GWT and Angular applications have a main CSS file to define the general visual aspect of the application.

- **Web Page Definition.** In the GWT framework, only one Java file is necessary to define a web page (an excerpt is proposed in Figure 5, page 7). The Java (GWT) file includes the main graphical components (widgets) of the web page, their positions and hierarchical organization. In the case of an actionable widget (as a button), the action is implemented in the same file. In Angular, there is a file hierarchy for each web page. Each web page is considered as a sub-project independent of the others. A sub-project contains two files: an HTML file, containing the widgets of the web page and their organizations; and a TypeScript file, containing the code to execute when an action is performed.
- **Visual Aspect.** The visual aspect of a web page includes color or dimension of specific displayed elements. In the case of GWT, the specific visual aspect is defined in the Java file of the web page definition. In Angular, there is an optional distinct CSS file.

```
1 <application name="CORE-Incubator">
2   <module name="KITCHENSINK">
3     <phase codePhase="KITCHENSINK_HOME"
4       className="fr.bl.client.kitchensink.
      PhaseHomeKitchenSink"
5       title="Home"/>
6   </module>
7 </application>
8
```

Figure 3: Example of a GWT configuration file in XML

- **Configuration Files.** For the configuration files, GWT uses one XML file that defines the binding between a Java file, a web page and the URL of the web page. Figure 3 presents a snippet of the XML file of a Berger-Levrault application. The tag **phase** (line 3) defines a web page of the GWT application: The web page title is "Home"; it is defined by the Java class `PhaseHomeKitchenSink` (in package `fr.bl.client.kitchensink`); and the URL to access the web page is `http://myserver.com/KITCHENSINK_HOME`. For Angular, there are two configuration files: *module*, defines the components of the application, *e.g.* web pages, distant services and graphical component; and, *routing*, defines for each web page, its associated URL.

3.3 Front-end application structure

As proposed by Hayakawa *et al.* [10], we divided the migration project in multiple sub-problems. To do so, we define three categories of source code: the visual code; the behavioral code; and the business code.

- **Visual Code** The visual code describes the visual aspect of the GUI. It contains the elements of the interface. It defines the inherent characteristics of the components, such as the ability to be clicked or their color and size. It also describes the position of these components compared to others.
- **Behavioral code** The behavioral code defines the action/navigation flow that is executed when a user interacts with the GUI. The behavioral code contains control structures (*i.e.* loop and alternative).
- **Business code** The business code is specific to an application. It includes the rule of the application, the distant server address and the application-specific data.

Because of the size and diversity of source code, migrating one of this code category is already an important problem.

4 Migration Approach

This section presents the migration approach we designed. In Section 4.1, we describe the migration process we designed. Section 4.2 presents our GUI meta-model.

4.1 Migration process

From the state of the art, the constraints and the decomposition of the user interfaces, we designed an approach for the migration.

The process, represented in Figure 4, is divided into the three following steps:

1. *Extraction of the source code model.* We build a model representing the source code of the original application. In our case study, the source program is written in Java GWT. The extraction produces a

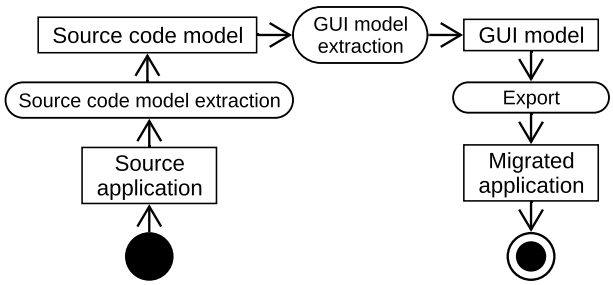


Figure 4: Our GUI migration process

FAMIX model [5] of the application using a meta-model capturing Java concepts. We also need to parse the XML configuration file described in Section 3.2.

2. *Extraction of the GUI model.* We analyze the source code model to detect the *Visual code* elements describing the GUI and we build a GUI model from these elements. The GUI meta-model is described Section 4.2.

3. *Export.* We re-create the GUI in the target language. This step exports the user interface files and the configuration files of the application.

