Diogo Manuel Gonçalves Romão

Bachelor in Computer Science

Migration from Legacy to Reactive Applications in OutSystems

Dissertation submitted in partial fulfillment
of the requirements for the degree of

Master of Science in
Computer Science and Engineering

Adviser: Miguel Goulão, Associate Professor,
NOVA University of Lisbon

Co-adviser: Carlos Xavier, Lead Software Engineer,
OutSystems

Examination Committee

Chair: João Moura Pires, Associate Professor, FCT-NOVA
Rapporteur: Fernando Brito e Abreu, Associate Professor, ISCTE-IUL
Member: Miguel Goulão, Associate Professor, FCT-NOVA

FACULDADE DE
CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

March, 2021

Migration from Legacy to Reactive Applications in OutSystems

Copyright © Diogo Manuel Gonçalves Romão, Faculty of Sciences and Technology, NOVA University Lisbon.
The Faculty of Sciences and Technology and the NOVA University Lisbon have the right, perpetual and without geographical boundaries, to file and publish this dissertation through printed copies reproduced on paper or on digital form, or by any other means known or that may be invented, and to disseminate through scientific repositories and admit its copying and distribution for non-commercial, educational or research purposes, as long as credit is given to the author and editor.

This document was created using the (pdf)LaTeX processor, based on the "novathesis" template[1], developed at the Dep. Informática of FCT-NOVA [2].
[1] https://github.com/joaomlourenco/novathesis [2] http://www.di.fct.unl.pt

Acknowledgements

To begin with, I would like to thank Faculdade de Ciências e Tecnologia from NOVA University and the Informatics Department. Throughout the past five years, I had the opportunity of receiving the necessary knowledge and skills to turn into the professional I am today. Also, I would like to thank OutSystems for giving me the opportunity to work on a challenging, yet rewarding project, as well as acknowledging the work presented.

Essential to this dissertation's success, Carlos Xavier and Miguel Goulão deserve a special appreciation. As advisers, your mentoring and commitment not only helped but motivated me to always give my best and go beyond the goals initially set.

Thank you to everyone who was a part of my experience during this dissertation, with a greater emphasis on the App Runtime and Target App Fit teams. Since the beginning, you made me feel welcome and your willingness to help, combined with an incredible internal culture is something that I will always take with me.

To my friends and colleagues who spent the last five years with me, thank you for being there. Regardless of the situations, we always pulled through, allowing us to learn as well as having fun.

Maybe most importantly, I would like to thank my close friends and family. The love and support you showed me gave me the strength to be who I wanted and inspired me to never quit and continuously outperform myself. To my parents, for always giving me the necessary conditions and opportunities, a special thank you. The lessons and examples I received all my life led to this moment and my success is, and always will be, because of you.

Finally, I would like to thank Inês for the unconditional support and always believing in me, you are amazing.

“The world as we have created it is a process of our thinking. It cannot be changed without changing our thinking.”

- Albert Einstein

Abstract

A legacy system is an information system that significantly resists evolution. Through a migration, these systems can be moved to a more modernized environment without having to be redeveloped. OutSystems is a software company with a platform to develop and maintain applications using abstraction to increase productivity.

In October 2019, OutSystems launched a new paradigm to allow developers to build reactive web applications. Because of this, the applications implemented in the old web paradigm turned into legacy systems. The OutSystems' approach to this problem was a manual migration. However, it discards a considerable part of the effort previously made on the legacy system. A well-founded case study took place and allowed us to classify the UI as the most prioritized feature, but coincidently, the major bottleneck in migrations.

So, this project had the following objectives: (1) The design and implementation of an automatic migration approach capable of converting UI elements to accelerate the manual migration; (2) The integration of the developed tool in the OutSystems platform.

To transform the OutSystems paradigm's elements, model-driven transformation rules must be set to receive the source UI elements and produce the target equivalent implementation in the new paradigm (each according to their model). However, the transformations may not be straightforward, and a set of elements may need to be migrated to a different implementation due to Reactive Web's best practices. Via the creation and search of UI patterns, it is possible to make special transformations for such scenarios.

As a result, a migration approach was developed, allowing for the migration of UI (and other) elements. To complement this objective, the developed tool was integrated into the OutSystems platform with an easy to use interaction. Performance and usability tests proved the necessity and impact the final result had on the migration problem.

This dissertation's objectives were fully met and even exceeded, accelerating the manual migration by providing an automatic UI conversion. This provided a quality increase in the existing process and results, giving OutSystems and its users the possibility of evolving their applications with considerable less effort and investment.

Keywords: Legacy Systems, Migration, Model-Driven Engineering, Model Transformations, Pattern Search, UI, OutSystems, OutSystems Reactive Web.

Resumo

Um sistema legado é um sistema de informação que resiste à evolução. Através de uma migração, estes sistemas podem ser movidos para um ambiente modernizado sem necessitar de re-implementação. A OutSystems é uma empresa de *software* com uma plataforma para desenvolver e manter aplicações usando abstracção para aumentar a produtividade.

Em Outubro de 2019, a OutSystems lançou um novo paradigma para desenvolver aplicações *reactive web*. Assim, as aplicações implementadas no antigo paradigma *web* tornaram-se sistemas legados. A abordagem da OutSystems ao problema foi uma migração manual, no entanto, esta abordagem desconsidera uma parte significativa do investimento feito no sistema legado. Uma análise permitiu classificar a *UI* como a característica mais priorizada, mas também como o maior obstáculo em migrações.

Assim, este projecto tem como objectivos: (1) O desenho e implementação de uma migração automática capaz de converter os elementos de *UI* para acelerar a migração manual; (2) A integração da ferramenta desenvolvida na plataforma da OutSystems.

Para transformar os elementos dos paradigmas OutSystems, transformações de modelos têm de ser definidas para receber os elementos *UI* e produzir a implementação equivalente no novo paradigma (de acordo com o seu modelo). No entanto, as transformações podem não ser lineares, e um conjunto de elementos pode necessitar de uma migração para uma implementação diferente devido ao *Reactive Web*. Com a definição e procura de padrões de *UI*, é possível fazer transformações especiais para esses cenários.

Como resultado, a migração foi desenvolvida, permitindo a conversão de elementos de *UI* (e não só). Para complementar, a ferramenta desenvolvida foi integrada na plataforma da OutSystems com uma interacção de fácil uso. Testes de desempenho e usabilidade provaram a necessidade e impacto da ferramenta no contexto da migração manual.

Os objectivos desta dissertação foram completados na totalidade, acelerando a migração manual com a automação da migração de *UI*. Isto traz um aumento da qualidade no processo existente e nos seus resultados, dando à OutSystems e aos seus utilizadores a possibilidade de evoluírem as suas aplicações com um esforço e investimento menores.

Palavras-chave: Sistema Legado, Migração, Engenharia de Modelos, Transformações de Modelos, Procura de Padrões, UI, OutSystems, OutSystems *Reactive Web*.

Contents

List of Figures xvii

List of Tables xix

Acronyms xxi

1 Introduction 1

1.1 Context 1
1.2 Motivation 2
1.3 Objectives 3
1.4 Key Contributions 3
1.5 Structure 4

2 Background 5

2.1 OutSystems Overview 5
2.1.1 OutSystems Architecture 5
2.1.2 OutSystems Development 8
2.1.3 OutSystems Application Development Paradigms 9
2.2 Legacy Systems Overview 12
2.3 Migration Overview 12
2.3.1 Phases of a Migration 13
2.3.2 Migration compared to other modernization techniques 13
2.3.3 Migration approaches 14
2.4 Model-Driven Engineering 15
2.4.1 Modeling 15
2.4.2 Modeling language 16
2.4.3 Model Transformations 17
2.5 Pattern Recognition 18
2.5.1 Pattern Search 19
2.5.2 Graph Pattern Search 19
2.5.3 Tree Pattern Search 20
2.6 Communicated Information 20
2.6.1 Metrics 20

2.6.2 Logging 21
2.6.3 Traces 21

3 Related Work 23
3.1 OutSystems Migration 23
3.1.1 Previous Work 23
3.1.2 Manual Migration Initiative 24
3.1.3 Automatic Migration 26
3.2 Migration Approaches 27
3.2.1 SOA (Service Oriented Architecture) Migration Approaches 28
3.2.2 Cloud Migration Approaches 29
3.3 Model-Driven Engineering 29
3.3.1 Modeling languages 30
3.3.2 Modeling in OutSystems 31
3.3.3 Model Transformations 32
3.3.4 Migration model-based Approach 33
3.4 Search Alghorithms 33
3.4.1 Graph Search Alghorithms 34
3.4.2 Tree Search Alghorithms 35
3.5 Discussion 35

4 Case Study and Initial Considerations 39
4.1 Requirements Gathering 39
4.1.1 Platform Analysis and Community Interest 39
4.1.2 Interviews 40
4.1.3 Summary 42
4.2 Migration Approaches 42
4.2.1 Elementary Migration 43
4.2.2 Pattern Driven Migration 44
4.2.3 Mixed Approach 45
4.3 Stakeholders panel 46
4.4 Solution's Requirements and Considerations 47

5 Implementation 49
5.1 Preprocessing and Auxiliary Structures 49
5.1.1 Initial Widget Tree 50
5.1.2 Abstracted Tree 51
5.1.3 Patterns 52
5.2 Alghorithms 59
5.2.1 Tree Abstraction 60
5.2.2 Pattern Type Search 61
5.2.3 Pattern Creation and Insertion 63

5.2.4 Transformations . . . 66
5.3 Progress Beyond UI . . . 73
5.3.1 Full Screen Migration . . . 75
5.3.2 Inputs and Variables Migration . . . 77
5.3.3 Aggregates Migration . . . 78
5.3.4 References Repairing . . . 79
5.4 Migration Information . . . 80
5.4.1 Migration Logs . . . 81
5.4.2 Migration Metrics . . . 83
5.5 Overview . . . 84
5.6 Integration in the OutSystems Platform . . . 84

6 Evaluation **87**
6.1 Coverage Analysis . . . 87
6.2 Queries in OutSystems Accounts . . . 88
6.3 Performance Comparison . . . 90
6.4 Usability Experiment . . . 92
6.4.1 SUS . . . 93
6.4.2 Results and Analysis . . . 95

7 Conclusions **97**
7.1 Contributions . . . 98
7.2 Future Work . . . 99

Bibliography **101**

Appendices **111**

A Migration Result **111**
A.1 Original Screen . . . 111
A.2 Migration Result . . . 112
A.3 Migration Metrics . . . 113

Annexes **115**

I Comparison Of Migration Approaches **115**

List of Figures

2.1 OutSystems Platform Architecture [61] 6
2.2 Action development in Service Studio 7
2.3 OutSystems UI Architecture [59] 8
2.4 OutSystems UI development in Service Studio 9
2.5 Choosing the paradigm of an application in Service Studio 10
2.6 Metamodels, Models and Language Defintions [43] 16
2.7 Model transformation. [83] 18

3.1 Screen created via Scaffolding patterns 26
3.2 Migration Horseshoe [90] 34

4.1 Tree structure example. 43
4.2 Edit Record Widget in Tradtitional Web widget tree. 44
4.3 Label and Input widgets pattern 44
4.4 Rich Widgets 45

5.1 Widget Tree examples 50
5.2 Widget Tree representations 53
5.3 Differences in Search widgets implementation between paradigms 56
5.4 Differences in Radio Button implementation between paradigms 57
5.5 Differences in Table and Pagination implementation between paradigms . . 58
5.6 Overview of the migration architecture and its different processes. 60
5.7 Example of an abstracted tree 62
5.8 Abstracted tree with patterns varying according to the pattern search order . 62
5.9 Tree Abstraction and Pattern Search processes. 65
5.10 Transformations architecture 66
5.11 Button Widget in Traditional Web application 70
5.12 Button Widget in Reactive Web application 70
5.13 Differences in Input Password implementation between paradigms 71
5.14 Differences in Show Record implementation between paradigms 72
5.15 Migrated Screen in the different applications' UI Flows 75
5.16 Error in Expression widget due to Aggregate not existing. 76

5.17 (41) Errors in a Screen where the Input Parameters, Local Variables, and Aggregates from the Preparation were not migrated. 77
5.18 Migrated Screen result 78
5.19 Entities migrated to the new paradigm as references. 78
5.20 Preparation Aggregates before and after being migrated (in the different Web paradigms) 79
5.21 Form and Input widget with Form Agreggate reference in Traditional Web application 80
5.22 Migrated Form and Input in Reactive Web application with correct reference 80
5.23 Aggregates, Input Parameters and Local Variables migration logs example. . 82
5.24 Patterns found migration logs example. 83
5.25 Widget migration logs examples. 83
5.26 Screen migration's performance metrics example. 83
5.27 Screen migration processes and operations. 84
5.28 Copy of widget from Traditional Web application and paste in Reactive Web application. 86
5.29 Copy of screen from Traditional Web application and paste in Reactive Web application's UI Flow. 86

6.1 Total number of widgets and number of widgets possible to migrate by category. 88
6.2 Adjective ratings, acceptability scores, and school grading scales, in relation to the average SUS score [4]. 93
6.3 Boxplot for the SUS Score distribution. 94

A.1 Screen to be migrated. 111
A.2 Screen migration result. 112
A.3 Screen data imported as reference. 113
A.4 Errors created by the automatic migration. 113

List of Tables

6.1 Average number of elements per Traditional Web Screen and Web Block. . . 90
6.2 Migration elapsed time (in seconds) and number of elements migrated for different Screens 91
6.3 Migration elapsed time (in seconds) and number of elements migrated for different Web Blocks 91
6.4 Mean SUS score for each UI migration approach per question. 94
6.5 SUS descriptive statistics for the manual migration 94
6.6 SUS descriptive statistics for the developed tool 94

I.1 Comparison of migration approaches [29] 116

Acronyms

CI Communicated Information

CSP Constraint Satisfaction Problem

DBMS Database Management Systems

DSL Domain-Specific Language

IDE Integrated Development Environment

MDE Model-Driven Engineering

OML OutSystems Modeling Language

REST Representational State Transfer

ROI Return Of Investment

SOA Service-Oriented Architecture

SOAP Simple Object Access Protocol

UI User Interface

UML Unified Modeling Language

UX User Experience

XML eXtensible Markup Language

Chapter 1

Introduction

This thesis and the related project were developed in a collaboration between Faculdade de Ciências e Tecnologia from NOVA University and OutSystems. In this chapter, we will explain the context, motivation, objectives, and key contributions of the thesis.

