Migrating GUI behavior: from GWT to Angular

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

Migrating GUI behavior: from GWT to Angular

Benoît Verhaeghe[3,1] Anas Shatnawi[3] Abderrahmane Seriai[3] Nicolas Anquetil[1]

Anne Etien[1] Stéphane Ducasse[2] Mustapha Derras[3]

[1]Univ. Lille, CNRS, Inria, Centrale Lille, UMR 9189 CRIStAL, France
{firstname.lastname}@inria.fr

[2]Univ. Lille, Inria, CNRS, Centrale Lille, UMR 9189 CRIStAL, France
{firstname.lastname}@inria.fr

[3]Berger-Levrault, France
{firstname.lastname}@berger-levrault.com

***Abstract*—In a collaboration with Berger-Levrault, a major IT company, we are working on the migration of GWT applications to Angular. We focus on the GUI aspect of this migration which requires a framework switch (GWT to Angular) and a programming language switch (Java to TypeScript). Previous work identified that the GUI can be split into the UI structure and the GUI behavioral code. GUI behavioral code is the code executed when the user interacts with the UI. Although the migration of UI structure has already been studied, the migration of the GUI behavioral code has not. To help developers during the migration of their applications, we propose a generic approach in four steps that uses a meta-model to represent the GUI behavioral code. This approach includes a separation of the GUI behavioral code into events (caller code) and the code executed when an event is fired (called code). We present the approach and its implementation for a real industrial case study. The application comprises 470 Java (GWT) classes representing 56 web pages. We give examples of the migrated code. We evaluate the quality of the generated code with standard tools (SonarQube, codelizer) and compare it to another Java to TypeScript converter. The results show that our code has 53% fewer warnings and rule violations for SonarQube, and 99% fewer for codelizer.**

***Index Terms*—Graphical User Interface, GUI behavioral code, Model-Driven Engineering, Migration**

I. Introduction

Our work takes place in a collaboration with Berger-Levrault, a major IT company developing large applications using the Google Web Toolkit (GWT) framework. GWT allows developers to create web-based front-end Java applications. The GWT framework is no longer intensively maintained with only two minor releases since 2015. As a consequence, Berger-Levrault decided to migrate the GUI of its applications to Angular 12.

The migration of an application's GUI has already been studied [1, 2, 3, 4]. However, the authors of previous work only considered the visual part of the application, *i.e.* migrating the widgets and their position; and not the GUI behavior. Thus, there is still a need to support the migration of the GUI behavioral code.

The GUI behavioral code is the code executed when the user interacts with the UI. For example, when users click on a button, display a Popup. So, migrating GUI behavioral code consists of migrating the source code executed when the end-user interacts with the UI.

Tools and approaches that migrate source code from one framework to another or from one programming language to another have already been developed [5, 6]. However, the migration of source code relative to the GUI is still one of their most complex challenges [7]. Moreover, unlike approaches proposed by language translators such as JSweet[1] and CodeTranslater[2], GUI behavioral code migration requires a language migration and an API migration, *e.g.* migrating GUI handlers between GUI framework requires a specific approach.

Thus, we must provide an approach that helps to migrate GUI behavioral code. Such an approach should migrate the code and ensure the maintainability of the new application by generating natural code, *i.e.* respecting the conventions of the target language as used by an expert.

This paper presents an approach to migrate the GUI behavioral code. The approach comes with a meta-model representing the GUI behavioral code and is linked to the visual part of the application. We detail the steps to extract behavior and generate the target code. We implemented a prototype that performs the migration of GWT code to Angular. Then, we applied it on a real industrial context, and assess the result both manually and with standard quality evaluation tools.

The contributions of the paper are:

- an approach to migrate application behavioral code;
- a meta-model to represent GUI behavioral code linked to the GUI visual part;
- a tool that implements our approach for GWT behavioral migration; and
- a validation of the approach in an industrial context.

In Section II, we review the literature on language migration. In Section III, we define the GUI behavioral code from an example. In Section IV, we detail the migration process of GUI behavioral code. In Section V and Section VI, we present an implementation of our approach. In Section VII, we present an evaluation of the migration of our partner company application. In Section VIII, we discuss our results and the

[1]JSweet: http://www.jsweet.org/papers-and-publications/

[2]CodeTranslater: https://www.carlosag.net/tools/codetranslator/

genericity of our approach. In Section IX, we conclude and present future work.

II. State of the art

To the best of our knowledge, there is no study on the migration of code executed when the user interacts with the UI. So, we present related work on the GUI structure (in Section II-A). In Section II-B, we present existing migration approaches. Finally, we detail language migration tools and approaches (in Section II-C).

A. GUI Structure

Hayakawa et al. [8] split the GUI code into multiple parts. For instance, GUI is divided into two categories of code: visual and behavioral.

Visual code describes the visual aspect of the GUI. It is composed of the *Meta*, *Widget*, and the *Style* part. They correspond respectively to UI meta-information (such as the UI title), the type of the widgets and the Domain Object Model (DOM), and the style of the element (color, size, *etc.*).

Behavioral code is defined as the executed script when an event (such as a click event) is fired.

Many approaches have considered the migration of the visual code [1, 2, 3, 9, 10]. However, the behavioral code must also be considered to migrate the GUI completely. We will thus detail the migration approaches found in the literature.

B. Migration approaches

Sneed and Verhoef [11] described three ways to migrate an application: *conversion*, *reimplementation*, and *wrapping*.

Conversion consists in a one-step approach that translates statement by statement the source code to its target language counterpart [7].

Reimplementation is used by many approaches that migrate the GUI visual part [1, 2, 3, 9]. It follows this process:

- The old application is extracted into a source language-specific model.
- Then, the model is transformed to a higher-level representation.
- Finally, the high-level model is transformed into a target language-specific model or directly used to generate the target application.