Note that currently we treat neither the *Business code* nor the *Behavioral code* of the application. This will be the focus of future work.

4.2 GUI meta-model

To represent the user interfaces of desktop or web-based applications, we designed a GUI meta-model from the ones presented in Section 2.2.2. In the rest of this section, we present the entities of the meta-model.

Our meta-model is an adaptation of KDM meta-model(see Figure 1). As many others, we separate the graphical resources corresponding to the DOM from the actions and events. In our meta-model, graphical resources are called **Widget**. They can be refined as **Leafs** or **Containers**

In our context, the user interface will always be displayed on a screen. So we do not represent all the kind of UIDisplay and define an entity **Page**. The **Page** represents the main container of a graphical interface. It is either a *window* of a desktop application or a web page. The **Page** is a kind of **Container**.

As proposed by many other authors, we added the entity **Attribute** in our GUI meta-model. An **Attribute** represents the information of a widget and can change its visual aspect or behavior. Some common attributes are the height and the width to precisely define the size of a widget. There are also attributes that contain data. For example, a widget representing a button may have a *text* attribute that contains the text of the button. An attribute can change the behavior of a widget, this is the case of the attribute *enable*. A button with the *enable* attribute set to *false* represents a button on which one cannot click. Finally, the widgets can have attributes that impact the visual aspect of the application. This type of attribute allows us to define a layout to be respected by the widgets contained in the main one and potentially the dimensions of the latter to respect a particular layout.

5 Implementation

To test our approach, we implemented a migration tool. It is implemented in Pharo[2] and the meta-model is represented using the Moose platform[3].

5.1 Case Study

Applications at Berger-Levrault (our industrial partner) are based on the BLCore framework. This framework consists in 763 classes in 169 packages. It is developed by the company on top of GWT and defines the widgets that developers should use, the default visual aspect of the applications, and Java classes to connect the front-end of the application to the back-end. It also encourages some coding conventions.

For the Berger-Levrault applications, we add a new entity (**Business Page**) to the GUI meta-model presented Section 4.2 to fit the company's specific organization. It is a kind of **Container**. One convention is that each **Page** has one or more **Business Pages** represented as tabs in the **Page**. The **widgets** (buttons, text fields, tables, ...) are included in the **Business Pages**, never directly in the **Page**.

5.2 Import

In part because of the complexity of setting up a tool to run automatically and capture all screens of such large web applications, we rely on static analysis to create our model. The results so far seem to indicate that it will be sufficient.

As presented Section 4.1, the creation of the GUI model is divided in two steps: the source code model extraction and the GUI model extraction. For the source code meta-model, we use the Java meta-model of Moose [5, 21] which comes with a Java extractor[4]. Figure 5 presents a snippet of the source code of a Berger-Levrault application. It shows the method buildPageUi(Object object) that builds the GUI of the business page SPMetier1 (a "simple business page").

For the second step of the extraction, our tool creates the GUI model from the source code model and an analysis of the XML configuration file. The entities we want to extract are, first, the **Pages**. We parse the XML configuration file in which is defined the information about the pages (see

[2]Pharo is a pure object-oriented programming language inspired by Smalltalk. http://pharo.org/

[3]Moose is a platform for software and data analysis. http://www.moosetechnology.org/

[4]verveineJ : https://github.com/moosetechnology/verveineJ

```
1  class SPMetier1 extends AbstractSimplePageMetier
       {
2    @Override
3    public void buildPageUi(Object object) {
4      BLLinkLabel lblPg = new BLLinkLabel("Next");
5      lblPg.addClickHandler(new ClickHandler(){
6        public void onClick(ClickEvent event) {
7          SPMetier1.this.fireOnSuccess("param");
8        }
9      });
10     lblPg.setEnabled(false);
11     vpMain.add(new Label("<Business content>"));
12     vpMain.add(lblPg);
13     super.setBuild(true);
14   }
15 }
```

Figure 5: User interface creation in GWT

Section 3.2). It provides for each **Page** (called *phase* in the XML file, Figure 3) its name and the name of the Java class that defines it. Then, the tool looks for **Widgets**.