1.1 Context

Low-Code platforms have increased their popularity in the last few years. Being a market leader [86], OutSystems keeps on researching and investing to enrich its product - a platform that contains a development environment that allows developers to create and maintain web and mobile applications. These applications are scalable, secure and most of the implementation and deployment details are abstracted by the platform and OutSystems visual programming language, providing an easy, fast, and safer way to develop applications.

However, this need to change and improve the product to meet market demands results in multiple paradigms (e.g. Traditional Web and Reactive Web, denominations to specify different types of development paradigms in the OutSystems platform) and versions (the platform is now on version 11). In October 2019, OutSystems announced a new paradigm available on its platform which allows developers to build reactive applications. It was named Reactive Web and was built to take advantage of modern web features, presenting multiple differences to the previous paradigm to develop Web Applications (Traditional Web). Because of this, OutSystems was confronted with a challenge: 63% of its applications were implemented in the old Web paradigm, how could they benefit from the new paradigm's evolution and capabilities?

Legacy Information Systems can be defined as "any information system that significantly resists modification and evolution" [10]. So, organizations want to move these

systems to be easily maintained and adapted to new business requirements, all while retaining the existing functionalities. The process of moving these systems without having to completely redevelop them is the essence of a Legacy System Migration [9].

Migration is a type of system modernization and consists of moving a system from out-of-date languages or platforms to a more modernized environment [70]. A common method to this type of modernization is a manual migration, which involves rewriting legacy applications [82]. On the other hand, the automation of some of the migration processes can typically be achieved through automated reverse engineering tools [82].

In this context, Model-Driven Engineering introduces models that capture designs at a higher level of abstraction that conform to an appropriate metamodel. So, during the migration of a system, it is frequent to use model-driven transformations to identify the right abstractions to represent high-level requirements and design encoded in the legacy system and desired in the target system.

Since the models and domain-specific languages vary between the OutSystems development paradigms, a migration initiative becomes a complex process. To perform it, the legacy and modernized systems must be understood, interpreted, and manipulated to allow for the necessary transformations.

1.2 Motivation

The new Web paradigm presents multiple advantages when compared to the old runtime (Traditional Web), such as asynchronous data fetching and other advantages listed in section 2.1.3.5. Thus, it is normal for the clients and developers who use the platform to want to have their systems implemented in the modern web. This makes a migration necessary to transform the paradigm in which a product is implemented, as is the case of a conversion from Traditional Web to Reactive Web. A migration can save a considerable amount of time and money when compared to writing a new system and can help to avoid the creation of new bugs [82].

The initial approach to this problem was a manual migration. Such a process starts by analyzing the legacy code to understand its functionalities, which are then used to identify business processes carried out by the system. After the initial analysis, the business processes are implemented in the new system. However, it may be difficult for the migration product to contain the same business logic as the legacy system.

Nonetheless, fully manual migration is a type of migration that uses modern architecture and tools but discards a considerable part of the effort previously made on the legacy system. If it proves to be possible to automate some part of the process, while maintaining quality standards, a migration is facilitated, allowing more systems to evolve more rapidly, thus making the ecosystem less dependant on the legacy technologies.

The automation of certain aspects of the migrations grants developers more time to focus on the aspects that can manually improve the efficiency of the migrated system.

Hence, given the growth of information technologies and the surpassing need for migration tools and frameworks, providing such automation is of the uttermost priority for any community continuously changing. This is the case of OutSystems.

A case study allowed us to classify the User Interface (UI) as the feature on which clients invest the most, making it the most difficult and time-consuming part of the manual migration. It was possible to understand that, faced with the possibility of migrating, developers usually do not want to change the UI (unlike the logic, which has to be completely changed). Thus, the most prioritized feature is coincidently the major bottleneck in migrations (needing the most investment), but also the one where least human decisions are required.

Since OutSystems intends to remove the complexity of creating and maintaining applications, the same goal must be taken upon when evolving them. Identified as the major obstacle, the automation of the UI migration could constitute a great motivation for users to migrate their existing legacy applications.

1.3 Objectives

The main goal for this dissertation was to develop a solution to accelerate the migration of an application or module created with the Traditional Web development paradigm to the Reactive Web paradigm. The solution intended consisted of designing and implementing an automatic migration approach capable of converting UI elements to complement the manual migration.

As another objective, it was expected that the approach allowed to migrate a different granularity of elements, from single elements to big components, to grant the user the choice on what to migrate. Also, the developed approach should allow for the creation and search of UI patterns, thus making the migration result according to the Reactive Web's best practices.

Parallel to those first objectives, a migration of this type requires for its information and metrics to be communicated to the user.

As a final goal, the possibility of integrating the developed tool in the OutSystems platform (specifically, on the Integrated Development Environment (IDE), the Service Studio) with an easy to use interaction would be of great importance.

1.4 Key Contributions

As a final result of this thesis, a migration approach capable of automatically migrating the UI of a Traditional Web application (screens, web blocks, or individual widgets) to the Reactive Web paradigm was designed and developed as a migration tool. This tool meets every defined goal and is integrated into the OutSystems platform with a simple to use interaction. Also, progress was made beyond the objectives initially set, with the migration of elements besides the UI, as well as a thorough case study and evaluation.

Additionally, patterns and auxiliary structures were implemented to better abstract, search, and manipulate UI model elements. This brought multiple possibilities, some of which can be used in future scenarios.

This work opens a new chapter in migrating applications from different paradigms in OutSystems by automating the major obstacle in the manual migration of web applications. This brings value for OutSystems and its community, as it facilitates the migration of 57 105 applications, with a sum of 1 018 475 screens and web blocks. Also, it contributes to a Return Of Investment (ROI) to all of the developers and customers interested in using the Reactive Web paradigm by taking advantage of the formerly produced efforts in other paradigms. Not only that, but the applications in question become a step closer to using state-of-the-art technology and modern web features, hence, also impacting their final users.

Outside of OutSystems, the developed algorithms and techniques are adaptable to other models abstracting UI development and manipulation. Thus, the theoretical work contributes to advances on the subject of low-code/visual programming migrations state-of-the-art.

1.5 Structure

The remainder of this dissertation report is structured as follows:

- Chapter 2 - Background: The research performed regarding the dissertation fundamental concepts, being those the OutSystems platform and paradigms, Legacy Systems, Migrations, Model Driven Engineering, Pattern Search and Communicated Information.

- Chapter 3 - Related Work: Methodologies and approaches related to Migration inside OutSystems, Migration outside of OutSystems, Model-Driven Engineering, and Search algorithms. Also, a small discussion on how these are related to the context of the dissertation.

- Chapter 4 - Case Study and Initial Considerations: An analysis of the problem faced and possible options to define the course of this dissertation.

- Chapter 5 - Implementation: The implementation and integration of the developed migration approach. It details the necessary preprocessing and structures, implemented algorithms, presentation of the migration information, and integration in the OutSystems' platform.

- Chapter 6 - Evaluation: Conducted experiments to test the dissertation results and respective results.

- Chapter 7 - Conclusions: A brief overview of this dissertation and final result, as well as an identification of possible future works.

Chapter 2

Background

2.1 OutSystems Overview

OutSystems is a software company with a platform built to create web and mobile applications with low-code (visual modeling in a graphical interface), which can be called a visual programming language [66]. The OutSystems Platform intends to improve performance, speed, scalability, security, storage, and other aspects of developed web applications through a high-level abstraction. This frees developers from the main complexities of designing, developing, and sending an application to production by supplying them the necessary tools and resources to facilitate these aspects and increase productivity [58].

2.1.1 OutSystems Architecture

The OutSystems Platform is divided into 2 different Environments connected over Web Services:

- Development Environment

  The combination of two complementary products: Service Studio (environment to build web and mobile OutSystems applications) and Integration Studio (environment to integrate existing third-party systems or other components).

- Platform Server

  Service with the system features to generate, compile and publish native C# or Java Web Applications (or even native Android or iOS applications for mobile).

The main components of the OutSystems platform and Platform architecture are depicted in Figure 2.1.

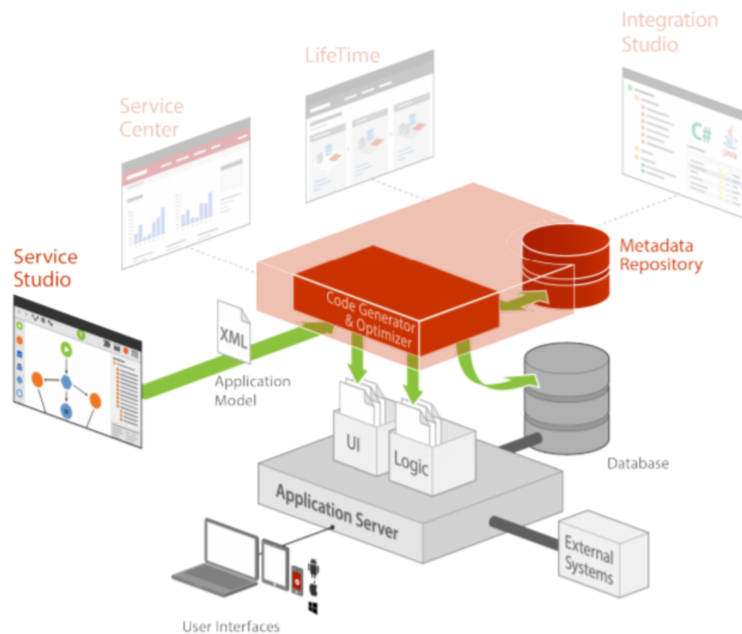


Figure 2.1: OutSystems Platform Architecture [61]

2.1.1.1 Service Studio

Service Studio (figure 2.2) is an IDE that provides the tools and components necessary to create, build and change web and mobile applications through visual modeling in a drag and drop graphical interface. It can be used to delineate the Business Processes, the application UI, the Data Layer (including Databases), Logic, Integrated REST and SOAP Web Services, and Security concerns of the chosen application.

OutSystems ensures through Service Studio a "full reference-checking and self-healing that works behind the scenes to ensure that changes will not impact existing applications" [61]. When a publication of a web application occurs, the application model is saved as an eXtensible Markup Language (XML) document and sent to the Platform Server [61].

2.1.1.2 Integration Studio

Integration Studio is an environment where, according to [61]: components can be created to integrate with existing third-party Systems, Microservices, and Databases, and also, developers can extend OutSystems with customized code.

Developers use Visual Studio to code integration components and, once those components are deployed to the Integration Studio, they can be reused by all of the OutSystems applications [61]. Also, using Visual Studio, one can use existing .NET Libraries and when publishing a component, the development environment compiles it with a standard .NET compiler. The generated DLLs are sent to the Platform Server [61].

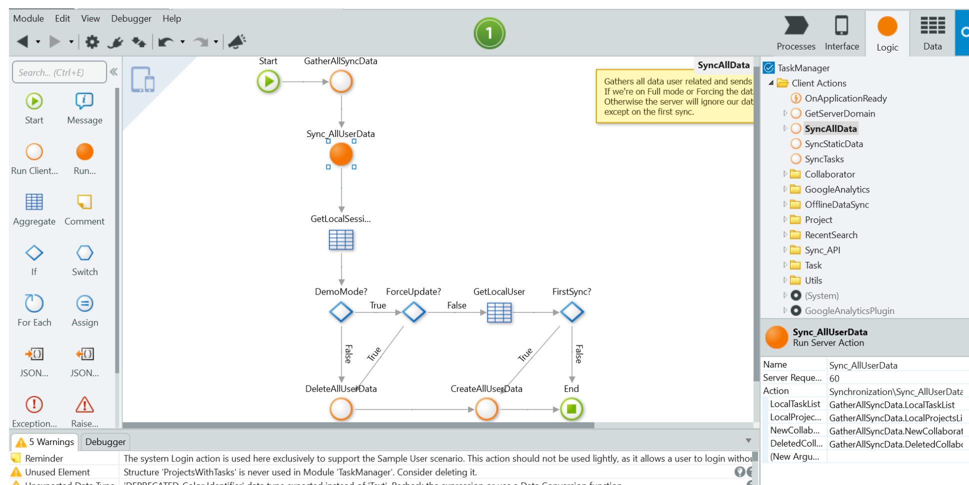


Figure 2.2: Action development in Service Studio

2.1.1.3 Platform Server

The Platform Server is responsible for the steps to generate, optimize, compile, and deploy OutSystems applications in native .NET or Java Web [61]. This is done using specialized services such as:

- Code generator: This component takes the application modeled in the IDE and generates native .NET code. Applications are optimized for performance, security, and run on top of standard technology [61].

- Deployment services: These services deploy the generated .NET application to a standard web application server. It ensures that an application is consistently installed on each front-end of the server farm [61].

- Application services: These services manage the execution of scheduled batch jobs and provide asynchronous logging services to store events (such as errors), audits, and performance metrics [61].

Platform Server generates versions for the application model and completes a dependency analysis to check other applications affected by the changes.

2.1.1.4 High level Components

A Web or Mobile application in OutSystems is a set of numerous modules, and a solution is a set of numerous applications. A module can be an eSpace or an Extension.

An eSpace is a module where an application is created and screens, logic, and entities are developed using the OutSystems visual language. An Extension is code written in .NET or Java that can be used to extend the functionality of OutSystems applications.

So, as previously mentioned, a solution is composed of a set of modules and extensions, as well as the relations between them. In other words, the content of the environment

(including eSpaces developed and extensions used), when structured and related among itself, defines a solution.

2.1.2 OutSystems Development

OutSystems languages lets developers develop and build actions (Logic), screens (UI) and processes [59]. In this chapter, only the interfaces and the logic will be addressed due to the scope of this dissertation and the resulting project.

In OutSystems, the application logic is implemented through Actions. This means that custom actions can be created and used when programming in OutSystems [59]. There are three types of actions:

- OutSystems built-in actions: Actions defined by the platform that cannot be modified or inspected (can be used in action flows, such as Entity Actions, System Actions, or Role Actions) [59].

- Custom actions: Actions that can be created to define business rules, fetch data from the database, run integrations with external systems, and other operations [59].

- Actions to handle System Events: Actions that run at specific moments of the application life cycle, such as when a web session starts or a mobile app resumes. It is possible to design the flow of these actions according to business rules [59].

The OutSystems UI Framework is the base of all user interfaces and provides UI patterns for Web and Mobile applications with built-in responsive screen templates. It also allows the developer to create customized templates with which it is possible to create a Style Guide, that defines all the patterns and styles for building the applications of a user [59]. Some of the elements of OutSystems UI are columns, cards, dropdowns, notifications, search, videos, bottom bars, and a variety of widgets [65].