Wrapping "is an established re-engineering technique to provide access to existing functionality through a preferred interface" [12]. In consequence, the source code is not migrated but called by the new code.

Whereas each approach allows one to execute code with the target GUI framework, only *conversion*, and *reimplementation* perform a migration. Moreover, *reimplementation* is the most used one for GUI migration. Thus, it is the one we will focus on.

When performing a *reimplementation* to a GUI framework defined in another language, one needs to perform a *language migration*.

C. Language migration

The language migration field focuses on migrating applications written in one programming language to another language. The main goal is to be able to run the migrated application.

Malton [13] classified language migrations according to their difficulties into three categories:

Dialect conversion deals with the migration from one version of a programming language to another. For example, from Python 2 to Python 3 [14].

API migration is the switch of frameworks and keeping the same programming language [15]. For example, moving from Java Swing to JavaFX.

Language migration deals with the migration from one language to another. It better fits our context.

Brant et al. [7] migrated a Delphi application into C#. To do so, they used and developed SmaCC, a transformation engine that allows one to write transformation patterns.

Terwilliger et al. [5] worked on the conversion of Fortran to C++ code. To do so, they developed FABLE which is a tool that automatically rewrites the code in C++. The authors wanted to generate C++ code suitable for future development, and at the same time "similar to the original Fortran code".

Martin and Muller [6] translated C code to Java. To translate the application they used a traditional approach: create an Abstract Syntax Tree (AST) representation of the source code, transform it into an AST for Java, and then traverse this AST to generate the target source code.

Trudel et al. [16] developed a tool that migrates C to Eiffel (an object-oriented language). To do so, they built an AST of the original source code and they applied successive transformations on this AST. Thus, they incrementally transform the code from C to Eiffel. They also manually wrote helper classes that ensure the Eiffel classes have the same capabilities as their C structure counterpart. For example, there is a helper class to access the *stdio* library aiming to help Eiffel translated code using *stdio* specific features.

III. Behavioral code

None of the previous studies propose a solution to help developers migrating GUI behavioral code. Yet, it is essential to provide such support during migration projects in addition to the migration of the visual part of the application. Moreover, GUI behavioral code is only described as "the executed script when an event is fired" [8].

Thus, before designing a migration approach, we detail what GUI behavioral code is. To do so, we first present a concrete example in Section III-A. Then, we split the GUI behavioral code into two parts and present them in Section III-B.

A. GUI behavioral code

To clarify the definition of GUI behavioral code, we use the following concrete example.

Figure 1 shows an example of GWT Java code. It corresponds to a method executed when the end-user clicks on a button of the interface to migrate. The method reads the value

```
1  button.addClickHandler(new ClickHandler() {
2    public void onClick(final ClickEvent event) {
3      String values = emailBox.getText();
4      if (values != null) {
5        List<String> results = values.split(",");
6        IGwtService.sendEmail(results, new
       AsyncCallback<List<String>>() {
7          public void onSuccess(List<String> result){
8           EventPopup.displayInfo(result.toString());
9          }
10       });
11 }}});
```

Fig. 1: GUI behavioral code in Java

of an inputText (line 3) looking for email addresses separated by commas (line 5) and uses a service to send an email to each address (line 6).

More specifically, line 1, `addClickHandler(new ClickHandler ... )` corresponds to the creation of the event click handler and attaches it to the widget `button`. This is the entry-point of the GUI behavioral code. When the click event is fired, the method `onClick` line 2 is executed.

Then, line 3, there are two behavioral elements. `emailBox` is an access to a UI element, here an input text declared in the UI. And `.getText()` is an access to the value of the attribute *text* of the widget `emailBox`.

Finally, on line 8, there is a declaration and usage of Popup. The Popup is identified by the usage of the class `EventPopup`.

All other parts of the code, not directly linked to the UI, do not belong to the GUI behavioral code. They are: control flow (*if*, line 4), business code (call to a distant service, line 6), or algorithm details (converting a string as a `List`, line 5).

Note that a similar separation of the code can be done in other programming languages (Java, TypeScript, C#, *etc.*).

B. GUI behavioral code structure

From the previous example, we subdivide the GUI behavioral code into two categories: the events and the GUI manipulation code.

Events correspond to the events raised by the system or when end users interact with the UI. Each GUI framework has a set of recognized events, however, there are some common ones, and there is an exhaustive list that can be found in each browser documentation[3].

GUI manipulation code impact or reference part of the visual aspect of the application. Examples of GUI manipulation code include showing or hiding UI elements.

For the **Events**, by analyzing the GWT applications of our industrial partner, we identified the events used in our context:

- *Click* corresponds to a user clicking on any UI element of the DOM. It can be a button as well as a table, a text, or an empty zone.
- *Change* corresponds to modification in a text input or a table.
- *Error* corresponds to a problem, for example when loading an image and the resource is unavailable.
- *Submit* corresponds to a user submitting a form.
- *SubmitComplete* is raised by the system after a successful *Submit* event, *e.g.* when the form fields were correctly filled and there is no network problem.

Since we work with web applications, this list might show some bias towards web events, however, one could easily extend the list with other events without impacting our approach. Note that events with the same name in two different GUI frameworks might behave differently [7].

For **GUI manipulations code**, there is no exhaustive list of possible expressions that impact the UI. So, we propose a first list of GUI manipulations code we found in our context.

- *Widget access.* For example, Figure 1 line 3: emailBox.
- *Widget attribute getter or setter.* For example, Figure 1 line 3: getText().
- *Navigating* corresponds to the navigation from one page to another.
- *Open Popup* shows a Popup in the application. Popups can be: *info*; *warning*; or *error*.
- *Open Dialog*[4] is the piece of code used to open a dialog in the GUI.

Again, other kinds of GUI manipulations code may exist, *e.g.* adding or removing an element in the DOM, or animations, but they were not found in our context so we did not consider them in the following. However, they could be considered without impacting our approach.