First, the tool determines the available widgets. To do so, it collects all the Java subclasses of the GWT class Widget. For the **Buisiness pages**, the tools looks for the classes that implement the IPageMetier interface. Then, the tool looks where the **Widget** constructors are called and creates the links between the **Widgets** and their parent **Widget**. In Figure 5, there are two calls to **Widget** constructors: line 4, the constructor of BLLinkLabel is called, and line 11, the constructor of Label. The variable vpMain corresponds to the main panel of the **Business page**. Lines 11 and 12 correspond to adding a widget inside a panel widget thanks to the method add().

Finally, to detect attributes and actions which belong to a widget, the tool detects in which Java variable the widget has been assigned. Then, it searches the methods invoked on this variable. If a widget invokes the method *"addClickHandler"*, it creates an event. If it invokes a method *"setX"*, it creates an attribute. These heuristics were found in the literature [22, 25]. In Figure 5, the BLLinkLabel, whose variable is lblPg, is linked to one event and one attribute. Lines 5 to 9 correspond to the creation of one event with the executable code. Line 10 corresponds to adding the attribute enabled with the value false.

5.3 Export

Once the GUI model is generated, it is possible to export the application. To generate the code of the target application, the tool includes an exporter. The exporter creates the folders of the target application and the configuration files. Then, it visits the pages. For each **Page**, the exporter creates an Angular sub-project in the form of a directory containing several configuration files and a default blank web page. Then, for each business page of the current visited **Page**, the exporter generates one HTML file and one TypeScript file. For the HTML file, the exporter builds the DOM thanks to the Composite pattern used by the GUI meta-model (see Section 4.2). Each widget provides its attributes and actions to the exporter.

6 Validation

In this section, we describe the industrial application on which we used our tool to validate our approach. Section 6.1 presents the industrial application. Section 6.2 presents the metrics we used to evaluate our approach.

6.1 Industrial application

We experimented our strategy on Berger-Levrault's *kitchensink* application. This software system, dedicated to developers, aims to gather inside a single simple application all the components available for building a user interface. This application is smaller than a production one but still uses the BLCore framework. The company framework guarantees us the *kitchensink* application works exactly the same way as the industrial applications. It contains 470 Java classes and represents 56 web pages. Although it is the sample and demo for developers, the *kitchensink* application contains code misuses.

Note that the *kitchensink* application, as the other industrial applications of the company, does not have test. Therefore, there is no possibility to use tests to validate the migration.

6.2 Validation metrics

The validation is done in three steps: First, we check the constraints defined in Section 3.1; Second, we validate that all GUI entities of interest are extracted and correctly extracted; Third, we validate that we can re-export these entities in Angular and that the result is correct.

For the first validation, we manually identify and count the entities in the *kitchensink* application and compare the results of the tool to this count. Our analysis focuses on the migration of three entities: **Pages**, **Business Pages** and **Widgets**

- **Pages.** From the XML configuration file of the application we manually count 56 pages. This configuration file also provides the name of each page.
- **Business Pages.** As explained before, the business pages correspond to a concept specific to Berger-Levrault. They are defined in the BLCore framework as a Java class which implements the interface IPageMetier. Thanks to this heuristic, we manually count 76 **Business Page** instances in the original application.
- **Widgets.** In the literature survey, we did not find an automatic way to evaluate the detection of widgets. Checking all widgets in the application would be long

and error-prone as there are thousands of them. As a fallback solution, we take a sample of the pages of the *kitchensink* application and count the widgets in the DOM of these pages. We consider a sample of 6 **Pages** which represents a bit more than 10% of the **Pages** of the application. These **Pages** are of different sizes and contain different kinds of widgets. In total, we found 238 **Widgets** in these 6 **Pages**. To get a more exact idea of the representativeness of our sample, we also count the number of **Widget** creation (*i.e.* `new AWidgetClass()`) in the code. There are 2,081 such creation. This may not represent the exact number of widgets in the entire application, but it is a good estimate. We note that the number of **Widgets** in our sample (slightly more than 10% of the pages) is also slightly more than 10% of our estimate of the total number of widgets.

For the evaluation, we also check that the **Widgets** are correctly placed in the DOM of the interface (*i.e.*, they belong to the right **Container** in the GUI model).