Figure 2.3 represents the UI Architecture and the figure 2.4 shows the UI development of an application in Service Studio.

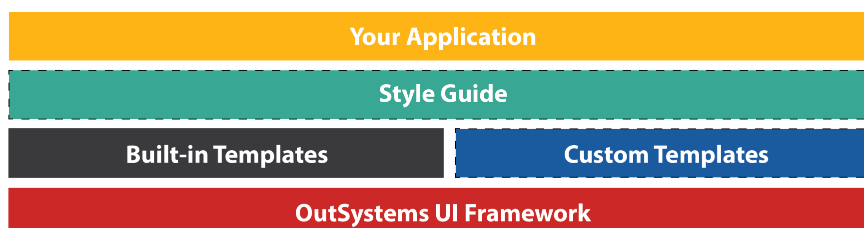


Figure 2.3: OutSystems UI Architecture [59]

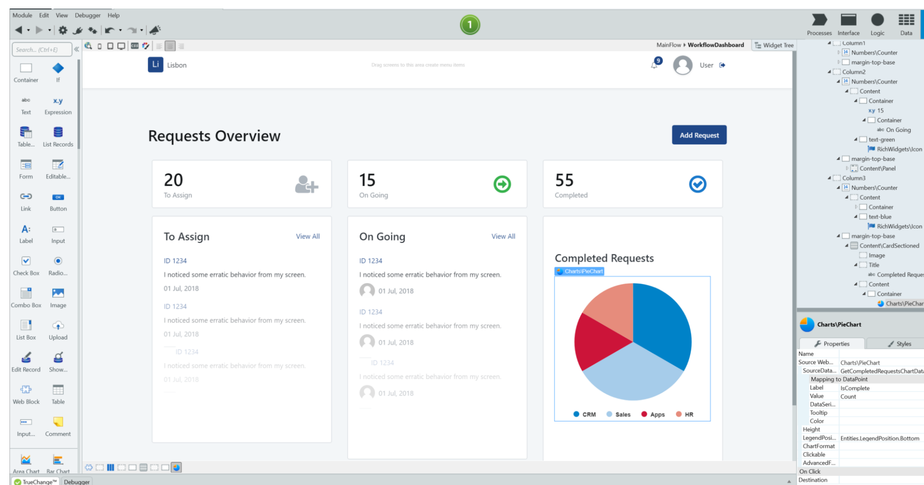


Figure 2.4: OutSystems UI development in Service Studio

2.1.3 OutSystems Application Development Paradigms

When developing an OutSystems application, the development paradigm may be chosen according to the application. Besides, it is possible to structure the application into several modules, each implementing a specific concept [59].

The existing paradigms are Reactive Web, Mobile (used to create Phone and Tablet Applications), Traditional Web, and Service, and all of them have their particular user processes, UI flows, Screens, and Blocks. So, depending on the paradigm, different actions and even UI can be used (some elements present in a paradigm may not be present in others). The preferred paradigm is chosen when creating a new application in Service Studio, as seen in figure 2.5.

Web applications and Mobile applications have different programming models. In this chapter, we will highlight the paradigms of Web Development (Traditional and Reactive) as the ones included in the scope of this dissertation.

2.1.3.1 Mobile

A Mobile application is a native app shell, developed using Apache Cordova [27], that wraps a Web Application developed using the OutSystems visual programming language [59]. When the application is built adopting this paradigm, the User Experience (UX) is optimized for mobile devices and can access their resources and features using plugins. Besides, it can also work offline and have data-caching features using local storage. The developed code is cross-platform and runs on all of the supported mobile platforms (iOS and Android) [59].

2.1.3.2 Service

Services can be used to abstract specific business concepts or business-agnostic services that extend the framework. In a Service module, the elements which compose the core

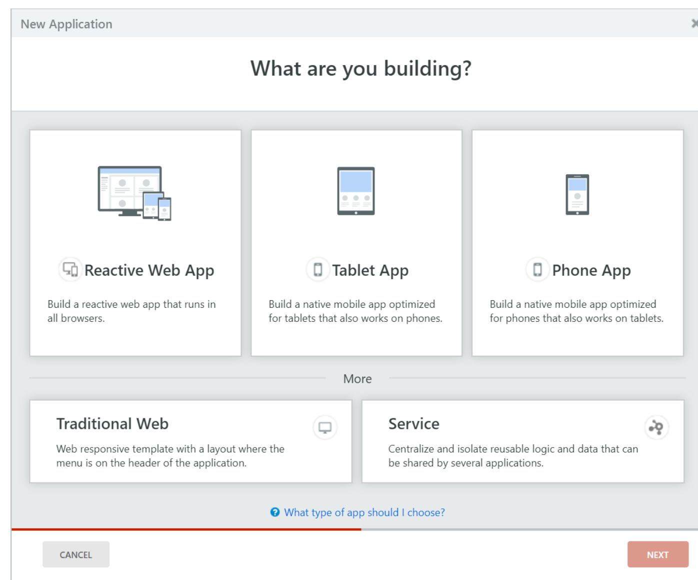


Figure 2.5: Choosing the paradigm of an application in Service Studio

services can be defined (e.g. the service logic, integrations (SOAP, REST and SAP), and database entities) and later be exposed as functionalities to other applications, following a service-oriented architecture [59].

2.1.3.3 Traditional Web

Traditional Web is an earlier type of OutSystems application centered on server-side development [59]. This paradigm allows a developer to do Full-stack Web development and build a web application from the ground up. This type of application is based on request-response interactions between the user and the server. The user uses the browser and communicates with web servers using the HTTP protocol. The server response contains the requested resource (as an HTML page with references to CSS and JavaScript elements) [56].

A Web application can have one or more modules of type: Web Responsive, Web Blank, Service, or Extension. Modules are where developers create the data model, define business logic and build web pages, and they can share elements with other modules (producer and consumer modules, of the same or different applications) [56].

In this paradigm, the server is responsible for logic and database operations. Also, it is characterized by explicit server-side rendering.

2.1.3.4 Reactive Web

In OutSystems, the Reactive Web is a fairly recent paradigm with the goal of building applications with a responsive interface (adjusts the layout depending on device type and screen size) running in the browser, displaying a user experience adapted to all kinds of devices and screen sizes. It concedes the developer the ability to interact with the device's

features and capabilities by extending the application code using HTML5 and JavaScript. This new type of application is mostly used for displaying a high volume of data, such as dashboards and tables, and it is crucial when targeting web desktops and responsive applications [59].

According to [57], the Reactive Web paradigm brings value to the built applications due to the following (new) features:

- Asynchronous data fetching
- Reactive client-side UI rendering and client-side logic
- Only data that is being used is sent to the browser or device, dramatically reducing the payload when fetching data
- Patterns optimized for the client-first development paradigm, now introduced in Web Applications developed through the OutSystems Platform
- Reusability of components between the Mobile and Reactive web applications
- New OutSystems UI framework (customizable screen templates and UI patterns)

All of these features result in better performance by revisiting the way data is handled, better UX/UI, and a state-of-the-art framework with a language constantly adapted to the modern frontend architectures [62].

The applications built with Reactive Web can be any kind of application, but those particularly reliant on data-intensive interactions and with dynamic interface patterns are the ones who can benefit the most from the technology [62].

2.1.3.5 Differences Between Traditional and Reactive Web in OutSystems

There are some elements that were commonly used in the Traditional Web that are not available in Reactive Web Applications [64]. One of those is the Ajax Refresh, which in Traditional Web refreshes parts of the interface. This element disappeared since the UI elements in Reactive applications refresh automatically on data change. Another difference is the Preparation, a dedicated server-side action that loads initial data for screens, which does not exist in the client-side Reactive interface (to promote data fetching optimization according to the application). Other differences can be seen in elements such as the Entry node, Notify, Exception Handler, and Session Variable (a server-side feature of the Traditional Web paradigm to store session information that can be accessed across the application, replaced by Client Variables in Reactive Web)[64]. Not only that, but many UI elements have considerable differences regarding their implementation in the Web paradigms. In some cases, the elements of a paradigm may not even exist in the other paradigm.

Apart from the differences, some practices were improved in Reactive, namely the appearance of Client Actions (reducing custom JavaScript), optimized data fetching, the use of libraries for a solid architecture, and the introduction of client-side validation [64].

To take advantage of these improvements, a developer can build new applications using the Reactive Web paradigm. However, the applications implemented before the Reactive Web Applications release in the OutSystems platform could benefit from using the new features available (mentioned in section 2.1.3.4), so, to make use of the reactive value, a migration is necessary. Consequently, the differences in the paradigms' implementation will have to be taken into account when migrating a Traditional Web UI to a Reactive Web application.

2.2 Legacy Systems Overview

Legacy Information Systems are the foundation of an organization's information flow and the main vehicle for consolidating business information. This type of systems are mission-critical, and a disruption in their factors would result in a serious impact on business [8]. A Legacy System can also be defined as "any information system that significantly resists modification and evolution" [10].

More and more, organizations want to move their legacy systems to new environments due to their appealing features, such as loose coupling, abstraction of underlying logic, agility, flexibility, reusability, autonomy, statelessness, discoverability, and reduced costs [2]. This lets information systems be easily maintained and adapted to new business requirements but retain the functionality of existing systems without having to completely redevelop them. This is the essence of Legacy System Migration [9].

In the context of this dissertation, the Traditional Web paradigm will be considered the Legacy System.

2.3 Migration Overview

Migration is a type of Information System modernization and consists of moving a system from out-of-date languages or platforms to a more modernized environment which allows the system to be easier to maintain and fit current business demands [70]. As migration includes a large scope of processes, a definition for it can be:

Migration is the passage of a current operating environment of a system to another usually better, and can range from single systems to multiple systems or applications. The transition can be to new hardware or software or both, ensuring continuity of operations. [80]

According to Chithralekha et al. [29], migration can be a combination of Language or Code migration, Operating System migration, Data migration, UI migration, Architecture migration, System Software and Hardware migration or migration of any of these

individually [31]. It can be executed automatically, semi-automatically, or manually.

2.3.1 Phases of a Migration

To migrate a system, it takes a process that incorporates multiple steps or phases detailed by D. O'Sullivan et al. in their work [9] and by J. Hage et al. [40]. These are:

1. Legacy System Understanding

   For a migration to succeed it is important to understand the functionality of the legacy system and its interaction with the domain [9]. Multiple techniques can be used, for instance, reverse engineering, program understanding, and architectural recovery. This is usually done with aid to tool support [40].

2. Target System Understanding

   The indicated phase portrays the target environment for the system and consists of tasks such as defining the principal components/functionalities of the environment, specific technologies and standards to be used, and search the state of the targeted system and availability of existing identical services to reuse [40].

3. Evolution feasibility determination

   Taking into account the previous steps, the feasibility of evolution has to be determined. The feasibility estimation is achieved at a technical, economic, and organizational level. The code complexity in technical evaluation and evaluation of the ROI regarding economical feasibility can be included in the estimation [40].

4. Implementation

   A target system is developed according to a requirements specification based on the previous phases with the goal of having the same functionality as the legacy system [9]. Using various approaches supported by different tools (such as wrapping, program slicing, concept slicing, graph transformation, code translation, model-driven program transformation, screen scraping, code query technology, and graph transformation) the legacy code can be extracted/used as services [40].

5. Deployment & Provisioning

   The last phase is concerned with services deployment and management after the extraction of the legacy code. After the extraction, services are deployed in the service infrastructure of the new system [40].

2.3.2 Migration compared to other modernization techniques

Modernization is a set of modifications with the goal of improving legacy systems, sometimes involving system restructuring, functional enhancements, and the implementation of new features [17]. This process is used when a legacy system needs a bigger change

than the one provided through maintenance, to increase its value and adapt to modern technology.

Other types of information system modernization are:

- Redevelopment/Replacement, which involves rewriting the legacy application from the ground up. Redevelopment is a technique successful in producing the desired result, however, it is usually costly in terms of time, money, and effort. This is because redevelopment does not leverage the legacy system's investment [82]. This approach is pertinent when another type of modernization is not cost-effective or even possible (when the system is undocumented, outdated, or not extensible) [18]. To sum up, "replacement is basically building a system from scratch and is very resource intensive"[17].

- Wrapping, which consists of gluing useful legacy code with wrapper code to embody it into a new modern system. This technique consists of determining the relevant legacy code (typically through automated reverse engineering tools), extracting it, and building a new component using it. Much of this work can be automated, and the complicated part deals with the extraction and decoupling of applicable code, as well as the production of an interface to wrap it [82]. All in all, "wrapping consists of surrounding the legacy system with a software layer that hides the unwanted complexity of the old system and exports a modern interface"[18][17].

Compared to the other techniques, E. Stehle et al. [82] considered migration to be the direction of all of the other techniques, but in its core a variation of the wrapping methodology. In other words, a migration is a combination of different phases of wrapping and redevelopment in different quantities, using as support reengineering techniques.

2.3.3 Migration approaches

The process of migrating a system can be approached in many different ways. D. O'Sullivan et al. conducted a detailed study in [9] where they described multiple data-intense migration approaches which could be classified as:

- Gateway Migration Approaches

  According to [29], Gateway Migration approaches are approaches that make use of gateways to successfully deliver the migration. Gateways are required for concurrent access to the legacy and target system, allowing interoperation between the two heterogeneous information systems.

  - Examples: Database First Approach, Database Last Approach, Composite Database Approach, and Chicken Little Strategy.

- Non-Gateway Migration Approaches

The Non-Gateway approaches are gateway free techniques that do not adopt a gateway for the migration of legacy systems [29]. The suitability of each approach and respective techniques differs for different legacy systems and target systems.

- Examples: Big Bang Approach and Butterfly Method

A Gateway is a software module between components to mediate between them [10]. The approaches will be detailed in the Related Work of this dissertation (chapter 3).

2.4 Model-Driven Engineering

Model-Driven Engineering (MDE), can be defined as:

The unification of initiatives that aim to improve software development by employing high-level, domain-specific models in the implementation, integration, maintenance, and testing of software systems. [85] [6]

During the migration of a system, it is common to use model-driven transformations to identify the right abstractions to represent high-level requirements and design encoded in the legacy system and desired in the target system. Specifically, MDE introduces models that capture designs at a higher level of abstraction. After that, developers represent designs using models that conform to an appropriate metamodel, which are then automatically transformed into implementations [85].