IV. Migration Process

Here, we extend the approach of [3] to support GUI behavioral code migration.

First, Section IV-A, we present the migration approach and highlight our contribution. Then, Section IV-B, we present the behavioral meta-model designed to represent the GUI behavioral code. Finally, Section IV-C, we detail the step for the extraction and the generation of the GUI behavioral code.

A. Migration approach

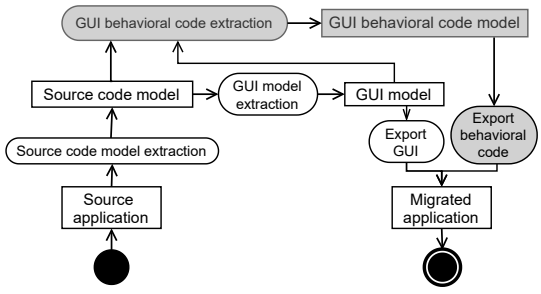

Fig. 2: Migration approach (white [3], gray our extension)

Figure 2 presents the GUI migration approach with the contributions of this paper on the behavioral code migration in gray. The migration is divided into 5 steps:

[3]For example, for Firefox: https://developer.mozilla.org/en-US/docs/Web/Events

[4]A Dialog is a window "box or other interactive components, such as a dismissable alert, inspector, or subwindow" (https://developer.mozilla.org/en-US/docs/Web/HTML/Element/dialog)

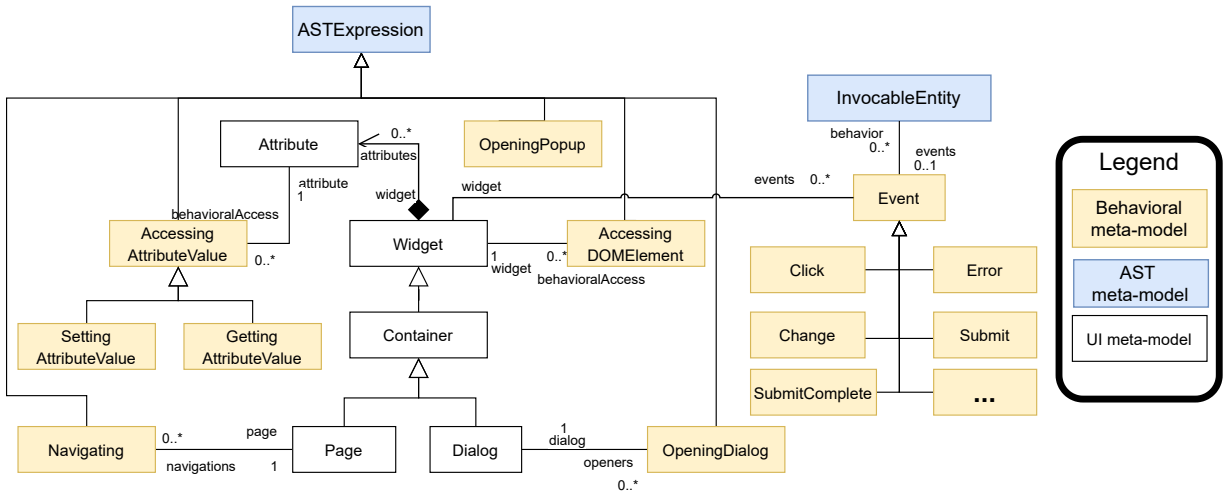

Fig. 3: Behavioral meta-model — The UI meta-model (white boxes) is based on the one described in [3]

1. *Source code model extraction* takes as input the source application and generates a source code model (in Famix [17]). One can query the model to extract concepts such as classes, methods, and method invocations.
2. *GUI model extraction* takes as input the source code model. It queries this model to extract the GUI concepts such as the widgets and their attributes.
3. ***GUI behavioral code extraction*** takes as input the extracted GUI model and the source code model. The source code model includes the GUI manipulations code and the creation of event handlers. The GUI model includes the widgets already extracted with which users can interact. It produces a GUI behavioral code model.
4. *Export GUI* takes as input the GUI model and generates the code of the visual part of the application.
5. ***Export behavioral*** exports the extracted behavioral code in the target language inside the generated GUI code.

In this paper, we focus on the behavioral part of the migration.

B. Behavioral meta-model

To represent the GUI behavioral code, we designed a meta-model Figure 3. It is based on an AST meta-model, and so it comes as an extension of FAST[5] a generic AST meta-model. As described above in Section III-B, the meta-model is divided into two parts: the `Events` raised by user interaction, and the `GUI manipulation code`.

To integrate all the behavioral concepts inside the generic AST, we defined all GUI manipulations code entities as `AST Expression`. Thus, we can transform any specific AST model into our GUI behavioral model using the `AST Expression` concept.

`Event` corresponds to the events that will be raised when the end-user interacts with the UI. It can be refined as `Click`, `Change`, `Error`, `Submit`, and `SubmitComplete` or any other event. An `Event` is linked to the AST concept `InvocableEntity` which represents an element that can be invoked, *e.g.* a method or a lambda expression. An `Event` is also linked to a `Widget` (part of the UI meta-model [3]) on which it is attached.

`AccessingDOMElement` represents the reference to any widget of the DOM. For example, it corresponds to an access to a variable containing a widget in Java. A widget can have multiple references.

`AccessingAttributeValue` represents an access to a widget attribute, and so is linked to the `Attribute` concept of the UI meta-model. It can be refined as a `GettingAttributeValue` or a `SettingAttributeValue`.

`Navigating` corresponds to the GUI manipulations code to navigate from one page of the application to another. This concept is linked to the `Page` concept of the UI meta-model.

`OpeningPopup` corresponds to the code executed to open a Popup.