In our results we consider only the recall of the tool because the precision is always 100% (there are no false positive). This is a sign that the BLCore framework provides clear (if not complete) heuristics to identify the entities.

For the second validation, we check that the entities are exported correctly. In the Angular application, each **Page** corresponds to a sub-project and is represented by a folder. The name of the folder must correspond to the name of the **Page**. The **Business pages** are represented by a sub-folder inside the **Page** project. The names must also match at this level.

We also check visually that the exported **Page** "looks like" the original one. This is a subjective evaluation, and we are looking for options to automate it in the future.

7 Results

This section presents the results of the migration validation on Berger-Levrault's *kitchensink* application. Section 7.2 summarizes the extraction results. In Section 7.1, we confront the exported result with the constraints defined in Section 3.1.

7.1 Satisfaction of constraints

We set the following constraints in Section 3.1: *From GWT to Angular*, *Approach adaptability*, *Code quality conservation*, *Keep visual aspect*, and *Automatic*.

Our tool can use Java code as input and generate Angular code. The exported code is compilable and executable. The target application can be displayed. We can thus confirm that our tool fulfill the GWT to Angular constraint.

Our tool is applicable on other source target technologies. Our heuristics have been designed to be easy to adapt, a user of our tool can thus add a new kind of widget for the import or the export phases. We shortly describe a small experiment in that sense in Section 8.5. Those possibilities satisfied the adaptability constraint.

The *Code quality conservation* and *Keep visual aspect* constraints are discussed in Section 7.3, in the third validation results.

Finally, the results described here were obtained automatically from application of our tool to the subject application. This validates the last constraint.

7.2 Extraction results

Table 2 summarizes the extraction results.

Table 2: Extraction results

	Pages	**Business Pages**	**Widgets** (sample)
Number	56	76	238
Correctly imported	100%	100%	89%

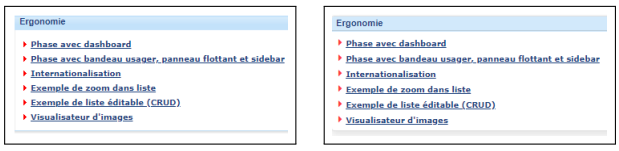
The tool extracted 56 **Pages** from the original GUI. This corresponds to the number of pages defined in the configuration file of the *kitchensink* application.

The tool extracted 76 **Business pages**. This value corresponds exactly to the number of business pages in the original application. Moreover, the tool correctly assigned each **Business page** to its proper **Page**.

We got 100% of the **Widgets** on the evaluated sample were correctly detected. However, 27 out of the 238 **Widgets** of our sample (11%) were not correctly assigned to their parent container. All these problems come from one single **Page** (containing 75 **Widgets** in total).

7.3 Export results

We manually checked the name of all the 56 exported pages. They all conserve their original name.

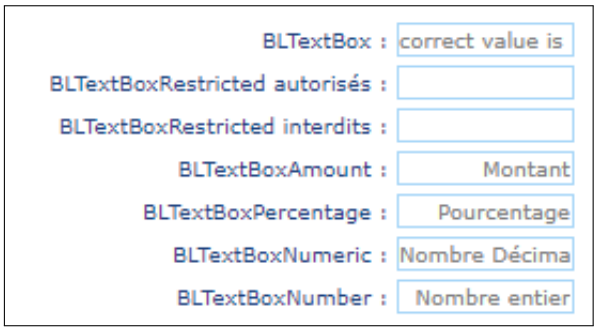


(a) GWT original (b) Angular migration

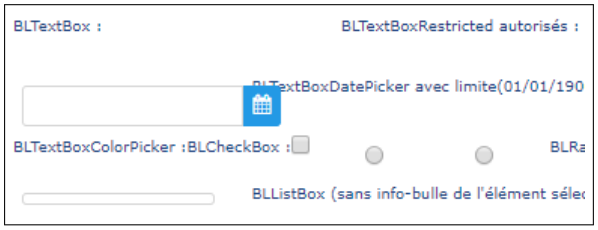
Figure 6: Visual comparison of a **Page** migration

Figure 6 presents the visual differences between the original (GWT) version, left hand, and the migrated (Angular 6) one, right-hand. We can see that there are only minimal differences. In the exported version, the color of the header is a bit clearer, and the lines are a little more distant.

