GWT to Angular: Migration of graphic applications through Models

Benoît Verhaeghe[1,2], Anne Etien[1],
Nicolas Anquetil[1], Stéphane Ducasse[1]
[1]Université de Lille, CNRS, Inria,
Centrale Lille, UMR 9189 – CRIStAL, France

Abderrahmane Seriai[2], Laurent Deruelle[2],
Mustapha Derras[2]
[2]Berger-Levrault, France

***Abstract*—In the context of the graphical application evolution, it happens that the application must switch of implementation language. This evolution should be done by companies which want to stay *Up-to-Date* with the latest technology. It is also an important commercial argument to keep and attract clients. Berger-Levrault is a major IT company which owns the biggest GWT application in the world. The company needs to switch the application implementation of its software from GWT to Angular. Rewriting graphical application is complex and can lead to errors. Furthermore, the transformation from an implementation to another implies modifications of the application structure. Moreover, the migration must not impact the clients of Berger-Levrault. And so the visual aspects of the migrated application have to be the same as the original one. The company has estimated the duration of the migration to X. To reduce the amont of time needed, we've created a tool that semi-automatically does the migration of the application. To achieve this tool, we designed a metamodel to represent a graphical user interface (GUI[1] metamodel). Then we used a three parts application migration strategy and the tools to apply such strategy. Firstly, we instantiated a metamodel from the source code. Then, transform the resulted model to a model that respects GUI metamodel. Finally, we exported the *"graphic"* model to the target code.**

***Index Terms*—Graphical User Interfaces, Industrial case study, Java, Angular, GWT, Migration, Model-Driven Engineering, MDE**

I. Introduction

Desc de BL, et de leurs besoins Berger-Levrault is a major IT company. It uses the framework GWT to develop its graphics applications. GWT doesn't evolve almost any more with only one major release since 2015 whereas there was six major releases between 2010 and 2013. So, Berger-Levrault has decided to migrate all its application from Java using GWT to the Angular technology. The main aim is to avoid to be stuck in an old technology.

Desc des applications de BL (général) et que leurs tailles et tmps de vie est un pb Software of Berger-Levrault are long life applications (more than ten years), they have more than X MLOC and represent X web pages. The lifespan of the program implies that no one in the company has a *"perfect"* knowledge of the applications. This lack of information leads to difficulties for the migration. Indeed, migrating unused code and *hack* that were important in the source language create errors in the target one. This is why such migration is time-consuming and error-prone. Berger-Levrault has estimated the duration of the migration to X.

Definition du problème. Solution générique de migration d'interface The migration of an application can be done by rewriting all the application in the target language. But, in the case of a migration in a company, it must respect many criteria. Those criteria are detailed section II-A. Moreover, re-writing separately the applications can be done but it is time-consuming. The main problem is to define a strategy to pilot the migration.

Présente solutions autour de la notre Benoit ▶*TODO*◀

what our solution is ? un outil reutilisable/une stratégie de migration We present a strategy that will let the developers to migrate their graphical application. This strategy include a GUI metamodel, a strategy to generate this model, a way to generate the target application whatever the target framework Architecture is. The tool detects well X% of the widget for a web page. After the migration, X% widgets are well placed and X% are visually identical.

The main contributions of our work are:

- a modular tool that can be reused for other graphics application migration.
- a metamodel to represent a GUI application.
- a migration strategy to migrate graphics application.

In Section II, we give background information on our problem. Then, in Section III, we describe the solution we've proposed. Section IV presents the results and threats to the validity of our works. Finally, we present the related works and conclude in Section V and VII respectively.

II. Problem Description

Prez context In the context of the evolution of the applications of Berger-Levrault from GWT-based program to Angular, the company has estimated X days of development to complete the migration process. This is due to the X MLOC that composed the software they have to migrate. A aim of our work with Berger-Levrault is to provide a strategy that support the migration process to reduce the necessary time of the program transformation. Our work was tested on the showroom application of Berger-Levrault. This application is used only by the company's developers as an

[1]Graphical User Interface

example application of the usage of the widget available for the development.