`OpeningDialog` corresponds to code executed to open a dialog. The dialog is already defined in the UI meta-model, and many openers exist.

C. Behavioral code migration approach

Because the GUI behavioral meta-model is linked to a UI meta-model, the first step to extract the behavioral code is the extraction of the UI model. This step is part of the GUI extraction and we follow the same approach as [3]. It consists of identifying the widgets, their attributes, and the DOM of the visual part of the application.

Then, it is possible to extract the GUI behavioral code model. To do so, we first extract the event handlers, and then extract the GUI behavioral code from the AST.

The extraction of event handlers is done into three sub-steps: **Identify event handler types**, **Detect handlers instances**, and **Attach handlers to widgets**. This extraction works on the GUI model and follows 3 steps to extract widgets and their attributes: identify, detect, and attach.

[5]FAST (generic AST): https://github.com/moosetechnology/FAST/

Identify event handler types: We identify all the possible event handlers that exist in the source application's framework. A handler is a class that executes the GUI behavioral code when an event occurs. Example of events handler types are: *click*, *change*, *hover*...

Detect handler instances: We determine where, in the source code, the event handlers are created (*i.e.* instantiations of the event handler types).

Attach handlers to widgets: We link the handler instances to their widget owners. From this sub-step, we know the interactions allowed by the application for each widget.

Once we have extracted the event handlers, it is possible to extract the GUI manipulation code. This step is divided into four sub-steps: **Build AST**, **Define patterns**, **Apply pattern**, and **Model transformation**.

Build AST: First, we build an AST of the code that will be executed. Note that the executed code can be spread over a group of methods or classes that must be parsed to obtain a complete AST of the GUI manipulations code.

Define patterns: Then, we manually define patterns that identify GUI manipulation code inside the AST model. Each GUI manipulation code can be detected by one or multiple patterns. Since the GUI manipulation code depends on the GUI framework, one must redefine patterns for each framework, but patterns are common to all applications that use the same framework.

Apply pattern: Then, we use a pattern matcher with the defined patterns on the ASTs. It provides the location of the GUI manipulation code in the source code.

Model transformation: Finally, for each detected GUI manipulation code, we apply model transformations. It consists in creating the behavioral entity associated to the pattern (*e.g.* `OpeningPopup` or `Navigating`) with its associations, and replace the old AST expressions with the new GUI behavioral entities.

Having presented our approach to migrate the GUI manipulations code as well as the behavioral meta-model used in our approach, in the following, we exemplify the application of this approach on a real system.

V. Concrete example

In the following, we present an implementation[6] of our migration approach. As the approach is split into two parts, we split the extraction into two parts: extractings events, Section V-A, and extracting the GUI manipulations code, Section V-B.

A. Extracting Events

We are now detailing the three sub-steps of the event extraction. Figure 4 presents a snippet of code that illustrates the creation of event handlers. The code consists of the creation of three widgets, line 1 a panel, line 2 a linkbutton, and line 5 an anonymous button (new Button(){...}).

[6]anonymous URL to the implementation on GitHub

```
1 Panel panel = new Panel();
2 LinkButton linkbutton = new LinkButton("Send");
3 linkbutton.addClickHandler(new ClickHandler() {
4   public void onClick(ClickEvent event) { ... }});
5 panel.add((new Button()).addClickHandler(new
      ClickHandler() { ... }));
```

Fig. 4: Creating event handlers in Java/GWT

Identify event handler types: In our context, the application is developed in Java with the GWT framework. The GWT documentation[7] defines the class `EventHandler` as the most abstract event handler type. Thus, the available event handlers in the source application are the subclasses of EventHandler.

In Figure 4 only the ClickHandler type is represented (lines 3 and 5).

Detect handlers instances: To detect handlers' instances, one needs to look for the invocations of the Java constructor of the events handler types. For instance, creating a *click* event is made by calling `new ClickHandler(...)`.

In Figure 4, there are two event handler creations, line 3 and 5, both identified by new ClickHandler.

Attach handlers to widgets: For each handler instance, our implementation extracts its widget owner. To do so, it looks for the receiver of the handler creation. The receiver of the handler creation might not be in the same statement as the handler creation. In such a case, the extraction performs static analysis to retrieve the correct widget owner. The owner can be declared in the same method, or in another method or class.

In Figure 4, the first click handler (line 3) is created inside the method addClickHandler sent by the variable linkbutton. And the variable linkbutton holds the linkbutton widget defined line 2. Thus, the event handler owner is the linkbutton widget. The second click handler (line 5) is created inside the method addClickHandler sent by the anonymous button. Thus, the event handler owner is the anonymous button.

B. Extracting GUI manipulation code

In the following, we detail an example in our context that extracts GUI manipulation code from the methods presented Figure 5. In this example, the method onClick() is called when the end user clicks on a button of the UI which, in turn, calls the method generateError(). In case the application has been launched in debug mode (line 5), a Popup is displayed with the message *"I am an error"* (line 7). Otherwise, the navigation to the page `APage` is performed (line 9). The extraction of the GUI manipulation code is divided into four sub-steps.

Build AST: From the identified event handler instances, it is possible to build an AST of the executed code. In our context, the handlers' instances are represented by anonymous classes (*e.g.* new ClicHandler() {...}).

The executed code might be spread over several methods or classes. To identify all the methods, our implementation first uses the Famix model [17] that identifies the call graph of the handlers instances' methods. For instance, in Figure 5, the

[7]http://www.gwtproject.org/javadoc/latest/

```
1  public void onClick(final ClickEvent event) {
2    this.generateError();
3  }
4  private void generateError() {
5    if(debugMode){
6      System.err.println("logging error");
7      EventPopup.displayError("I am an error");
8    } else {
9      Workspace.getPhaseManager().displayPhase(
         ConstantsPhase.APage());
10   }
11 }
```

Fig. 5: Example of GUI manipulations code

method generateError() is called by the method onClick() which is the method called by the event handler. Thus, our prototype builds the AST of both methods.