2.4.1 Modeling

2.4.1.1 Model

As stated by [25], the simplification of a system with a planned goal can be considered a model. In other words, a model is an abstraction of a system that exists or is intended to exist at a certain point in the future. From the software engineering perspective, a model is a fabrication specified in a modeling language, which describes a system (real or language-based) and allows predictions or inferences to be made [45]. Being a set of statements about a system under study, the model is the system's reduced representation and highlights the properties concerning a particular perspective [72]. In some cases, it replaces the need to consider the original system directly.

It was reported by Stachowiak, that for a model to be identified and distinguished from other types of artifacts, it must meet the following criteria [49]: (1) Mapping criteria: There is an object or original phenomenon of the system represented or mapped in the model; (2) Reduction criteria: Not all the features of the original are depicted in the model, hence, the model is a simplified version of the original; (3) Pragmatism criteria: The model should be able to replace the original for certain purposes, making it useful.

2.4.1.2 Metamodel

A metamodel is a model of models [45], or a model of a modeling language [25]. This concept is many times used without a strong definition or in a very simple way, as is the case of the OMG's definition that states: "a metamodel is a model of models" [1].

Some authors went further and gave the following definitions: a metamodel is a definition of a language to express a model [43]; a metamodel is a model of a language of models [26]; a metamodel is a specification model where each system specified is a model expressed in a modeling language [71]. According to Favre and NGuyen [26], the relation between a model and a metamodel relies on the fact that "a model must conform to a metamodel". The figure Figure 2.6, adapted from [43], depicts the relations between a system, model, metamodel and language.

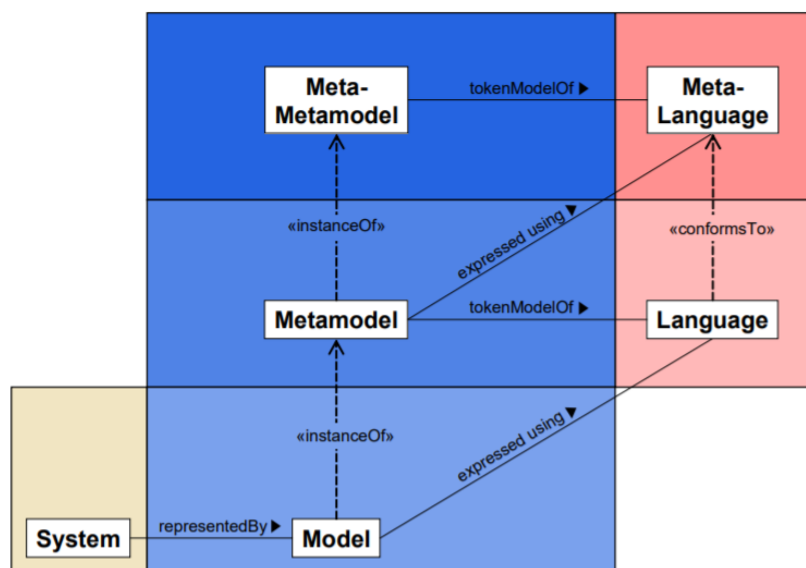


Figure 2.6: Metamodels, Models and Language Defintions [43]

2.4.2 Modeling language

A modeling language allows someone to express the statements in models of some class of System Under Study (SUS) [71], i. e. a model is an element of a modeling language. Besides, a modeling language is defined by a metamodel and comprises all the possible models consistent with such metamodel.

We can classify a modeling language as general-purpose (GPML) or domain-specific (modeling) language (DS(M)L) [52][69][23][34]. A GPML is a language distinguished by a larger number of generic constructs, which facilitates a broader and widespread use in various application fields. A DS(M)L tends to use few constructs or definitions that are closer to their application domain. The Domain-Specific Languages (DSLs) are expressed

[1] OMG: Object Management Group – MDA (Model Driven Architecture) available at http://www.omg.org/mda/

using domain concepts, so they have the following benefits: A DSL allows a solution to be expressed at the problem domain's level of abstraction, so it is usually easier to domain experts to understand, validate, modify, and often even develop DSL programs [23][34]; Programs using DSLs are concise and can be reused for different processes, making them enhance productivity, reliability, maintainability, and portability [23][34]; This type of languages also provides the possibility to validate and optimize at the domain level [23]. As for downsides, DSLs have an associated cost to learn, implement, and maintain [52].

2.4.3 Model Transformations

According to S. Sendall and W. Kozaczynski [73], to guarantee the overall consistency of models related to each other, a significant amount of work is required. A model transformation is a set of automated processes that take one or more models as input and produce one or more target models as output, by following a set of transformation rules. This automation reduces the effort of activities like reverse engineering, view generation, application of patterns, and refactoring [73].

To execute a model transformation, one must have a deep knowledge of the abstract syntax (commonly defined with a metamodel) and concrete syntax of the source and target models. The kind of transformations from a model to a different model is called model-to-model (M2M) [32] and are deployed as software. So, like other types of software, they need to be analyzed, designed, implemented, and tested, requiring engineering processes, notations, methods, and tools [32].

Regarding implementation, model-to-model transformations can be defined for various purposes and with specific modeling paradigms through common languages, such as the standard programming languages, but also through specialized model transformation languages [83]. A model transformation has as input a source model and as output a target model, where both models conform to their meta-model. The meta-model of the source and target models can be the same, or different, making the transformation endogenous or exogenous, respectively. Thus, a transformation is specified at the meta-model level [83].

A transformation is generated automatically and is carried out on any source model that conforms to the source meta-model. This makes the source model, the target model, and the transformation specification models in itself, conforming to their respective meta-model [83].

Figure 2.7 illustrates the process around model transformations (adapted from [83]).

T. Kühne, E. Syriani and others [44] considered modeling a transformation language (including its semantics) a substantial initial investment. However, the authors also considered such investment worthwhile due to "the prospect to more easily experiment with language features, customize them for certain purposes, and allow transformations to be reasoned about and/or modified". To create the transformation language's mapping procedure, rule-based transformations need to be modeled.

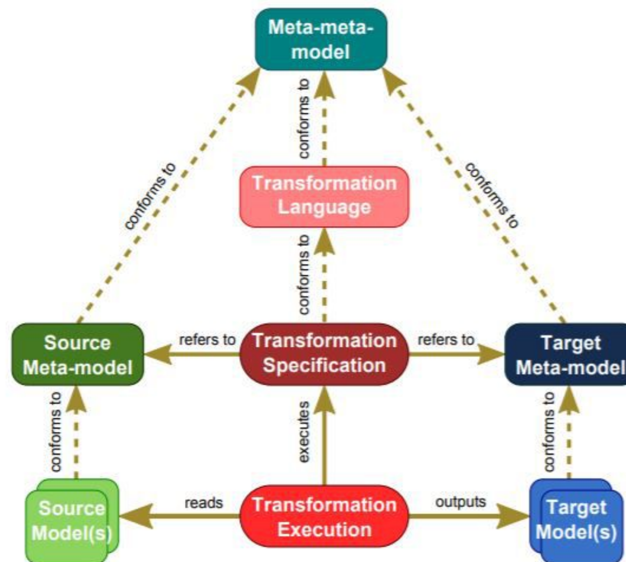


Figure 2.7: Model transformation. [83]

The specification to define the transformation language and mapping rules rely on other languages, namely the input and output language. These languages on the ends of the transformations define the pattern specifications for the left-hand side (LHS) and right-hand side (RHS) languages. Thus, only the well-formed input and output specifications on both sides can be involved in a transformation [44].

A rule is a declarative construct that specifies what can be transformed and into what [83]. It is made of pre-condition (determines the applicability of a rule) and post-condition (what is found after the rule was applied) patterns. These conditions are described through the left-hand side (LHS) and righthand side (RHS), respectively, where the LHS must be found in the input model to apply the rule, and the RHS must be found in the output after the rule is applied [83].

2.5 Pattern Recognition

OutSystems UIs are composed of multiple widgets, which can be grouped as Web Blocks, menus, and other complex elements. Also, the bindings between the UI elements and the data and actions vary from application to application. This means that, while some widgets can be read from a Traditional Web application and recreated directly in a Reactive Web application, other widgets cannot. Also, some widgets and types of data fetch/binding are supported in one paradigm but not the other. This creates different scenarios in a migration, including some where widgets and bindings must be read as patterns with certain functionality, which have to be replicated on the product of the migration.

To sum up, in a migration there may be cases where it is better to identify what to migrate, not by looking at the elements, but by looking at the patterns they belong to. This is a common approach in model migrations using the concept of MDE [85],

as also explained in sections 2.4.3 and 3.3.3, but also in other types of migrations [87]. Not only that, but this technique is appropriate for the OutSystems platform due to the incompatibilities between paradigms and their models (listed in this section and section 2.1.3.5).

2.5.1 Pattern Search

"Pattern matching is a basic problem in computer science and it occurs naturally as part of data processing, information retrieval, speech recognition, vision for two-dimensional image recognition and computational biology"[53].

Pattern matching consists of comparing tokens with the parts of a pattern to check and locate similarities. If the matching tolerates small differences between the objects and the pattern, it is called pattern recognition and consists of the automated recognition of patterns and regularities in data.

The UI in OutSystems will be the focus of this dissertation (section 4.1), so, to search the patterns, the algorithm will have to take into account the structure of the interface. Since the UI is structured and connected in the Widget Tree (present in Figure 2.4), the background and related work analysis will focus on graph pattern search and, more specifically, on tree pattern search.

2.5.2 Graph Pattern Search

Graph Pattern Matching, or Graph Pattern Search, is typically defined in terms of subgraph isomorphism, making it an NP-complete problem [24]. Besides the subgraph isomorphism problem (where matches are evaluated based on the graph structure, Graph Pattern search can also be used to discover matches similar (but not the same) to complex patterns [28].

According to [28], Graph Pattern search objective is to find matches in a graph for a specified pattern. A formal specification is as follows:

1. A graph $G = (V, E)$, composed of a set of vertices V and a set of edges E. Each $e \in E$ is a pair (v_i, v_j) where $v_i, v_j \in V$. The vertices and/or edges of G may be typed and/or attributed.

2. A pattern graph (or pattern query) $P = (V_P, E_P)$, which specifies the structural and semantic requirements that a subgraph of G must satisfy in order to match the pattern P.

The objective is to discover the set M of subgraphs of G equal to the pattern P. A graph $G' = (V', E')$ is a subgraph of G if and only if $V' \subseteq V$ and $E' \subseteq E$. Problem formulations often require that P represents a single connected graph and, hence, that $m \in M$ is also connected [28]. A graph is connected if there is some path between every pair of its vertices.

The algorithms to search graphs can be classified as approximate (with polynomial complexity but are not guaranteed to find a correct solution), inexact, and exact (both find correct answers and consequently have exponential worst-case complexity) [76].

2.5.3 Tree Pattern Search

Tree Pattern Search is similar to Graph Pattern Search. In this scenario, there is a match between a pattern tree P and a tree T at node v if there exists a one-to-one map between the nodes of P into the nodes of T such that: (1) the root of P maps to v; (2) if x maps to y and x is not a leaf, then x and y have equal degrees and the ith child of x maps to the ith child of y; (3) if x maps to y, then x and y have the same labels [42]. This is a description of an exact match.

A tree is a data structure that can be either binary or non-binary, so, the type of search algorithm and approach depends on the type of tree. Equivalent to the graph search, the algorithms to search trees can be classified as exact or inexact. Inexact matches can have some flexibility in the sense that some of the nodes are not relevant to determine whether the tree and the pattern are a match (there may be some differences between some nodes).

According to [36], in pursuance of extracting useful information from tree-structured data, it is necessary to extract tree patterns that are common to that data.

2.6 Communicated Information

The result of an automatic migration between paradigms is a process where multiple conversions are made, some more direct than others. So, there is always the possibility of the final product needing a manual quality assurance or, in the worst-case scenario, some components were unable to be transformed or produced an undesired conversion. Therefore, during or after the migration tool execution, the transformations completed or in need of manual operations (or verifications) must be documented and presented to the user. Consequently, to register and present such documentation, the program will need to communicate some sort of information.

As reported by W. Shang et al. [75], "system administrators and developers typically rely on the software system's Communicated Information (CI), consisting of the major system activities (e.g. events) and their associated contexts (e.g. a timestamp) to understand the high-level field behavior of large systems and to diagnose and repair bugs". So, it is the developers' role to choose which information is important to the system operation and analysis [75]. A typical technique to display CI is logging.

2.6.1 Metrics

As stated in [81], "Logs, metrics, and traces are often known as the three pillars of observability". Metrics are a way of depicting the data measured using numerical values. This

data is usually gathered over intervals of time [81] and can take distinct representations such as counters or gauges [38].

2.6.2 Logging

Logging records useful information during system execution. The information is used for maintaining the system, bug-fixing, detecting anomalies, and transferring knowledge [39]. Logs are strings of text containing considerably more data than metrics, and generally require parsing to get useful information without a human reading them [38].

Logging statements (also called execution logs) consist of a textual part to detail the context, a variable part to give further information, and a "log verbosity level"[39].

Thus, logs are advantageous to help engineers and programmers called to diagnose and solve issues during or after production run, as well as understanding the operations and overall program performance [54]. This type of feedback is essential in a migration between distinct paradigms as is the case of this dissertation's objective, to enable a higher understanding of the process result, and manual verification of the results and problems encountered. Not only that, but due to the differences between paradigms and their models, migration logs are a practical way of communicating to the user which elements and relations were not possible to migrate, or had to suffer significant changes to be transformed into elements of the new paradigm.

While metric storage is inexpensive, storing logs can be costly since it is frequent to generate large quantities worth of data per day [38]. So, the observability of the migration process (metrics and mostly logging) may bring an expected, but a small penalty to the tool's execution time.

2.6.3 Traces

A trace is a representation of a set of events related to each other and it shows the end-to-end flow of requests through a distributed system. Another way of defining traces is as a representation of logs, where "the data structure of traces looks almost like that of an event log"[81].

Tracing helps a software engineer to monitor the sequence of distributed events (different services involved) and the complex interactions characteristic of a microservice architecture [38]. Traces are also used to present the work done at each layer, all while preserving causality [81].

CHAPTER 3

Related Work

3.1 OutSystems Migration

In 2017 OutSystems released the Mobile paradigm (which allowed developers to build applications for mobile phones and tablets) and the Modern Web became a trending theme inside the company. The new way of designing and building applications with a better performance brought up the idea of a possible migration between paradigms. Such possibility turned into an initiative to study the possibilities, advantages, risks, and obstacles to a migration path. Some approaches were investigated and most of the problems were identified but, in the end, it was considered a migration would not be worth the investment at the time. So, the selected path was maintaining the old runtime (Traditional Web) and not investing in a Migration for the reasons detailed below in section 3.1.1.