A. Constraints

The solution we propose must respect some criteria:

- *Source independence*. The solution should be easy to adapt for all projects written in GWT using Java and for applications not written in Java but describing a GUI. This constraint must be respected to be able to use the solution for different project. In the context of Berger-Levrault, it will allow the developers to apply the solution to the different piece of software they have to migrate.
- *Target independence*. It has to be easy to change the target language without restructuring most of the implementation of the solution. This constraint guarantee that the solution can be used whatever the architecture of the target language.
  In our case study, we try to migrate GWT-based application to Angular. Angular got six different major versions since 2016. Thanks to the target independence, the strategy is still valid for the migration to another target language or a different version of the original program source code language.
- *Modularity*. The migration would be split into many little stages. This offer the possibility to easily extend a step or replace one step by another one without introducing instability. It is essential for a company that need fine-grained control on the migration process. With the Modularity, the companies can customize every part of the solution to respect specific constraint they own.
- *Preserving Architecture*. After the migration, we should find the same architecture between the different component of the GUI (*e.g.* a button which belongs to a panel in the source application still belongs to the same panel in the target application). This constraint has two interests. It will facilitated the work of the developers of the application because, it should be easier for them to understand an application with the same architecture Benoit ►*ref?*◄ .
- *Visual Layout-preserving*. It should not have visual differences between the source application and the target one. This is especially important for commercial software. Indeed, the user of the application must not be disturb by the migration. So, preserving the layout will lead to no interface impact for the final user.

The development team has to continue the maintenance of the source application. This is a constraint inherent to industrial companies.

B. GWT and Angular Comparison

The source language and the target language can have two different architectures. Their difference can be syntactical, semantical and also architectural. In the case of the migration of the application from Java using GWT to Angular, the .java file will be separated into multiple Angular files.

With GWT, the source program is composed with one class per web page. So, it is possible to have only one file that represents one web page. In this class, we find the widgets' architecture and organization. It contains layout information such as *VerticalPanel*, *HorizontalPanel*, *FlexPanel*, etc. and style information by using setter call to widgets to change the height, width, color, etc. The .java file contains also the business code link to the widgets. GWT demand one configuration file to declare all the web page's Java class to determine the URL path to access to the web pages.

TABLE I
COMPARISON OF GWT ARCHITECTURE AND ANGULAR ONE

	Java using GWT	Angular
number of configuration file	one configuration file	four configuration files plus two files per subprojects
web page	one java class	one Typescript file plus one HTML file
style for a web page	include in java class	one optional CSS file

Benoit ►*citer tab*◄

With Angular, those pieces of information are dispatched in multiple files. First of all, Angular is a Single Page framework based on Component, which means there will be only one web page and we will navigate into this web page. A component is a sub-part of a web page. It can be a complex widget like a calendar, a part of a webpage like a sidebar or a complete webpage. A component can contain other components, so the component representing a webpage can contain the sidebar component, and this last contains the calendar. Only the reached web page content is loaded. At the root of the Angular project, it has the main configuration files. Those files explicit the frameworks that will be used in the application. Then for each web page, we will find the following structure. A folder that contains at least an HTML file, a TypeScript file. It can also contain a CSS file, and some configuration file. The configuration files contain specific needed of the webpage. It can be compared to the *import* call in Java. The CSS file contains the specific style for this webpage. The TypeScript file contains the behavioral business logic of the webpage. The HTML file contains the architecture of the webpage and the layout information. The layout information is expressed by the class attribute defined in a general CSS file at the root of the project.

C. Migration strategy solutions

There are different solutions to migrate an application. All the solutions have to conclude with the respect of the constraints defined section II-A

Benoit ►*Justifier les Migrations*◄

- *Manual Migration*. This strategy is the re-development of the all application without using semi-automatic migration tool. This strategy offers the possibility to easily fix some error of the old application and to redesign the application following architecture principle of the target

language [3] **Benoit** ▶*comment bien faire une citation ? (intro sec 5)*◀ .