Define patterns: To detect GUI manipulation code inside an AST model, we defined manually for each GUI manipulation code one or multiple patterns. In the following, we present the 8 patterns we defined for the 6 GUI manipulations code. Table I presents the mapping of patterns to produced GUI manipulations code.

TABLE I: Mapping of pattern to GUI manipulations code

	AST pattern		GUI manipulations code
(1)	ErrBox	→	`OpeningPopup`
(2)	EventPopup	→	`OpeningPopup`
(3)	Window.alert(...)	→	`OpeningPopup`
(4)	Workspace.getPhaseManager() .displayPhase(...)	→	`Navigating`
(5)	*aDialog*.show()	→	`OpeningDialog`
(6)	*aWidget*	→	`AccessingDOM`
(7)	`AccessingDOM`.set*X*(...)	→	`SettingAttribute`
(8)	`AccessingDOM`.get*X*(...)	→	`GettingAttribute`

For `OpeningPopup`, we defined three pattern: (1) reference to `ErrBox`; (2) reference to `EventPopup`; or (3) invocation of `alert` method of the `Window` class (*i.e.* Window.alert(...)).

For `Navigating`, we defined one pattern that matches in the AST model Workspace.getPhaseManager().displayPhase(...).

For `OpeningDialog`, the pattern matches an invocation of `show` method on a variable that is a dialog. At this stage, we remind the reader that the widgets and the variables in which they are assigned were extracted during the GUI extraction.

For `AccessingDOM`, the pattern matches a reference to a variable that contains a widget already extracted during the GUI extraction.

For `GettingAttribute` and `SettingAttribute`, the patterns looks for already matched `AccessingDOM` GUI manipulation code that called respectively a getter or a setter. The name of the attribute is retrieved from the name of the getter or setter, *i.e.* setTitle() corresponds to a setter of the attribute *title*.

Apply pattern: Then, we use a pattern matcher with the defined patterns on the ASTs. In our example (Figure 5), our implementation identifies two GUI manipulations code. EventPopup.displayError(...) matches one of the `OpeningPopup` pattern, and Workspace.getPhaseManager().displayPhase(...) matches the `Navigating` pattern.

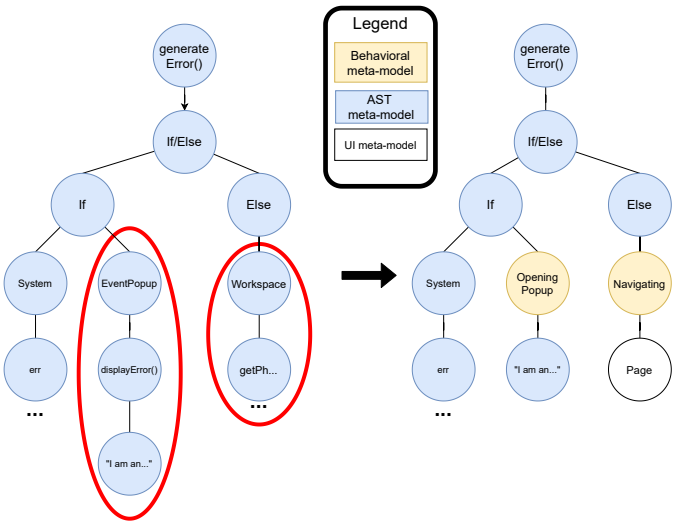

Fig. 6: Example of model transformation for Figure 5
left: Original AST ; right: Transformed AST

Model transformation: Finally, we perform model transformations on each AST. Figure 6 presents the model transformation performed for the method generateError() of Figure 5. The left-hand side presents a simplified version of the original method AST. The circled entities (Worspace, getPh..., EventPopup, displayError, and "I am...") are the entities found by the pattern matcher. The right-hand side presents a simplified version of the produced behavioral model. For instance, for the `OpeningPopup` (Figure 5, line 7), we replace EventPopup.displayError by a `OpeningPopup` entity. The string parameter is preserved during the transformation. The case of `Navigating` is more complex, as the parameter ConstantsPhase.APage() refers to a `Page` defined in the UI model, our approach also retrieves the page (in white in Figure 6).

VI. Generating

From the behavioral model, it is possible to generate the target code which is the last step of our approach. The generation is done by visiting the modified AST model and generating for each node its target language counterpart. It does not present major difficulties. Nevertheless, in the following, we present the different features we have implemented on the generator to produce natural code and help developers in the migration process.

Figure 7 and Figure 8 present respectively an example of Java code, and the TypeScript counterpart migrated using our approach. These examples illustrate five features we have developed to produce natural code.

Migration of Java to TypeScript: Additionally to the migration of the GUI manipulation code, our implementation migrates the rest of the code (*i.e.* variable declaration, control flow, *etc.*). Although it is not one of our main goals, it helps developers understand the target language[8], and it speeds

[8]Note that most of them are experts in GWT but novices in Angular

```
1  public void onClick(final ClickEvent event) {
2    String values = emailBox.getText();
3    if (values != null) {
4      values.split(",");
5      EventPopup.displayInfo("can access");
6    } else {
7      Workspace.getPhaseManager().displayPhase(
       ConstantsPhase.AnotherPage());
8    }
9  }
```

Fig. 7: Example of Java code

```
1  constructor(
2    protected _desktopService: DesktopService,
3    private _toastrService: ToastrService,) {
4  }
5
6  onClick() {
7    let values = (<any>this.input).nativeElement.
       value;
8    if (values != null) {
9      values.split(',');  // <ToReview> : Unknown
       invocation: split(...)
10     this._toastrService.success('can access');
11   } else {
12     openPage('AnotherPage');
13   }
14 }
```

Fig. 8: Example of TypeScript code migrated from Java code in Figure 7

up the migration process. For instance, Figure 7 line 2, the variable `values` is a string declared in Java. Our generator produced, Figure 8 line 7, `let values` which corresponds to the `values` variable declaration in Angular.