Figure 7 presents the visual differences for the **Page** *Input box*. Again on the left-hand side there is the original **Page** and on the right-hand side the same **Page** after the migration. Because the two images are large, we



(a) GWT original



(b) Angular migration

Figure 7: Visual comparison of a **Page** migration: All the **Widgets** are migrated but with a wrong layout.

trimmed them to display this area of interest. Even though the two images look completely different, all the widgets are present in the migrated version. The visual differences are due to a problem in the layout management. The visual constraint is thus partially satisfied. This point is discussed Section 8.1.

8 Discussion

Section 8.1 and Section 8.2 presents two parts of the user interface we did not work on. Section 8.3 discusses the impact of the choice of the *kitchensink* application as case study. Section 8.4 highlights the difficulties in large scale validation of our tool. Finally, Section 8.5 discusses the impact of the BLCore framework.

8.1 Layout management

As shown in Section 7.3, the export may give incorrect results because of layout issues (Figure 7). It is due to the representation of the layout in our GUI meta-model. Currently, layouts are represented in our GUI meta-model as an attribute on a **Container widget** defining if the children of this widget are placed ones beside the other or one below the other (*i.e.* vertical or horizontal flow layouts). However, many other layouts exist [14]. For example, the BLCore framework offers `BLGrid`, a **Widget** inheriting from the GWT `Grid` class and implementing a grid layout. Currently, complex layouts are not considered in our GUI meta-model.

A solution is proposed by Sánchez Ramán *et al.* [23]. They designed a layout meta-model. The idea consists in linking widgets with a layout and combining the layouts to create a precise representation. The authors defined a subset of possible layout to connect to widgets.

Moreover, with such layouts, the position of the children **Widgets** might be computed at run time. For example, in a grid layout, the children may be positioned according to the values of some `row` and `col` variable. Guessing these values with a static analysis is not practical, and this is a case where an hybrid approach might be necessary.

8.2 Managing behavioral and business code

Currently, only the visual part of the GUI is migrated. To take into account the whole application, the migrations of the *Behavioral* and *Business code* (see Section 3.3) are needed. The *Behavioral* represent the user interactions (*i.e.* click, drag-and-drop, hover, ...) and the control structures (*i.e.* loop and alternative). In the case of a client/server application, requesting a server is part of *Behavioral code*, whereas the query in itself and the data belongs to *Business code*.

8.3 Demo application

Although the results are encouraging, we only evaluated our tool on the *kitchensink* application. The *kitchensink* application is a good training ground for our tool as it contains all kinds of widgets that developers have at their disposal and the way to use them. However, it might diverge from production applications as it should contain less irregularities or coding tricks than the later.

8.4 Validation tools

The automatic validation of the screens migration is currently an unsolved problem. It is possible to manually check the result of the migration for a few pages but it would be better to do it automatically for hundreds of pages (more than 400 on Berger-Levrault applications).

We found, in the literature, only few approaches considering automatic visual validation. In two papers [11, 23], the authors simply count the number of widgets in the source application and target applications. But we saw in Figure 7 that this not guarantee visual similarity. An other article [19] propose to compare screenshots of the original and the exported applications pixel by pixel. However, we saw in Figure 6 that barely distinguishable screens may have differences at the level of pixels.

8.5 Impact of BLCore

As explained in Section 5.1, the Berger-Levrault applications are based on the BLCore framework. By specializing GWT, BLCore provides specific widgets and a dedicated

API. This may have an impact on our approach or not. To evaluate this possible impact, and also to validate the generality of our approach, we performed two small experiments considering (i) Spec (a desktop UI framework in Pharo [6]) as the source framework and, (ii) Seaside (a web framework in Pharo [4] – also described at seaside.st) as the target framework. These experiments thus consider different programming languages (Pharo instead of Java (GWT) and TypeScript), different GUI frameworks, and desktop and web applications. We experimented migrating the GUI of small demo application from Spec to Angular, and migrating the Berger-Levrault *kitchensink* application to Seaside.