- *Rule Engine*. Using a rule engine to migrate the totality or part of an application has already been applied in some project [1, 2, 6]. We have to define and create transformation rules to use this solution. The rules takes as input some piece of source code and gives as output the migrated code. It is possible that the migration is not completed by the strategy, so the developers will have to end the process of migration by some manual work. Using a rule engine can be efficient but it implies to not be source independent nor target independent.
- *Model-Driven Engineering (MDE)*. Model-Driven Engineering implies the development of metamodels to do the migration. This strategy follows all the constraint we defined. It is also possible to an automatic or a semi-automatic migration with MDE strategy. This last implies to do some manual work to complete the migration processus.

Because we need to conform our solution to the constraint of this kind of migration and of the company, we developed a migration tool that uses Model-Driven Engineering strategy.

III. Proposed Solution

context solution proposé We designed and implemented a migration strategy based on Model-Driven Engineering. The migration strategy is divided into three steps, the creation of a model of the source code, the transformation of this model to a GUI model, finally the generation of the target code from the GUI model.

explication des sous parties In the following, section III-A first presents the migration process we developed using model-driven engineering, section III-B exposes the metamodel we designed to represent a GUI, section III-C describes the implementation of the metamodel, how instantiate it and explains the process to generate the target code from the metamodel.

A. Migration Process

Benoit ▶*TODO*◀

B. Metamodel

In order to represent a GUI, we designed a metamodel presented Figure 1. In the following, we present the different entities of the metamodel.

The **Phase** represents the main container of a user interface page. It can be a *Window* for Desktop application, a web page or a tab in some specific case. For example, if a web site is composed by only one web page, and this web page has multiple tab. To represent the website as the content of the different tabs, the Phases will represent a tab. A Phase can contains multiple Business Page. It can also be called by any Widget with a *Call Phase Action*. When a Phase is called, the interface changes to display the phase. In desktop application, the current Window interface changes from the previous Phase to the called one or a new Window with the interface owned by the called Phase is opened. In web application, it can be a new

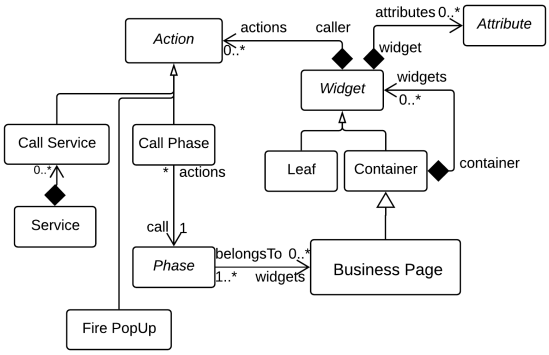

Fig. 1. GUI Application Metamodel

web page, the changement of active tab, the transformation of the current web page.

The **Widgets** are the different interface components and layout components. There are two types of Widgets. The **Leaf** is a widget which does not contain another widget such as some text in the interface. The **Containers** can contain other widget. It could be for organization purposed like separating widgets which belongs to a category and widgets which belongs to another category.

The **Business Pages** blabla **Benoit** ▶*Très BL specific... Il faudrait supprimer Page Métier*◀ .

The **Attribute** represents informations that belongs to a Widget and can change its visual aspect or its behavior. The common attributes are the height and the width to define precisely the dimension of a widget. There are also attributes to contains attributes such as the text contained by a widget. For example, a widget which represent a button can have an attribute *text* to explicit the text of the button. An attribute can change the behavior, it could be the case of an attribute *enable*. A button with the attribute *enable* positioned at *false* represent a button we can not click. Finally, the widgets can have an attribute that will impact the visual of the application. This attribute defines the layout of its children and potentially its own dimension to respect the layout.

The **Actions** are own by the Widgets. They are actions that can be run in a Graphic Interface. **Call Service** represents a call to a distant service such Internet. **Fire PopUp** is the action that display a PopUp in the screen. The PopUp can not be considered as a Widget, it is not present in the GUI, it only appears and disappears.

The **Service** is the reference to the distant feature the application can call from its GUI. In a web context, it can be the server side of the application.