Add comments: Our generator adds comments with a tag *ToReview* at the end of each statement where part of the statement is not fully migrated. It helps the developers focus on problematic expressions. For instance, Figure 7 line 4, the `split(...)` method is not known by our tool, so, it is migrated as a TypeScript method invocation, Figure 8 line 9, and flagged the comment with `Unknown invocation`.

Follow target framework guidelines: The generated code should use the target framework features. Indeed, it is important to follow the target framework guideline to produce natural code. For example, in our context, we can use both JQuery or the Angular framework to access a DOM element. Whereas the code using JQuery would be more concise, we prefer to use Angular native features. Figure 8 line 7, `(<any>this.input).nativeElement` is an Angular DOM element access. Using JQuery, the code would be translated as `$("#input")`, which is more concise but does not use Angular features.

Another option would have been to use the Angular *data binding* feature. It allows one to access the value of a widget from TypeScript. However, this solution is more challenging because requires modifying the HTML source code.

Another example of an Angular feature supported by our

Fig. 9: Angular Event Binding feature

tool is the *event binding*. Figure 9 presents how event-binding feature is used. It consists of adding in the HTML source code the template statement (*e.g.* method) executed when an event is fired. Migrating without using the Angular feature would have resulted in calling the method addEventListener on the button field.

Allow API switching: API differences exist between the source GUI framework, the target GUI framework, and the UI meta-model. Our generator must take into account the differences to produce code. For example, the content of an input is represented by the attribute "text" in the UI meta-model. However, in Angular, it translates as "value". To handle such differences, we manually mapped each source UI concept to its target counterpart. In a concrete example, Figure 7 line 2, `emailBox.getText()` allows one to get the value of the input text `emailBox`. We manually map the GWT attribute "text" to the Angular attribute "value". So, Figure 8 line 7, our tool generated in Angular an access to the "value" property.

Initialize dependencies: To use GUI manipulation code, one needs to initialize their dependencies. For instance, the GUI manipulation code to navigate from one page to another needs the navigation service. To generate the code that initializes GUI manipulation code dependencies, we first manually map each GUI manipulation code to its required dependencies. Then, during the code generation step, our implementation generates the code to initialize the dependencies in the class constructor. In our context, it is the case for multiple GUI manipulations code elements. For instance, the `Navigating` and the `OpeningPopup` GUI manipulations code need to use three Angular services. Figure 8 lines 2 to 3, the generator automatically declared the `DesktopService` service to allow the navigation between pages and the `ToastrService` used by the OpeningPopup GUI manipulations code.

VII. Experiment

We evaluate our approach and its implementation on the migration of a real industrial application. First, Section VII-A, we present the case study. Then, Section VII-B, we discuss the evaluation metrics. Finally, Section VII-C, we present our results.

A. Industrial case study

This work is done in collaboration with an international industrial partner, Berger-Levrault. Berger-Levrault has developed several applications of different sizes in GWT. We performed our behavioral migration process on two of its applications. The first one is the most important of the company comprising more than 500 web pages, more than 50,000 widgets, and is maintained day-to-day by more than 40 full-time developers and engineers. The application includes more than 1 MLOCs, in 21,433 classes and 95,164 methods.

The second application is a *middle*-size application totaling 56 web pages comprising more than 4,000 widgets. It includes 200 KLOCs in 3,725 classes and 16,585 methods. Both applications are more than 10 years old.

B. Evaluation set-up

Our implementation runs without raising any errors for both applications. However, there is a lack of automatic tools to evaluate the correctness of the produced code. Thus, we performed a manual evaluation of the *middle*-size application that took us two full-time weeks (10 person/days). Note that performing the same evaluation for the most important application has been estimated by our industrial partner to 5 person/months.

Since, no approach on behavioral migration was found in the literature, we propose a new evaluation set-up, divided into two parts: check the structure and check the naturalness of the code.

For the **structure**, our solution checks that the event handlers are correctly detected, and migrated. It consists in the following three metrics already used in UI migration evaluation [2, 8, 18]:

- *The percentage of event handlers correctly detected*, *i.e.* the event handlers are detected regardless of whether their types are detected or attached to the correct widget. For example, the ondrag event handler type is not in our meta-model but our approach detects that an event handler exists and the owner widget has not been extracted during GUI extraction.
- *The percentage of event handlers types correctly detected*, *i.e.* a click handler in the original application corresponds to a click handler in the generated code.
- *The percentage of event handlers assigned to the correct widget*.

Because no tool performs such an evaluation, we rely on manual validation to check all these metrics. For the *percentage of event handlers correctly detected*, we looked at the source code of the application, file by file, and counted the number of created event handlers and compared it with the number of elements found by our tool. For the *percentage of event handlers types correctly detected* and the *percentage of event handlers assigned to the correct widget*, for each detected event handler, we manually checked, in the source code, that its type and owner were correctly extracted by our tool.

For the **naturalness of the code**, we want to check that the generated code respect the convention of code written by Angular developers. We used external tools that check the quality of the produced code. For instance, we applied SonarQube[9]; the TypeScript transpiler that reports badly written TypeScript code and potential problems (such as missing classes); and an Angular linter named *codelyzer*[10] that is less relevant because it only reports problems such as using spaces instead of tabs. To avoid bias in the results we used the default setting of each tool.

We also compare our approach to JSweet, a transpiler from Java to TypeScript. It claims to produce code that compiles and is "programmer-friendly". This is further discussed in Section VIII-B.

C. Results

First, after performing our approach on the company's application, we generate an Angular application that compiles and runs without raising any errors.

We now check that the event handlers of the original application are well detected, associated with the correct `Event` concept of our behavioral meta-model, and linked to the right exported widget.