As previously mentioned, the release of Reactive Web made the migration possibility relevant again, since many of the applications built using the Traditional Web could benefit from a Reactive Web implementation. Thus, a manual migration initiative was started by OutSystems to migrate the applications which would have a high ROI if converted to the Reactive Web paradigm.

To provide the necessary context, in this chapter we will detail the 2017 proof of concept, the current manual migration initiative, and how an automatic partial migration could accelerate the manual work.

3.1.1 Previous Work

The steps taken in the proof of concept made in 2017 concerning a migration possibility were: Identifying the differences between runtimes (at the time, only Mobile and Traditional Web existed), identifying problematic migration scenarios, estimating the

percentage of modules where a migration would be possible, identifying high-level migration approaches in the OutSystems context and an estimation of the migration costs and learning times. All of these topics will be detailed in this chapter, and so will the final decision of not following through with the project and the reasons for it.

Regarding the differences between the paradigms (Mobile and Traditional Web) at the time, most of them were similar to the differences mentioned in section 2.1.3.5 between the Traditional and Reactive Web. Some problematic scenarios were identified, such as modules with server-side functions in UI, which comprised 45% of the modules at the time, and modules using unescaped expressions, which were more than 51% at the time. Also, 53% of modules had Session Variables, that were not supported in mobile. Since the models had considerably different lifecycles, a big part of the migration could not be automated or even completed.

In the course of the proof of concept migration, several paths were identified and evaluated, including a Migration Tool, a Migration Tool with added Migration friendly features (adding new elements to the Traditional Web to prepare applications for migration), keeping the Old Runtime (another denomination for Traditional Web), and not providing a migration tool, thus simply discontinuing the Old Runtime (forcing clients to reimplement their applications in Modern Web).

At the time, only 11% of the modules could be migrated (with the approach taken, which consisted of migrating a part of the module automatically and another part manually). Besides, the different lifecycles made it very hard to migrate the UI. This culminated in the conclusions that the time spent migrating an application was unacceptable and there was no visible ROI in migrating an application in 2017 (as the result presented a worse performance, worse user experience, possible bugs, and security risks). The final decision was to keep the Old Runtime (Traditional Web) and support it alongside the Modern Web (Mobile and Reactive Web).

So, in the past three years, both the Traditional Web and Mobile and Reactive Web models have coexisted, with the platform supporting the different models and its runtimes simultaneously. That approach had an impact on Development, Maintenance, and Support costs.

3.1.2 Manual Migration Initiative

In October 2019 the Reactive Web was released. This brought multiple changes in OutSystems and the possibility to build web applications according to more state-of-the-art technologies and architectures. Thus, the migration subject became relevant again.

However the significant changes between Traditional and Reactive made it difficult to provide an automatic tool in time to meet the customers' demand when the Reactive Web was launched. The goal of this thesis is to provide such a tool, but in this chapter, we will focus on the manual migration initiative that was a consequence of the high demand by the OutSystems' clients.

The manual migration initiative comprises a series of steps to guide developers on how to manually transition an application implemented in the Traditional Web to an equivalent implementation in Reactive Web. The effort might be significant, depending on the size and complexity of the applications, especially in the UI, as it is the most detailed aspect in the majority of OutSystems applications [64]. Despite that, this might be a good opportunity for the developers to rethink and change some aspects of the applications, for instance, the user experience and interface (UX/UI), and the data fetching operations. Besides, there are applications in which the OutSystems best practices were not followed during the development, and the opportunity could be harnessed to change the implementation according to those practices [60].

There are many differences between the Web paradigms, mentioned in section 2.1.3.5. Those differences and the type and characteristics of the applications can change the stages of the manual migration [64]. The common steps in projects of this type, however, as stated in the OutSystems official support documentation [64] are:

1. Refactor the app to centralize the server calls

   Preparatory work before the migration, involves inspecting the logic for server calls and optimizing them by grouping them into a common logic (Traditional applications render most of the UI on the server-side, so the server calls bound to UI were not an issue. This is not true for Reactive apps, where forcing UI to wait for a response from the server makes an app appear slow).

2. Recreate the Screens using new widgets and OutSystems UI

   Reactive applications use the new UI framework (OutSystems UI), which is not compatible with the OutSystems Web UI or Silk UI (previous UI libraries). The developer then needs to recreate the Screens in new modules with new widgets and patterns, adapting existing user experience to the new UI.

3. Update the front-end logic

   This step amounts to focusing on the logic that now runs in the client-side, adapting all the bindings and data fetching to make sure no information is exposed on the client-side.

4. Fix the performance warnings

   The application must have all of the performance warnings in Service Studio (generated by the migration) fixed to guarantee a smooth and correct user experience.

There are a few accelerators to help speed up the migration of Traditional Web Applications [64]. Some of the accelerators consist of Copying and Pasting Server Actions and Entities (to Server or Client actions in Reactive Web), Copying and Pasting Aggregates (SQL nodes in OutSystems) from Traditional to Reactive Web, creating screens based on Scaffolding and Copying the Preparation and Pasting it as a Data Action.

Scaffolding patterns in OutSystems [63] are a way of accelerating the creation of screens, logic, and functionalities using only a few clicks or drag and drop. These patterns are mostly used to create screens with Create, Update, Retrieve, and Delete (CRUD) functionalities based on an entity. With scaffolding, a user can create, for example, a screen with a list of objects and another screen with the details of an object (or a screen to create a new object) simply by dragging the entity to the MainFlow (a UI flow to manage the interactions of the screens). The created screens can be further customized by the user after its creation. Figure 3.1 shows a screen created with a Scaffolding pattern by simply dragging a sample data Product entity to the MainFlow.

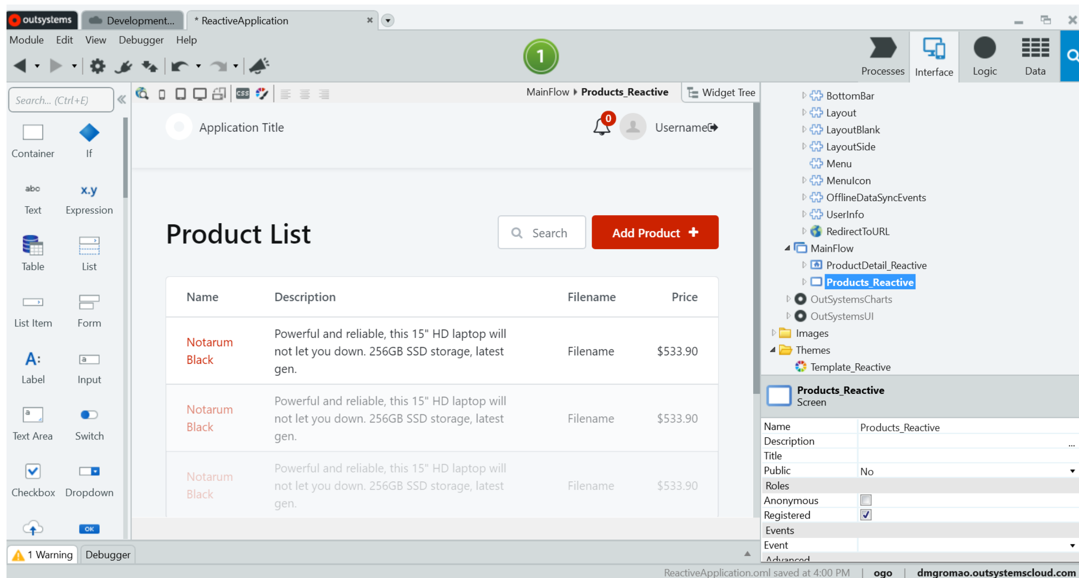


Figure 3.1: Screen created via Scaffolding patterns

In [64] we can find the documentation on how to manually migrate most of the module elements. However, while the logic, session variables, UI flows, processes, roles, and themes are all relatively accessible to migrate manually due to accelerators, the UI elements such as Web Blocks, menus, and widgets are not so facilitated.

3.1.3 Automatic Migration

The 2017 migration proof of concept came to show that the risk and cost of a full automatic migration would be too high and that a blind migration would present a worse result than the original product. Not only that, but some components of an application may be worth redeveloping to fully extract the Reactive Web paradigm capabilities.

If we also consider the fact that the manual migration initiative has been a success, with considerable adherence from OutSystems clients, one could question the value of an automatic migration, especially considering all the accelerators already available. So, after a thorough analysis of the problem, it was possible to conclude that the focus of this project should not be an automatic alternative to the manual migration, but an automation

or an accelerator to complement the manual migration initiative. In section 4.1, we will explain the focus of the project (what to automate) and its discovery.

3.2 Migration Approaches

There can be many types of legacy systems, so, different systems might require different migration approaches (according to the architectures, features, and resources). Bisbal et al. conducted a survey and described different data-intense migration approaches (that prioritize the migration of databases) existent at the time [9]. The study considered:

- Database First Approach or Forward Migration Approach

  The data is migrated before the rest and the application logic and interfaces are migrated gradually. During the redevelopment of the application and interfaces, the Legacy System can access the data environment of target systems through a forward gateway, making it possible for both systems to operate in parallel.

- Database Last Approach

  The application is incrementally migrated, and only after that, is the database migrated. A reverse gateway is responsible for the mapping between the target database schema and the Legacy System database.

- Composite Database Approach

  Both the data and application are gradually migrated, and the development can be incremental. Data integrity of the Database Management Systems (DBMS) can be guaranteed by a transaction co-coordinator. Forward and reverse gateways can be used.

- Chicken Little Strategy

  This approach is similar to the composite database approach except for the functionality and placement of gateways. Like the previous approach, Forward and Reverse gateways can be used. Besides, Legacy and Target systems can work simultaneously during the migration, and the operating system is a composite of target and legacy information systems using gateways.

- Big Bang Approach

  The Legacy System is redeveloped from scratch in the new environment, with new architecture, tools, and database.

- Butterfly Methodology

  This approach does not use gateways, so the target system is not operational during the process. The major focus of the migration is on legacy data migration in a mission-critical environment.

Ganesan et al. [29] presented a table with a summary and comparison of the advantages and disadvantages of each of the aforementioned approaches. It can be consulted in appendix I (table I.1).

3.2.1 SOA (Service Oriented Architecture) Migration Approaches

In Service-Oriented Architecture (SOA) migrations, the intended enterprise model defines the business needs of the target design, which is determined by services and their interactions [90]. Some of the approaches utilized (and mentioned in [29]) are:

SMART(Service MigrAtion and Reuse Technique)

SMART is a technique to analyze legacy components and their potential of being reused as services to perform a legacy migration [78]. It can also be defined as a set of processes to help decide the utility of exposing legacy systems as services in a service-oriented architecture.

Sneed's Approach

Sneed developed an approach [79] (integrating legacy software into a service-oriented architecture) and showed how legacy can be reused in building web services. The contribution consisted of a method capable of wrapping legacy code behind an XML shell to be offered as web services to external sources.

MASHUP (MigrAtion to Service Harmonization compUting Platform technology)

The combination of content from multiple sources into an integrated experience is called "mashup" technology. Based on that, Cetin et al. proposed the MASHUP migration technique [13], which focuses on behavioral and architectural aspects of the migration. It has the following six stages: MODEL (Modeling of target enterprise business), ANALYZE (Analysis of the legacy systems and infrastructure), MAP & IDENTIFY (Mapping business requirements to system components and services identification), DESIGN (Designing a concrete MASHUP architecture with domain-specific kits), DEFINE (Defining Service Level Agreement) and IMPLEMENT & DEPLOY (Implementation and deployment of services).

SOMA (Service Oriented Modeling and Analysis)

SOMA was defined as a process to integrate systems by analyzing legacy applications to better understand the use of services in a service-oriented architecture [3]. It also breaks the business functions of each application to identify potential services to use in the new architecture to complete business goals. Besides, problematic areas are discovered and areas where new services need to be built are signaled.

Two-View Approach

Razavian and Lago [68] described approaches using two views to analyze and categorize them: knowledge and activity. They presented a migration reference to select an approach among the existent or to develop a new migration approach.

3.2.2 Cloud Migration Approaches

In Cloud migrations, it is possible to access a set of shared resources such as networks, servers, and storage. By migrating Legacy to Cloud systems, the paradigm is changed at a business and technical level [88]. In the work of Jamshidi et al. [37] a systematic review of cloud migration takes place, characterizing 23 selected studies. The unexplored areas of cloud migration were identified (research determining applicability in an industrial context and lack of tools for migration execution, among others). This was done by synthesizing collected data and the results of classification and comparison are shown through tables and visual diagrams.

As a migration to Cloud services leads away from the scope of this project, only some names of the different approaches mentioned in [37] and [29] will be indicated: Cloud Migration approach (CloudMIG), SOA Migration Adoption and Reuse Technique (SMART) decision framework, Reuse and Migration of legacy application to Interoperable Cloud Services (REMICS) and Advanced Software-based Service provisioning and Migration of Legacy Software (ARTIST).

3.3 Model-Driven Engineering

Model-Driven Engineering (MDE) is widespread and adopted by multiple industries with a successful approach. However, it is still considered by some a niche technology [89]. Despite its increase in popularity with languages like Unified Modeling Language (UML), it is still not used as much as Java, C#, or other popular programming languages [55].

According to [55], the areas in which the advances in MDE have been considerable comprise model analysis, model transformations, model-based verifications and validation, and modeling languages. MDE has helped to address some software engineering problems, namely with foundational theories, tool support, and empirical evidence in the abovementioned areas [55].

These advances made MDE somewhat widespread and used in many different ways. However, the companies who better applied MDE tended to develop or use languages specifically created for their domains (instead of using general-purpose languages such as UML). The data collected in [89] shows that developing small DSLs is popular for limited, well-understood domains. It is perceived that significant time and effort must be spent in developing models, when in reality the applications of domain modeling can be swift and agile, using DSLs and associated generators [89].

It was a surprise to see that, as proved by a study found in [89], state-of-the-art modeling techniques and tools, such as UML, are generally not used by designers. Besides, when used, the designers used it selectively and informally. This can be explained by the conclusion that these techniques and tools present a poor performance when evaluating the support offered to software development activities [89]. In the investigation performed, there was not a consensus on modeling languages or tools, with more than 40

modeling languages and 100 tools listed as “regularly used”.