Some conclusions are:

- It was harder to import Spec code than GWT because of a larger variability in defining the GUI. We conclude that the BLCore framework eased our work on the import by standardizing how to build the pages.
- For Seaside, it was easy to migrate simple widgets (*e.g.* Label, Button, Panel), but the BLCore framework also defines complex widgets with no direct equivalent in Seaside. A library similar to BLCore should be defined in Seaside to ease the migration.
- the power of our GUI meta-model and the two steps extraction (first, source code model extraction, then GUI model extraction, see Figure 4) are validated by the fact that we were able to migrate a Pharo desktop application with little extra work.

9 Conclusion and Future works

We created a tool with promising results on the representation of GUI to migrate GWT applications toward Angular. In the following, we conclude the presentation of this work and propose some future research directions we want to explore.

9.1 Conclusion

In this paper, we exposed a preliminary work on the problem of visual preservation and respect of the target architecture during the GUI migration of an application. We proposed an approach based on a GUI meta-model and a migration process in three steps. We implemented this process in a tool to perform the migration fof GWT applications to Angular 6. Then, we validated our tool with an experiment on a *kitchensink* application. We were able to extract correctly all pages of the application and 89% of the widgets. The migration results are visualizing equivalent as long as complex widgets (*e.g.* GridLayout) are not used. Dealing with these layouts is our next challenge.

Our solution also allows us to respect the naming conventions used in the source application as well as the structure of the code as far as the differences in the GUI frameworks allow it.

9.2 Future work

To improve the migration of an application user interface, we will enhance our meta-model and our tool to support the management of the layout and the behavioral and business code.

We did not find an approach or metrics to automatically evaluate the validity of the migrated screens. So, it is important to find a new way to evaluate that the migrated screens conserve the visual aspect of the original ones.

Having a good GUI meta-model also opens the door for a generic GUI builder that could export the GUI in several different GUI frameworks.

References

[1] Marco Brambilla and Piero Fraternali. *Interaction flow modeling language: Model-driven UI engineering of web and mobile apps with IFML*. Morgan Kaufmann, 2014.

[2] John Brant, Don Roberts, Bill Plendl, and Jeff Prince. Extreme maintenance: Transforming Delphi into C#. In *ICSM'10*, 2010.

[3] Jonathan Cloutier, Segla Kpodjedo, and Ghizlane El Boussaidi. WAVI: A reverse engineering tool for web applications. In *2016 IEEE 24th International Conference on Program Comprehension (ICPC)*, pages 1–3. IEEE, 2016.

[4] Stéphane Ducasse, Lukas Renggli, C. David Shaffer, Rick Zaccone, and Michael Davies. *Dynamic Web Development with Seaside*. Square Bracket Associates, 2010.

[5] Stéphane Ducasse, Nicolas Anquetil, Usman Bhatti, Andre Cavalcante Hora, Jannik Laval, and Tudor Girba. MSE and FAMIX 3.0: an Interexchange Format and Source Code Model Family. Technical report, RMod – INRIA Lille-Nord Europe, 2011.

[6] Johan Fabry and Stéphane Ducasse. *The Spec UI Framework*. Square Bracket Associates, 2017.

[7] Franck Fleurey, Erwan Breton, Benoit Baudry, Alain Nicolas, and Jean-Marc Jezéquel. Model-Driven Engineering for Software Migration in a Large Industrial Context. In Gregor Engels, Bill Opdyke, Douglas C. Schmidt, and Frank Weil, editors, *Model Driven Engineering Languages and Systems*, volume 4735, pages 482–497, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

[8] Kelly Garcés, Rubby Casallas, Camilo Álvarez, Edgar Sandoval, Alejandro Salamanca, Fredy Viera, Fabián Melo, and Juan Manuel Soto. White-box modernization of legacy applications: The oracle forms case study. *Computer Standards & Interfaces*, pages 110–122, October 2017.

[9] Zineb Gotti and Samir Mbarki. Java swing modernization approach - complete abstract representation based on static and dynamic analysis:. In *Proceedings of the 11th International Joint Conference on Software Technologies*, pages 210–219. SCITEPRESS - Science and Technology Publications, 2016.