TABLE II: Result of manual event handler extraction check

event handler detected	event handler type detected	event handler correctly assigned
100% (232)	98% (228)	95% (221)

In parentheses: number of event handlers

Table II summarizes the results for the **structure** check. Our manual evaluation reported 232 event handlers created in the Java code.

Our prototype detected 100% of the created event handlers. Among them, it detected 98% of the event handlers types. The 4 event handler types not detected correspond to handlers for events created by the developers. This problem is further discussed Section VIII-D. 95% of the event handlers are assigned to the correct widget. Four out of the 11 missing widgets were the preceding unidentified ones, so our implementation did not try to assign them to a widget. For the other 7, their owners were not present in the UI model. So, improving the GUI extraction, which is out of the scope of this paper, would help here. To summarize, our approach detected 232 event handlers comprising 214 *click handlers*, 5 *change handlers*, 1 *hover handler*, and 1 *out handler*. The remaining 11 event handlers were not correctly extracted.

TABLE III: natural code

Approach	SonarQube (blocker/major/minor)	TypeScript transpiler	Linter (codelyser)
Our approach	520 (0/98/422)	130	367
JSweet	986 (3/566/417)	6,539	21,344

Table III summarizes the results for the **naturalness of code**. SonarQube categories the errors by severities[11]. Blockers have an important impact on the system and are likely to happen; majors have a limited impact and are likely to happen; minors have a limited impact and are unlikely to happen.

For SonarQube, our migrated code has 520 errors. It corresponds to 98 "major" problems and 422 "minor" ones.

[9]SonarQube is a well known open-source tool that analyzes code quality and security

[10]codelyzer: http://codelyzer.com/

[11]https://docs.sonarqube.org/display/SONARqube71/Rules+-+types+and+severities

More than half of the major problems (64/98) are "deprecated HTML attribute usage" and "missing table header". The first one should be avoided to ensure web browsers compatibility. The second one creates problems with web accessibility, for instance, assistive technologies, such as screen readers, use table headers to provide context to users. For the JSweet migrated version, SonarQube reports 986 errors with 3 "blocker problems", 566 "major", and 417 "minor". More than half of the major problems (492/566) are TypeScript problems with multiline blocks and control flow that might raise issues with non-empty statements. This analysis shows that our approach produces a code of better quality for SonarQube than the JSweet approach.

The TypeScript transpiler provides information about the number of unknown class usage, *i.e.* references to classes that do not exist. For our approach, the TypeScript transpiler reports 130 missing classes. They are helper classes and classes that represent data (DTO [19]). For the JSweet exported version, it reports 6,539 missing classes. They are helper classes, classes that represent data, and widget and behavioral classes (Button, ClickHandler, *etc.*). The transpiler does not report critical problems for both approaches. This analysis shows that our approach generates far fewer problems to fix.

The lint, *codelizer*, reports only minor problems due to code formatting. It reports 367 problems for our approach and 21,344 problems for the JSweet exported version. It also reports missing bracket for *if* and *for* statements for the JSweet exported version. Although brackets are not always mandatory, missing brackets might introduce bugs in the application. This analysis shows that our approach has a lot fewer problems than the JSweet approach.

VIII. Discussion

Section VIII-A discusses how the company environment might have eased our work. Section VIII-B presents the differences between our approach and JSweet. Section VIII-C discusses the genericity of our approach. Section VIII-D presents the custom events, that are events not present in our meta-model. Finally, Section VIII-E highlights the features in the target framework that eased migration compared to other migration projects.

A. Company environment

Although the results are encouraging, we only experimented our tool on two applications of Berger-Levrault. Berger-Levrault has a development guideline that must be followed by developers. Thus, the applications we are migrating do not use all the Java features (*i.e.*, lambda expression, dependency injections, *etc.*). As a result, our implementation did not need to handle such features. This might have eased the extraction of the GUI behavioral code.

B. Validation with JSweet

Since we did not find any other tool that migrates behavioral code, we used JSweet as another Java to TypeScript migration tool. However, JSweet **does not** migrate from GWT to Angular, and so we do not have the same goal. Whereas it gives us a baseline to compare to, another study with a tool that migrates GUI behavioral code would be preferable. However, there is no other tool in the literature that performs such a migration.

C. Genericity of the approach

Our approach relies on applying pattern matching on AST. We acknowledge that this kind of approach is complex to reproduce.

To reduce this problem, we executed our approach on different applications of Berger-Levrault. For instance, we experimented it on the major application of the company that is a human resources software. We report 3,469 event handler instances in the 519 pages. Whereas it does not validate that our tool extracts all the handlers, it shows that the same implementation can be used for different applications.

It would also be interesting to validate our approach against applications using another source framework. To do so, one needs an AST model and a parser for the source language. Then, it is possible to perform the approach as described Section IV-C. The major challenge is then to define patterns for each GUI manipulation code. Automatic recognition of GUI manipulation code patterns will be part of our future work.

In conclusion, the time needed to reproduce this work will depend on the already existing tools. In our context, we use the Moose platform [20] that comes with the Famix meta-model and an AST meta-model, and an extractor for the Java programming language. Then, manual work requiring good knowledge of the source GUI framework is necessary to extract the GUI manipulation code patterns.

D. Custom events

We identified 5 event types used in the applications of our industrial partner. Extracting and generating these events enable good results with our current implementation. However, as described Section III-B, more events exists. For instance, the Firefox developer page lists 112 common events and 102 uncommon events. Additionally, developers can create their own events. In this situation, it is not possible to add all possible events into a meta-model beforehand. To improve the genericity of our approach, one can easily add new concepts as a kind of Event when they are encountered.