Mussbacher et al. [55] published a survey in 2014 with information and details regarding MDE adherence and related work accomplished in the past 20 years. The paper also details a prediction for the future of this technology, lists the strengths of MDE, and contains an overview of the key developments of the MDE community. Additionally, it summarises the problems found in the technology, and the authors suggest the four major challenges for MDE: Cross-Disciplinary Model Fusion, Personal Model Experience, Flexible Model Integration, and Resemblance Modeling (From Models to Role Models).

3.3.1 Modeling languages

It was previously stated that a modeling language can either be a general-purpose (GPML) or domain-specific modeling language (DSML).

General Purpose Modeling Languages are somewhat more prominent in the research and industrial areas. Additionally, the use of modeling has increased to a point where modeling standards have appeared [55].

As examples of GPMLs, in the work in [69], UML and Systems Modeling Language (SysML) are mentioned as some of the most popular, that provide large sets of constructs and notations used for documenting and specifying software systems, or for system engineering. UML is used for modeling software systems at multiple abstraction levels, with the multiple viewpoints provided which include class, object, sequence, use cases, state machine, component diagrams, among others [69]. SysML, on the other hand, is a dialect of UML 2 and is defined as a UML 2 Profile. It supports the specification, analysis, design, verification, and validation of a broad range of systems and systems-of-systems. For the past few years, UML has been the de facto standard for object-oriented modeling [55]. Another popular general-purpose language is Business Process Model and Notation (BPMN), suitable for modeling business systems from a dynamic perspective and mostly at an independent computational abstraction level. BPMN provides the following viewpoints: process, collaboration, choreography, and conversation diagrams [69]. Besides the ones listed, there are other languages commonly used, like the Object Constraint Language (OCL).

Domain-specific modeling languages (DSMLs), when compared to GPMLs, comprise a smaller set of concepts and notations and are closer to the application domain. This type of languages is relevant to the OutSystems modeling techniques as detailed in section 3.3.2. Outside of OutSystems, XIS-Mobile and DSL3S languages are examples of DSMLs. XIS-Mobile is a DSML for mobile applications implemented using a cross-platform methodology, used for modeling through the definition as a UML profile and providing the viewpoints that follow: domain, business entities, architectural, use cases, navigation space, and interaction space views (or diagrams) [69]. DSL3S, on the other hand, is used for spatial stimulation in Geographic Information Systems by providing the following viewpoints to platform-independent models: simulation, scenario, animat, and

animat interactions views [69].

In [55], a couple of frameworks that support DSML development are listed: Amongst others, MOF, EMF, VisualStudio, JetBrains/MPS, GME, Epsilon, and Xtext.

3.3.2 Modeling in OutSystems

The OutSystems Platform is a visual model-driven development and delivery platform used via its IDE, the Service Studio. Via a set of integrated DSLs, the platform allows a user to build web and mobile applications. The DSLs are visual modeling languages to develop multiple aspects of a system with a high abstraction level [33] and can be used to implement all the aspects of an application. This includes the design and implementation of user interfaces, database models, business logic, integration with external services, as well as other features [48]. As a consequence, these languages help to hide low-level details and remove the complexity of creating, publishing, and maintaining applications from the user side [33]. Since every aspect is built with a (type-safe) domain-specific language, an OutSystems model can be described as consisting of many interdependent submodels.

Models and Metamodel

In memory, an OutSystems model is represented as a graph of C# objects (for example, when the model is being edited in the Service Studio or being processed by the compiler). The graph's entry point is an instance of the class ESpace. These models are saved and transported as binary XML files where the XML representation is a serialization of the objects' graph [48].

According to [48], OutSystems models have an associated meta-model definition saved separately from the model. So, a particular version of a platform component has an associated meta-model version to help it load the models. Using the meta-model definition, the set of C# classes can be generated and are responsible for serializing and de-serializing a model to/from an XML file, among other things.

The OutSystems language meta-model is also represented as an XML file, where for each class its children and properties are detailed. To do a parallel, properties are equivalent to attributes, and children are equivalent to aggregation relations in UML terminology, as stated by Lourenço et al. [48]. Nonetheless, the meta-model is used by the Service Studio and the compiler as C# classes generated from the meta-model, instead of its XML form. The classes present in the meta-model can serve as types of properties and collections in other classes.

The model classes generated by the meta-model include the multiple predefined behaviors, for example, the code to copy, paste, load, save, and verify [33]. Since the meta-model is used as input to generate the model classes, new languages can be defined as well as their meta-models. The languages depend on a set of non-generated classes like the class Type (for types) and the ModelObject class, used as the base class for all model

Listing 3.1: Metamodel for Actions

```
<MetaModel xmlns:xsi="http://www.w3.org/2001/XMLSchema−instance"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xsi:schemaLocation="http://www.outsystems.com/MetaModel.xsd">

    <Class name="ESpace">
        <Property name="Name" type="Text" />
        <Child name="Actions" type="Action" />
    </Class>
    <Class name="Action">
        <Property name="Name" type="Text" />
        <Child name="InputParameters" type="InputParameter" />
        <Child name="Nodes" type="ActionNode" />
    </Class>
    <Class name="InputParameter" >
        <Property name="Name" type="Text" />
        <Property name="Type" type="Type" />
        <Property name="IsMandatory" type="Bool" />
    </Class>
    <Class name="ActionNode">
        <Property name="Target" type ="ActionNode" />
    </Class>
    <Class name="Start" base="ActionNode" />
    <Class name="End" base="ActionNode" />
    <Class name="Execute" base="ActionNode">
        <Property name="Action" type="Action" />
        <Child name="Arguments" type="Argument" />
    </Class>
    <Class name="Argument" verifyDependencies="Parameter.IsMandatory, Parameter.Type">
        <Property name="Parameter" type="InputParameter" />
        <Property name="Value" type="Expression" isOptional="true" />
    </Class>
</MetaModel>
```

classes [48]. An example for the OutSystems metamodel definition for Actions can be found in Listing 3.1.

3.3.3 Model Transformations

Model transformations specify the dynamic semantics, execution, analysis, code synthesization, optimization, composition, and evolution of models. So, model transformations are a key part of MDE [83].

There are two main types of model transformations studied in MDE: model-to-text transformations (M2T) and model-to-model transformations (M2M) [69]. Model-to-text transformations generate or produce software artifacts like source code, text, and XML files from models. One of the techniques to do this is code generation, and Czarnecki [22] and others discussed multiple solutions to do so. Model-to-model transformations transform models into other models closer to the solution domain, and the transformations are specified via a language, whether a programming language or a specific model transformation language [69].

The tools used to perform model transformations can offer users one or more of three different architectural approaches for defining transformations: Direct model manipulation, Intermediate representation, and Transformation language support [73]. One of the benefits of the direct model manipulation approach is the fact that the language used to access and modify the model can be a general-purpose language such as Visual Basic or Java [73], facilitating the task to developers inexperienced in MDE.

Besides general-purpose programming languages, which are seldom used to specify transformations, there are M2M transformation languages specially tailored for the task of transforming models [32]. In that respect, multiple model transformation languages exist, each with its purpose and modeling paradigm. Some examples of these languages are QVT[1], Acceleo[2], ATL[3] and VIATRA[4] [69][83][55]. Also, some approaches use UML object diagrams to represent each rule's pre-conditions, post-conditions, and notations to represent rule control flow [32].

In many cases, a transformation is detailed as a set of patterns and, when these patterns are present in the model, a transformation can be applied. In other words, a pattern is the fundamental unit of a transformation [83]. Input and output languages of a transformation establish the pattern specifications, such as the specification language, that "should not be generic to fit all possible input and output languages, but specifically tailored to the input and output languages involved"[83].

3.3.4 Migration model-based Approach

Using the concepts of Model-Driven Engineering (Sections 2.4 and 3.3), software reengineering to recover the architecture of legacy systems, and model transformations (Sections 2.4.3 and 3.3.3) there is a prominent framework model called Horseshoe Model [1][68]. This same framework is comprised of a series of steps represented in figure 3.2 that implemented in a simplified matter, the process in figure 2.7.

This technique uses three levels: the code level, the model level, and the conceptual level. It consists of representing different levels of abstraction by different models, which provide the foundation for further reverse engineering, enterprise modeling, forward engineering, and legacy migration activities [90].

3.4 Search Alghorithms

As previously mentioned, a pattern is the fundamental unit of a transformation. However, the patterns must be searched in the models to be migrated. Many search algorithms have been proposed throughout the years to increase the quality of search operations on structured information. Considering the scope of this project and the OutSystems UI

[1] http://www.omg.org/spec/QVT/
[2] http://www.eclipse.org/acceleo/
[3] http://www.eclipse.org/atl/
[4] http://www.eclipse.org/viatra/

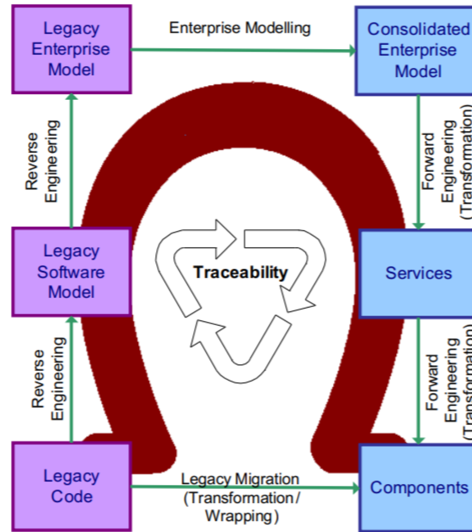


Figure 3.2: Migration Horseshoe [90]

structure, the related work concerning pattern search will focus on graph search and tree search.

Conte et al. [19] and Charaf et al. [21] conducted surveys on graph search and tree search, so we will start by analyzing the algorithms presented by them.

3.4.1 Graph Search Alghorithms

One of the most popular graph search algorithms is Ullmann's algorithm [84], which undertakes the problems of graph isomorphism, subgraph isomorphism, and monomorphism. The graph isomorphism problem aims to determine if two finite graphs are isomorphic (there is a bijection between their vertex sets). Due to the OutSystems UI being structured as a graph (more specifically, a tree), we want to find an isomorphic match to the patterns to be searched (also structured as trees). In order to prune unproductive matches, Ullman proposed a refinement procedure, that works on a matrix of possible future matched node pairs to remove those not consistent with the current partial matching [19]. Ghahraman et al. [30] worked on the so-called netgraph obtained from the Cartesian product of the nodes of two graphs being matched to prune the search space [19]. Cordella et al. [20] presented a more recent heuristic for both isomorphism and subgraph isomorphism based on the analysis of the sets of nodes adjacent to the ones already considered in the partial mapping [19]. Larrosa and Valiente [46] reformulated graph isomorphism as a Constraint Satisfaction Problem (CSP) in order to apply to graph matching some CSP heuristics [19].

Another algorithm is Nauty, developed by McKay in 1981 [51], that only covers the isomorphism problem, and is regarded by some authors as the fastest isomorphism algorithm available [19]. Messmer and Bunke [12] presented an algorithm to deal with

isomorphism and subgraph isomorphism by using a preprocessing phase to build a decision tree from the graph library and matching an input graph against it [19]. Shearer et al. [77] proposed an optimization for the case of a sequence of input graphs that are changing slowly over time, and Lazarescu et al. [47] proposed the use of decision trees for speeding up the matching against a large library of graphs [19].

There are many algorithms that produce approximations instead of exact solutions (where not all nodes are a match), called inexact matching or error-correcting algorithms, as they enable matching despite noise or errors in data [28]. Such a particular type of algorithms may be useful to identify UI patterns while ignoring user customization of certain widgets in OutSystems applications.

3.4.2 Tree Search Alghorithms

The problem of searching and matching a pattern in a tree can be defined in several ways. We will start by analyzing the algorithms that address the subtree isomorphism problem. The Näive Algorithm is based on determining that two trees are isomorphic if and only if one of them can be transformed into the other by permuting child nodes at any node [21]. This can be extended to the subtree isomorphism problem by using the basic algorithm presented by Matula [50] and Chung [15] and improved by Shamir and Tsur [74].

Another way of defining the problem at hand is to address the tree inclusion problem: Given a tree T and a pattern tree P, the objective is to locate the smallest subtree of T that contains P, in a way that P is equal to T', which is obtained by deleting nodes of T [21]. This problem was proved to be NP-hard by Kilpelainen and Manila [41], but they also detailed an algorithm to solve tree inclusion problem on ordered trees in polynomial time, which was later improved by Chen [14]. Bille and Gortz [7] later presented an algorithm with linear space bounds by creating data structures on tree T where certain operations called set procedures are executed [21].

This all leads to pattern matching algorithms, which solve the problem of matching a pattern P (or a set of patterns $P_1, ..., P_n$) containing nodes that are labeled. Hoffmann and O'Donnell [35] provided a bottom-up algorithm and a top-down method to reduce tree matching to a string matching problem [21]. This was done by including a preprocessing phase before the matching phase. In the analysis made in [21], the algorithm with the best performance in tree pattern matching was the one provided by Cole, Hariharan, Indyk [16]. Ramesh and Ramakrishnan [67] also detailed an algorithm to perform pattern matching in nonlinear patterns.

All of the time complexities of the algorithms mentioned can be found in [21].

3.5 Discussion

In this section, we will discuss which of the abovementioned related work will contribute to the implementation of this thesis. The contributions of the 2017 Proof of Concept and

OutSystems Manual Migration initiative to this project were considered in section 3.1.3.

Regarding the data-intensive migration approaches, if one were to be chosen, it would be the Butterfly Methodology approach since the legacy system (Traditional Web application) must remain operable throughout migration and there is no need to execute interoperations between the two systems (having no need for gateways and its complexities). However, this methodology focuses on data migration and develops the target system in an entirely separate process, which is not the objective of this dissertation and project. On the other hand, the SOA migration techniques mentioned in section 3.2.1, when working on the target system side, define services and interactions, also a different objective than the one of this dissertation. However, some define strategies for analyzing the legacy system, and that theoretical knowledge can be of use for this project, namely the one used in the MASHUP approach and its six stages (but instead of defining and designing services, we will have to define the elements in the new paradigm, the Reactive Web). This is also true for the SMART approach.

The Two-View approach can be used to select and define the path between the different possibilities. This work also helped to shed light on the topic of migrations and led to the Horseshoe Model, a combination of migration and model engineering operations.

The related work on Cloud Migration approaches was merely to gather information on the topic of migrations, but the techniques analyzed lead away from the project, so none of them will be pursued.