C. Migration implementation

IV. Discussion

Critères d'évaluation

- évaluation avec les développeurs de la solution qu'on leur propose

- Nombre de phase correctement détecté
  - Nombre de widget correctement détecté (on peut compter à la main sur 20 pages web) ou via comparaison de DOM
  - Nombre de widget correctement placé (dans l'architecture sûrement pas 100% mais pas mal)
- Nombre de widget visuellement correctement placé (notion d'attribut, de panel).. Ici on aura de mauvais résultat qui proviennent de l'analyse statique et non dynamique (qui est une contrainte forte pour nous)

V. Related Works

To extract information from the application, it is possible to use a dynamic solution or static one.

Samir *et al.* [4] proposed a dynamic solution to migrate Swing application to Ajax. Their tool runs an instance of the source application in a browser. Then, they are able to detect the widget displayed by the interface and the different actions available. The authors decided to use a dynamic solution to explore the interface of the application. Their contributions are a method and a middleware toolkit for re-architecting Swing application to Ajax one.

Sánchez Ramón *et al.* [5] developed a static solution to reverse engineer RAD (Rapid Application Development) based graphic user interface (GUI). The authors created different metamodel to represent a GUI. The authors used those to metamodels to create a chain of transformation between them which lead to the transformation of the source application the target one. Although we can't reapply the strategy of GUI extraction to our project, because RAD based application is too *simple* compared to GWT based application, the different metamodels and the chain of transformation inspired our work.

VI. Future Works

VII. Conclusion

In this paper, we exposed a solution to the problem of visual preservation and respect of the target architecture during the migration of an application. Our work has shown encouraging results with the case study of Berger-Levrault. We migrated complex and simple web pages from the applications of Berger-Levrault. The tool currently migrates correctly X% of a web page. To do so, we used a strategy based on a metamodel. We presented the strategy and its implementation in Pharo. Our solution allows us to migrate a web page following the target architecture and with a correct widget hierarchy. The visual aspect of the rendered application could differ from the original one because of the layout applied to the first one. This work can be used as a support to improve the migration of Graphical User Interface. The migration of the business code of the original application is a feature that can help developers. It would be also interesting to improve the way the layout is represented or to define which kind tool, on top of the model, can help the developers with the migration or their daily development.

Acknowledgements

This work was supported by Ministry of Higher Education and Research, Nord-Pas de Calais Regional Council, CPER Nord-Pas de Calais/FEDER DATA Advanced data science and technologies 2015-2020.

References

[1] John Brant, Don Roberts, Bill Plendl, and Jeff Prince. Extreme maintenance: Transforming delphi into c. In *Software Maintenance (ICSM), 2010 IEEE International Conference on*, pages 1–8. IEEE, 2010.

[2] S. I. Feldman, D. M. Gay, M. W. Maimone, and N. L. Schryer. Availability of F2c—a Fortran to C Converter. *SIGPLAN Fortran Forum*, 10(2):14–15, July 1991. ISSN 1061-7264. doi: 10.1145/122006.122007. URL http://doi.acm.org/10.1145/122006.122007.

[3] Franck Fleurey, Erwan Breton, Benoit Baudry, Alain Nicolas, and Jean-Marc Jézéquel. Model-driven engineering for software migration in a large industrial context. In *International Conference on Model Driven Engineering Languages and Systems*, pages 482–497. Springer, 2007.

[4] Hani Samir, Eleni Stroulia, and Amr Kamel. Swing2script: Migration of java-swing applications to ajax web applications. In *Reverse Engineering, 2007. WCRE 2007. 14th Working Conference on*, pages 179–188. IEEE, 2007.

[5] Óscar Sánchez Ramón, Jesús Sánchez Cuadrado, and Jesús García Molina. Model-driven reverse engineering of legacy graphical user interfaces. In *Proceedings of the IEEE/ACM international conference on Automated software engineering*, pages 147–186. ACM, 2014.

[6] Thomas Charles Terwilliger, Nicholas Sauter, and Paul D Adams. Automatic Fortran to C++ conversion with FABLE. *Source Code for Biology and Medicine*, 7(5), May 2012. doi: 10.1186/1751-0473-7-5.