E. Target language eases the work

Our migration case study is from GWT to Angular. That is to say from Java to TypeScript. It is important to note that moving from Java to TypeScript is eased by multiple factors. First, there is no switch of paradigm. Indeed, both are object-oriented languages[12]. Second, during migration, a common problem comes with type mismatch [21], but TypeScript is a dynamically typed language and several Java types are identical in TypeScript or merged into one. For instance, the

[12]https://rachelappel.com/2015/01/02/write-object-oriented-javascript-with-typescript/

Java `double`, `int`, `short`... are all merged into the TypeScript `number` type.

IX. Conclusion and Future work

In this paper, we propose a definition of GUI behavioral code divided into two parts. We designed a meta-model that allows one to represent the GUI behavioral code and an approach to perform a migration using this meta-model. In the context of an industrial partnership, we implemented our approach in a prototype that performs the migration of GWT code to TypeScript for an Angular application. Then, we validated our approach and implementation on a real industrial case study. To ensure the genericity of our approach, we also tested it on another company's project. We evaluated that our tool produces natural code and correctly migrates 95% of the event handlers.

Our approach has good results for the migration of the GUI behavioral code. To make the generated code executable, one next step is to investigate the problem of missing class usage generated by our tool and the other existing approaches.

We also would like to better study the genericity of our approach by experimenting with other GUI frameworks. To do so, we plan to reproduce the experiment with another GUI framework and then work on the automatic GUI manipulations code pattern recognition.

References

[1] F. Fleurey, E. Breton, B. Baudry, A. Nicolas, and J.-M. Jezéquel, "Model-Driven Engineering for Software Migration in a Large Industrial Context," in *Model Driven Engineering Languages and Systems*, vol. 4735. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 482–497.

[2] O. Sánchez Ramón, J. Sánchez Cuadrado, and J. García Molina, "Model-driven reverse engineering of legacy graphical user interfaces," *Automated Software Engineering*, vol. 21, no. 2, pp. 147–186, 2014.

[3] B. Verhaeghe, A. Etien, N. Anquetil, A. Seriai, L. Deruelle, S. Ducasse, and M. Derras, "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, 2019.

[4] M. P. Robillard and K. Kutschera, "Lessons learned while migrating from swing to javafx," *IEEE Software*, vol. 37, no. 3, pp. 78–85, 2019.

[5] T. C. Terwilliger, N. Sauter, and P. D. Adams, "Automatic Fortran to C++ conversion with FABLE," *Source Code for Biology and Medicine*, vol. 7, no. 5, May 2012. [Online]. Available: https://scfbm.biomedcentral.com/articles/10.1186/1751-0473-7-5

[6] J. Martin and H. A. Muller, "C to java migration experiences," in *Proceedings of the Sixth European Conference on Software Maintenance and Reengineering*. IEEE, 2002, pp. 143–153.

[7] J. Brant, D. Roberts, B. Plendl, and J. Prince, "Extreme maintenance: Transforming Delphi into C#," in *ICSM'10*, 2010.

[8] T. Hayakawa, S. Hasegawa, S. Yoshika, and T. Hikita, "Maintaining web applications by translating among different RIA technologies," *GSTF Journal on Computing*, p. 7, 2012.

[9] K. Garcés, R. Casallas, C. Álvarez, E. Sandoval, A. Salamanca, F. Viera, F. Melo, and J. M. Soto, "White-box modernization of legacy applications: The oracle forms case study," *Computer Standards & Interfaces*, pp. 110–122, Oct. 2017.

[10] H. Samir, A. Kamel, and E. Stroulia, "Swing2script: Migration of Java-Swing applications to Ajax Web applications," in *14th Working Conference on Reverse Engineering (WCRE 2007)*, 2007.

[11] H. M. Sneed and C. Verhoef, "Cost-driven software migration: An experience report," *Journal of Software: Evolution and Process*, p. e2236, 2020.

[12] T. Tonelli *et al.*, "Swing to swt and back: Patterns for api migration by wrapping," in *2010 IEEE International Conference on Software Maintenance*. IEEE, 2010, pp. 1–10.

[13] A. J. Malton, "The software migration barbell," in *ASERC Workshop on Software Architecture*. Citeseer, 2001.

[14] K. Aggarwal, M. Salameh, and A. Hindle, "Using machine translation for converting python 2 to python 3 code," PeerJ PrePrints, Tech. Rep., 2015.

[15] C. Teyton, J.-R. Falleri, and X. Blanc, "Automatic discovery of function mappings between similar libraries," in *2013 20th Working Conference on Reverse Engineering (WCRE)*. IEEE, 2013, pp. 192–201.

[16] M. Trudel, C. A. Furia, M. Nordio, B. Meyer, and M. Oriol, "C to oo translation: Beyond the easy stuff," in *2012 19th Working Conference on Reverse Engineering*. IEEE, 2012, pp. 19–28.

[17] S. Ducasse, N. Anquetil, U. Bhatti, A. Cavalcante Hora, J. Laval, and T. Girba, "MSE and FAMIX 3.0: an Interexchange Format and Source Code Model Family," RMod – INRIA Lille-Nord Europe, Tech. Rep., 2011.

[18] M. E. Joorabchi and A. Mesbah, "Reverse engineering iOS mobile applications," in *2012 19th Working Conference on Reverse Engineering*. IEEE, 2012, pp. 177–186.

[19] P. B. Monday, "Implementing the data transfer object pattern," in *Web Services Patterns: Java Platform Edition*. Springer, 2003, pp. 279–295.

[20] N. Anquetil, A. Etien, M. H. Houekpetodji, B. Verhaeghe, S. Ducasse, C. Toullec, F. Djareddir, J. Sudich, and M. Derras, "Modular moose: A new generation of software reengineering platform," in *International Conference on Software and Systems Reuse, ICSR2020*, Dec. 2020.

[21] A. A. Terekhov and C. Verhoef, "The realities of language conversions," *IEEE Software*, Nov. 2000.