One of the most important topics covered in this chapter, the Model-Driven Engineering study is a key part of the decisions made in the requirements gathering, implementation, and integration phases of this project. The fundaments approached concerning model-driven engineering and model transformations guided the course of the dissertation and respective proof of concept. Much of the theoretical and practical knowledge, for example, regarding transformation rules, direct model manipulation, and the concept of transformations as a set of patterns, were used to understand the processes and applied when developing the final tool. Not only that, but due to the OutSystems models implementation, the migration can not be separated from modeling techniques and model manipulation.

The Horseshoe Model, presented in section 3.3.4, is an example of a possible result when model-driven engineering and migration concepts are combined. So, due to the correlation of these areas to the OutSystems paradigms (both the Web and Reactive) and their models, a variation of this methodology contributed to the foundation of the migration tool implementation steps.

About the search algorithms priorly mentioned, since an OutSystems widget tree (which is how the UI is structured) has multiple types of widgets, each with its attributes, the various algorithms were used as examples to better understand how to cover the OutSystems model and build an algorithm optimized for searching UI components in the platform.

Besides the ones above, the search algorithm present in [36] was also analyzed. This

last algorithm, besides being more recent, considers the possibility of ignoring certain sub-trees while searching the tree, which is particularly relevant in the OutSystems platform, to allow the detection of patterns despite the customization made by the developers.

Last but not least, the communicated information, specifically logging and metrics, was a key part of the study of possible migration features. The fundaments presented in section 2.6 allowed the integration of logs and metrics in the tool development, which gave the user further feedback and details when a migration occurs.

Chapter 4

Case Study and Initial Considerations

This chapter comprises the requirements gathering, necessary to define the scope of the project, the possible migration approaches analyzed for the project implementation, and the stakeholder panel composition and its influence during this dissertation. Also, it introduces the initial proposal for the solution that was implemented.

4.1 Requirements Gathering

As formerly mentioned, a migration can be a process with a considerable scope depending on the migrated system. So, to better understand the OutSystems migration problem and initiative, an initial requirements gathering analysis had to take place. The objective of this study was to contextualize the migration and both paradigms (Traditional and Reactive Web), understanding the difficulties and finding the scope of the automatic migration and how it would be related to the manual migration initiative specified in section 3.1.2.

This problem discovery and understanding were done by analyzing the platform, understanding the community interest, and interviewing multiple people associated with OutSystems and the transition to Modern Web.

4.1.1 Platform Analysis and Community Interest

At the beginning of the thesis and its project, we performed a platform analysis to better define the paradigms and their differences (Section 2.1.3). Besides the analysis on the OutSystems resources, the opportunity was seized to observe the community interest in the new paradigm and in transitioning applications from Traditional Web to Reactive Web.

4.1.2 Interviews

The main breakthrough in the problem discovery was achieved through interviews. The study present in [68] concluded that in "industrial practice, however, knowledge is also transferred by person-to-person communication", and in fact, the migration subject was a common conversation topic between OutSystems employees and developers.

Thus, the scope of an automatic migration correlated to the manual migration could not be defined without the knowledge gathered by the people involved with the product.

4.1.2.1 Product Management

The first interviews were with 2 Product Managers. One of them is involved in the Manual Migration initiative (with the role of talking to clients to give them some context, explaining the advantages of migrating to Reactive Web and explain to them how to migrate). The other was the Product Manager involved in the 2017 Migration Proof of Concept and is now a Product Manager in the Development Experience area.

Both shared the opinion that the major difficulty felt while manually migrating was the UI, which comprised Widgets, new components, and CSS already made in the Traditional Web Application. They also considered that a blind and completely automatic migration would result in a worse product than the original application, or as an alternative, in an impossible initiative.

So, for them, having to reduce the automatic migration scope to a component feature, would lead to prioritizing the UI migration with a tool that made it possible in the platform. If possible, migrating the simpler aggregates present in the Preparation, in order to keep the widget bindings, would also bring great value to the final product.

4.1.2.2 Customer Office

The next interview was with a Technical Lead responsible for a manual migration of a client application from Traditional Web to Reactive Web. Besides, he also made multiple migrations from external systems to OutSystems.

In the interviewee's opinion, the aspects that would be more helpful if automated/accelerated are: The UI (more specifically, the page structure and the components which its transformation is known and direct, such as a Menu, the Main content, the Login Info, etc.), the CSS, the Server Actions, and the Preparation of Traditional Web applications.

Having to focus on only one aspect, the Technical Lead considered that to bring a larger ROI, a scope would have to be defined where the standard and customized UI were separated, and the effort was concentrated on automatically migrating the standard UI.

4.1.2.3 Demo Team

The interview that followed was with a member of the Demo Team (the team responsible to build example applications so that the Sales personnel can show the possibilities of

the OutSystems platform to interested clients). The person interviewed was involved in a migration of a very customized demo application from Traditional to Reactive Web.

The work of migrating the UI was somewhat tedious, as all the components had to be reimplemented. Contrary to the logic and actions, which can be interesting to reimplement using the new paradigm and have accelerators, the UI was supposed to be as similar to the original as possible but had to be reimplemented from scratch.

In this interview, the option of selecting an Interface component and choosing to migrate it (such as a Widget, a Menu, a Popup, etc.) was identified as the priority of an automatic migration.

4.1.2.4 Artificial Inteligence (AI) team

Two AI members were also interviewed, as they were responsible for migrating two applications from Traditional Web to Reactive Web. Those applications were products of the AI team.

The feedback from the interviews was similar to the other collected. However, for them, it would be helpful to automatically migrate JavaScript to client actions and the Preparation to an implementation with equivalent effect.

4.1.2.5 Developers

After interviewing the people at OutSystems, to interpret and confirm the possible scope of the problem, we interviewed 13 OutSystems developers. These people contributed to a rich sample since they were of a variety of ages, with 85% being male and 15% female developers. Their experience with the OutSystems platform ranged from one and a half years to fifteen years: 1 developer with fifteen years of experience, 6 developers with between five and ten years of experience, and 6 developers with between one and a half and five years of experience. The average experience of the developers with the OutSystems platform was of 5.77 years, and most of them were learning Reactive Web for the first time.

The general opinion was that the transition to reactive is simple, so a manual migration is also relatively smooth. However, without an automatic migration of the Web Blocks, it cannot be expected of a client to migrate a perfectly working application with lots of customization, since that is the aspect where the larger investment in terms of money and time was made. Thus, to the developers, the focus of this project should be the UI in a way that the logic and actions can be manually migrated to take advantage of the Reactive paradigm.

When asked if they would be willing to migrate a working application to Reactive Web, the interviewees answered the following: 30.8% indicated they would migrate right now; 30.8% indicated they would not migrate; 30.8% indicated they would migrate but not at the moment (without further automation and accelerators); 7.6% preferred not to answer. The focus of this project is developing a tool that convinces the 30.8% of the

interviewed who are undecided to migrate, and maybe make the 30.8% that would not migrate, consider to do so. In other words, of the developers not interested in migrating their legacy applications right now, half would be interested, and another half would consider.

When asked what they thought should be the focus of an automatic migration to assist in the manual migration, the answers were: 12 developers chose the UI, 4 developers chose the Preparation, and 1 developer chose the CSS. This was an open answer question and the developers could give multiple answers.

4.1.3 Summary

Based on the analysis conducted and the interviews, the UI (Widgets and Web Blocks) is the feature which brings the most advantages and gains when automatically migrated, for the following reasons: To start, the interfaces are one of the few features that do not have accelerators in the manual migration. Besides, this is one of the features in which clients invest the most and, when migrating, developers usually do not want to change much of it. Not only that, but all of the other features are worth re-implementing to take advantage of the Reactive Web paradigm. This is confirmed by all of the data gathered.

These conclusions are obtainable when considering the automatic migration as a complement to the manual migration, instead of a parallel project. Thus, it was possible to understand the problem and scope of this thesis, as well as strongly define its requirements and objectives.

4.2 Migration Approaches

As previously mentioned in section 2.5, OutSystems application and its interfaces are composed of multiple widgets that can be grouped in different ways. When a grouping of widgets can be found in multiple applications and screens from different OutSystems' developers and factories, they constitute OutSystems UI developing patterns. In certain cases, the implementation of a UI pattern may differ from paradigm to paradigm, for example, with different widgets, UI components, or properties. To improve the migration and the suitability of the final product regarding the Reactive Web paradigm (due to incompatibilities between paradigms and their models), the migrated elements should be identified (and consequently, migrated) as patterns whenever necessary.

Nonetheless, not every widget (UI element) will be integrated within a pattern, as it can be simply recreated as an equivalent element. Besides, sometimes when we choose to migrate an identified pattern, we need first to migrate a predecessor widget in its widget tree (e.g. if the pattern is inside a Container widget). So, despite in some cases it being better to identify what to migrate as patterns, a migration could never rely solely on that, and the widgets should also be identified and migrated by themselves when they do not belong to any pattern.

4.2.1 Elementary Migration

Following the reasoning above, the solution should be able to migrate all of the OutSystems UI widgets from the Traditional Web application to the Reactive Web application. Not only would this increase the solution's coverage, but at the same time would prevent the interruption of a widget tree migration with multiple patterns only because one or more widgets part of the subtree could not be migrated. This is exemplified in figure 4.1, where the pattern can only be migrated after the root node (its parent).

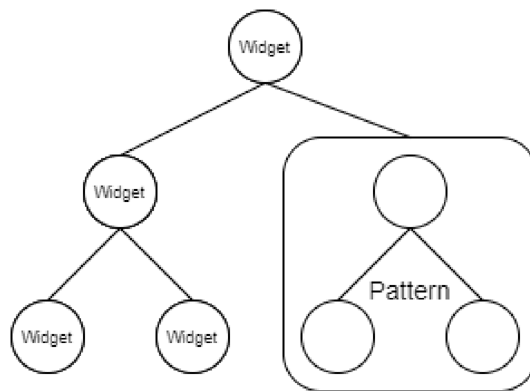


Figure 4.1: Tree structure example.

Furthermore, a pattern can be more than a set of widgets in the same widget tree, it can depend on the widgets' bindings and variables. So it is possible for a developer to want to migrate a screen or widget tree without any pattern. To achieve the thesis' objective which consisted of complementing the manual migration through the automation of the UI migration, the UI elements should be migrated whether or not they constitute a pattern. Thus, we could conclude that an elementary migration functionality was necessary notwithstanding the advantages and applicability of the pattern-driven migration to the OutSystems UI structure.

To enable such migration approach, every widget from a Traditional Web application must have a mapping to an equivalent implementation in the Reactive Web paradigm. This mapping can range from a simple creation of a similar widget to a transformation where the widget's functionalities and interface are replicated through other widgets. Since some widgets exist in both paradigms (and have representations in the respective models), those can be simply recreated and its variables assigned the same values whenever possible (e.g. the Button widget seen in figures 5.11 and 5.12), but some widgets were discontinued or their functionalities are implemented through other widgets (such as the Edit Record widget depicted in figure 4.2), which comprise more complex transformations. This needs to be taken into account in the migration through elementary mapping. For more information on the migration approach, the reader can consult the section detailing the transformations (5.2.4).

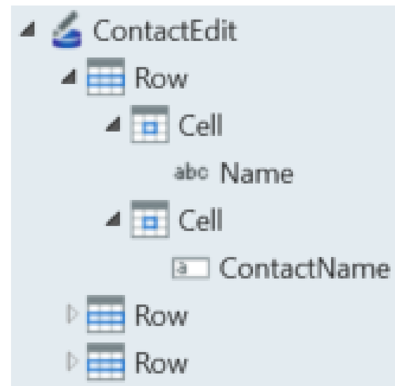
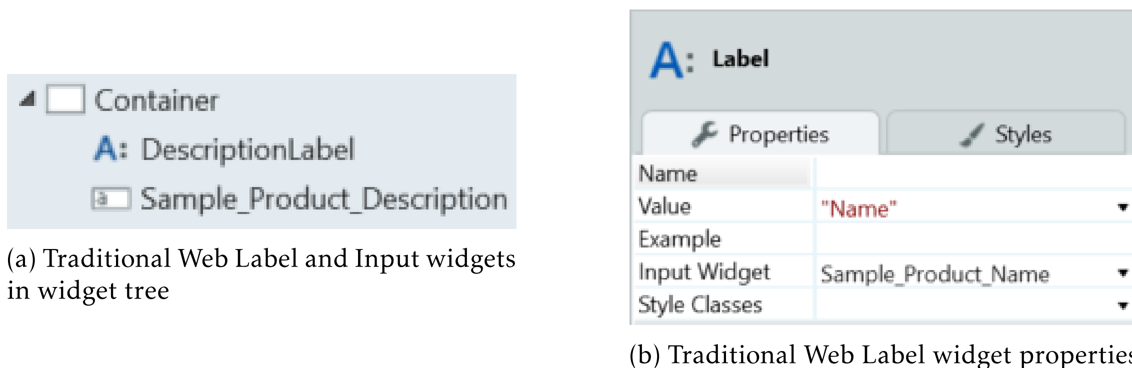


Figure 4.2: Edit Record Widget in Tradtitional Web widget tree.

4.2.2 Pattern Driven Migration

In some scenarios, it may be advantageous to perform a Pattern Driven Migration due to the differences between paradigms and their respective widgets, such as the fact that some widgets and types of data fetch/binding are supported in one paradigm but not the other.

As an example, we can look at a pattern created for the case where a Label widget references an Input widget, as observed in figure 4.3. Since the Label references the Input object, if the widgets were to be migrated individually, when the Label was migrated there would be no Input widget to set as the variable value. By migrating this specific combination of widgets as a pattern, it is possible to first migrate the Input widget, save its value, and then migrate the Label and assign that value to the property "Input Widget".



(a) Traditional Web Label and Input widgets in widget tree

(b) Traditional Web Label widget properties

Figure 4.3: Label and Input widgets pattern

There are other reasons to back up the pattern-driven migration necessity. OutSystems applications may use UI libraries with UI pattens to facilitate the implementation of certain functionalities and easier deployment of applications for the developers. Some of these libraries are the OutSystems UI or the Rich Widgets, which have UI elements implemented in the Traditional Web paradigm and can be used on Traditional Web applications. However, the same libraries do not exist in Reactive Web applications or, when they exist, contain different elements. The new runtime's applications have their libraries

and due to the different paradigms' models, the libraries from Traditional Web cannot be used in Reactive Web applications and the other way around.