[10] Tomokazu Hayakawa, Shinya Hasegawa, Shota Yoshika, and Teruo Hikita. Maintaining web applications by translating among different ria technologies. *GSTF Journal on Computing*, page 7, 2012.

[11] Mona Erfani Joorabchi and Ali Mesbah. Reverse engineering iOS mobile applications. In *2012 19th Working Conference on Reverse Engineering*, pages 177–186. IEEE, 2012.

[12] R. Kazman, S.G. Woods, and S.J. Carriére. Requirements for integrating software architecture and reengineering models: Corum ii. In *Proceedings of WCRE '98*, pages 154–163. IEEE Computer Society, 1998.

[13] Valéria Lelli, Arnaud Blouin, Benoit Baudry, Fabien Coulon, and Olivier Beaudoux. Automatic detection of GUI design smells: The case of blob listener. *EICS '16 Proceedings of the 8th ACM SIGCHI Symposium on Engineering Interactive Computing Systems*, page 12, 2016.

[14] Simon Lok and Steven Feiner. A survey of automated layout techniques for information presentations. *Proceedings of SmartGraphics*, 2001:61–68, 2001.

[15] Atif Memon, Ishan Banerjee, and Adithya Nagarajan. GUI ripping: reverse engineering of graphical user interfaces for testing. In *Reverse Engineering, 2003. WCRE 2003. Proceedings. 10th Working Conference on*, pages 260–269. IEEE, 2003.

[16] Atif M. Memon. An event-flow model of GUI-based applications for testing. *Software Testing, Verification and Reliability*, 17(3):137–157, 2007.

[17] Ali Mesbah, Arie van Deursen, and Stefan Lenselink. Crawling ajax-based web applications through dynamic analysis of user interface state changes. *ACM Transactions on the Web*, 6(1):1–30, 2012.

[18] Moore, Rugaber, and Seaver. Knowledge-based user interface migration. In *Proceedings 1994 International Conference on Software Maintenance*, pages 72–79. IEEE Comput. Soc. Press, 1994.

[19] Kevin Moran, Cody Watson, John Hoskins, George Purnell, and Denys Poshyvanyk. Detecting and Summarizing GUI Changes in Evolving Mobile Apps. *arXiv:1807.09440 [cs]*, July 2018.

[20] I Coimbra Morgado, Ana Paiva, and J Pascoal Faria. Reverse engineering of graphical user interfaces. In *ICSEA 2011 : The Sixth International Conference on Software Engineering Advances*, 2011.

[21] Oscar Nierstrasz, Stéphane Ducasse, and Tudor Gîrba. The story of Moose: an agile reengineering environment. In Michel Wermelinger and Harald Gall, editors, *Proceedings of the European Software Engineering Conference*, ESEC/FSE'05, pages 1–10, New York NY, 2005. ACM Press.

[22] Hani Samir, Amr Kamel, and Eleni Stroulia. Swing2script: Migration of Java-Swing applications to Ajax Web applications. In *14th Working Conference on Reverse Engineering (WCRE 2007)*, 2007.

[23] Óscar Sánchez Ramán, Jesús Sánchez Cuadrado, and Jesús García Molina. Model-driven reverse engineering of legacy graphical user interfaces. *Automated Software Engineering*, 21(2):147–186, 2014.

[24] Eeshan Shah and Eli Tilevich. 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*, pages 255–260. ACM, 2011.

[25] João Carlos Silva, Carlos C. Silva, Rui D. Goncalo, João Saraiva, and José Creissac Campos. 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*, pages 181–186. ACM Press, 2010.

[26] Stefan Staiger. Reverse engineering of graphical user interfaces using static analyses. In *14th Working Conference on Reverse Engineering (WCRE 2007)*, pages 189–198. IEEE, 2007.

[27] Christian Zirkelbach, Alexander Krause, and Wilhelm Hasselbring. On the modernization of explorviz towards a microservice architecture. In *Combined Proceedings of the Workshops of the German Software Engineering Conference 2018*. CEUR Workshop Proceedings, 2018.