The UI patterns available on the libraries are used in many applications (e.g. Input Calendar), and some are instantiated in as many applications as (if not more than) some widgets. Be that as it may, these elements are normally used under certain circumstances and their usage may be associated with other widgets, composing patterns common in many OutSystems Traditional Web applications. Not only that, but an element from the Rich Widgets library (from the Traditional Web paradigm) may have an equivalent implementation in the OutSystems UI library from the Reactive Web paradigm. However, another widget from the OutSystems UI library from the Traditional Web paradigm may not have an equivalent implementation in the Reactive Web paradigm. Some of the Traditional Web widgets from the Rich Widgets library are displayed in figure 4.4.

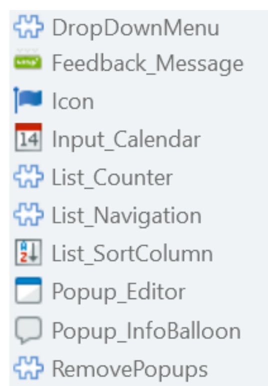


Figure 4.4: Rich Widgets

To summarise, different library elements from the old runtime may have different implementations in the new runtime and its libraries. These differences rely on multiple factors including the widgets they are associated with and their properties' values. The elements, their values, and the widgets they are associated with may constitute different patterns.

So, despite the need to support an individual mapping and migration of each widget, common combinations of widgets, data fetch/bindings and UI libraries' elements must be identified as patterns and migrated accordingly. The product of such migration should result in an equivalent implementation or one that results in a similar functionality with an overall similar aspect and experience (due to the objective of this thesis - the migration of the applications' interfaces between paradigms).

4.2.3 Mixed Approach

Based on the conclusions of the previous sections, a complete migration tool responsible for the migration of the UI between paradigms must comprise both an elementary and a pattern-driven migration. The type of migration will depend on the migrated elements and its relations and bindings, which means that in a particular widget tree or subtree,

both the different types of migrations may be appropriate for different widgets, therefore they should not be mutually exclusive.

To integrate the different approaches in the algorithm and final tool, a preprocessing is needed to transform the widget tree (present on the screen of an OutSystems application) into an appropriate structure. In this structure, patterns can be searched, identified, and bound to the widgets that must be migrated through an individual mapping. Then, it becomes possible to identify what to migrate with each approach without excluding the other and maintaining the tree hierarchy.

The solution, as well as the preprocessing phase and necessary structures and objects, will be detailed in chapter 5.

4.3 Stakeholders panel

Throughout the entirety of this dissertation, meetings were held regularly with what can be called a stakeholders panel. A stakeholder is, by definition, a person involved in a particular project due to their interest in it.

Therefore, in this project, the stakeholders were a group of OutSystems' employees interested in the project and whose contribution is valuable to the decisions made during the course of the requirements gathering, implementation, and validation phases. The stakeholders comprise multiple experts in different areas, each providing a distinct view on the migration problem, and the positions they occupy inside the company are: (1) Head of Advanced Development in Customer Office (Global Professional Service); (2) Principal Product Manager, UX/UI & Product (Software Automation); (3) Principal Product Designer in Engineering (Product Design); (4) Manager, SW Engineering in Engineering (Quality Ownership Team); (5) Lead Software Engineer in Engineering (App Runtime Team); (6) Manager, SW Engineering in Engineering (Target App Fit Team); (7) Lead Program Manager in Engineering (R&D Management).

Besides the positions they occupy, some of which are heavily involved with the manual migration initiative, they also contribute with their experience and valuable knowledge of the platform, respectively: (1) Expert on development using the OutSystems platform, responsible for training developers in both Web paradigms, and in charge of guiding developers on how to migrate their applications from Traditional to Reactive Web; (2) Product Manager of the Manual Migration Initiative and supervisor of clients' migration to Reactive Web paradigm; (3) Also responsible for supervising the clients' migration to Reactive Web paradigm and expert on UI differences between paradigms; (4) Quality expert inside OutSystems and extensive Migration experience in other companies and contexts (e.g. migration from Oracle Forms to .NET platforms); (5) Responsible for developing part of Reactive Web and expert on app runtime of Reactive Web; (6) Involved in the 2017' proof of concept and manager of Reactive Web development team; (7) Responsible for establishing relations between academic institutions and OutSystems, and emphasing the academic value of the project.

To sum up, this project had as stakeholders people involved in the multiple branches of the migration to the new paradigm, a person with extensive experience in migrations, a person involved in the 2017' attempt to a migration, a software Engineer expert on Reactive Web development, and a manager understanding of the academic value of certain approaches. Thus, it is possible to say that, during this project, we were accompanied by a panel of experts, responsible for giving valuable feedback and validate the feasibility of some of the decisions taken.

4.4 Solution's Requirements and Considerations

The solution which resulted from this dissertation is able to, through the OutSystems Traditional Web applicational model, automatically identify UI patterns and elements. After that, the solution is also capable of migrating the patterns and elements to the OutSystems Reactive applicational model, in a way to replicate the original UI aspect with corresponding functionality.

To perform a migration, a set of structures were created to integrate pattern objects in the OutSystems screens' widget tree, as well as perform a preprocessing to find such patterns among the identified widgets. Consequently, reverse engineering is applied to execute the necessary preprocessing, where the UI widgets are encoded in a suitable structure.

This, as well as the migration transformations, requires a model interpretation and transformations. Hence, MDE (section 2.4) and model transformations (section 2.4.3) concepts were combined to migration fundaments (section 2.3) to manipulate the OutSystems' model (section 3.3.2). The Horseshoe Model approach mentioned in 3.3.4 was used as an example in an initial approach.

The models' manipulation was done via direct model manipulation (explained in section 3.3.3) using C#, the same language used to generate, operate, and instantiate the OutSystems models and meta-models (section 3.3.2).

The implemented tool provides the user the possibility to migrate the selected widgets and elements, but also to migrate an entire screen or web block. The migration in question is logged and its metrics measured and presented to the user. As an additional achievement, it was interesting to observe the impact that the migration of screen elements besides UI components had on the UI migration.

After the implementation phase, the impact of the tool and respective algorithms were measured to understand how it changes the migration problem (section 6.4.2). This was done by analyzing the coverage of the tool regarding Traditional Web widgets, running queries in all of the OutSystems clients' accounts and respective applications (with their consent), doing a performance comparison between the automated and manual UI migration, and gathering user feedback.

Chapter 5

Implementation

This chapter describes the UI's automatic migration development, starting with its requirements and culminating in the implementation. We will present the structures and auxiliary objects used to provide the necessary abstractions, and also the algorithms responsible for performing the different parts of the migration. As proof of concept, the combination of all of the algorithms will be presented as the developed automation approach.

The implementation will be executed through model manipulation using C#/.NET to operate and transform the OutSystems' model objects which are also represented as a graph of C# objects. By using C# classes, it is also possible to serialize and de-serialize a model to/from an XML file, enabling the execution of the necessary reverse engineering and transformations.

5.1 Preprocessing and Auxiliary Structures

According to the approach analysis in section 4.2, the migration algorithm must be capable of performing multiple migration approaches depending on the widgets and patterns defined. Thus, before the migration per se, the patterns must be searched in the original widget tree (with OutSystems UI elements) and the widgets must be replaced by a pattern node. This node can be linked to the nodes representing other widgets, providing an abstraction of the elements composing the pattern.

This requires a preprocessing phase and, since the OutSystems widget tree cannot support nodes besides widgets and there is no way to define patterns within the platform, additional structures are required to provide the necessary operations and abstractions. The goal was to devise and implement structures capable of storing the Traditional Web model objects' information, while allowing for patterns to be encoded and connected to

those objects.

5.1.1 Initial Widget Tree

To start the migration, the user must select a widget tree (detailed in section 5.6). This widget tree is an objects' graph from the OutSystems Traditional Web models' objects, represented by instances of a set of C# classes mentioned in section 3.3.2.

The selection is done by choosing a widget (the graph's entry point that will be the root, and its subtree will be the descendants) or by selecting a screen and its widget tree will be the selected tree (with the root as the entry point). This will be explored in section 5.6. However, the tree's selection does not influence the algorithms' performances since, in the worst-case scenario, all the widgets are covered (when there are no patterns and the migration is done at an elementary level). Figure 5.1 depicts 3 examples of widget trees from OutSystems Traditional Web Applications.

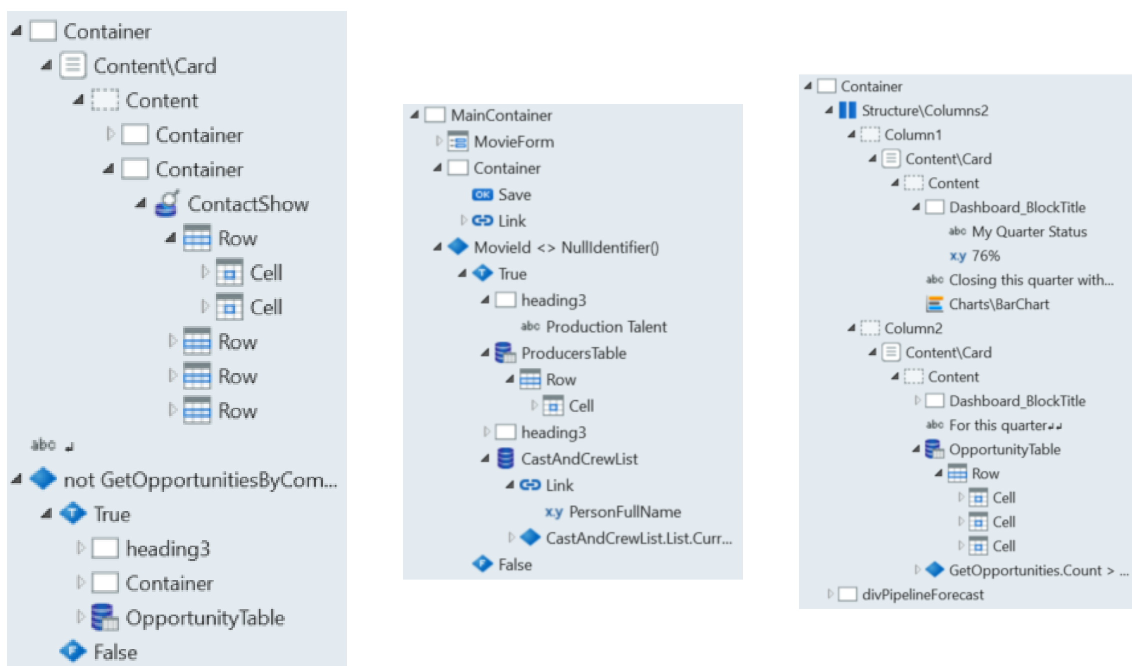


Figure 5.1: Widget Tree examples

The selected widget tree, which corresponds to the OutSystems UI elements tree, is comprised of a set of objects of the type Web Widgets and all of its descendants. This means that a widget in the tree has, as children, some properties besides its child widgets, requiring some complexity to navigate the widget elements. As an example, the Table Record widget has as children in the widget tree the following objects: Text Resources, the Header Row, the Data Row, its Source Record List, the Message to present in case there are no elements, the Line Count property, and the Start Index property. Out of all of these children, only the Header Row and the Data Row are children widgets (the rest

are properties and values associated with the widget). Nonetheless, all of the children are C# objects instancing OutSystems Traditional Web model elements.

Besides, the widget tree can only store nodes which are OutSystems elements (defined in the model), so it becomes impossible to, after finding the patterns, store them in the tree and connect them to other nodes. These patterns are a necessary component of this project and we must be able to integrate them in a tree. This requires the implementation of nodes capable of representing the patterns and elementary widgets (not every widget will belong to a pattern) without changing the behavior and structure of the original widget tree (the UI structure and elements must be migrated with as much similarity to the original widgets and structure as possible).

So, the widget tree contains all of the widgets but also other properties that, despite being necessary to the migration, add an unnecessary complexity layer to the widget tree navigation and pattern search. Not only that, but patterns cannot be represented as objects inserted in that same tree. As a solution, we opted by defining new structures, including a special tree capable of representing the widgets (model objects) as nodes, but also the pattern nodes (objects necessary for the migration but not represented in the model), as well as connecting these two types of nodes.

5.1.2 Abstracted Tree

As a solution to the widget tree's problems, a new structure was created to provide a representation of the widget tree capable of searching, storing, and connecting patterns to the nodes representing widgets. Also, this new structure provides an abstraction of the widget tree, reducing the accidental (and unnecessary) complexity when browsing and traversing it. We will call it Abstracted Tree.

Thus, as stated in the name, the Abstracted Tree is a data structure shaped like a tree, representing the hierarchic relations of the OutSystems UI widgets created in a Screen or Web Block. Its nodes can be of two types: Regular Tree Nodes and Pattern Nodes. Figure 5.2b shows an Abstracted tree corresponding to the widget tree seen in figure 5.2a without the pattern nodes. Figure 5.2c, shows the same Abstracted tree but with the patterns defined and bound to the rest of the tree nodes (which represent widgets).

This tree saves the root, the number of nodes belonging to the tree, and the logs documenting the tree migration. The abstracted tree may encompass two types of nodes:

- Tree node

  A type of node to encode a regular widget from an OutSystems Traditional Web screen. It has the following data: the widget associated and its properties, the parent tree node, the children tree nodes, the tree where it is inserted, and an ID to identify the widget conversion in the migration logs.

- Pattern node

A type of node that extends the tree node to encode a pattern (a set of elements combined in a certain way, as explained in section 5.1.3). It contains the same data as a Tree Node, and also the nodes abstracted by the pattern, as well as the pattern type (which contains the methods necessary to search, connect and migrate the pattern).

Both these types of nodes are implemented using C#, the language used to perform the direct model manipulation (since the already existing objects are also represented using this language, hence facilitating the manipulation and operations with the existent objects).

Using these two types of nodes and additional data, it is possible to create an abstracted tree that can represent OutSystems UI widgets and patterns found. This provides a way to easily navigate, and consequently migrate the widget tree, now abstracted. Additionally, some widgets not pertinent to the migration are not represented by the abstracted tree but its value is not lost (as an example, there are many types of Placeholder widgets, some that need to be represented in the abstracted tree, and others that do not - e.g. those used to store the children of a widget). The purpose and application of the abstracted tree will be further detailed in the section detailing the solution (5.1.3.2).

5.1.3 Patterns

When migrating a widget tree, the simplest way to perform the conversions is to transform a Traditional Web widget (UI element) into the equivalent Reactive Web implementation. However, the importance of identifying known structures and combinations of UI elements and migrating them as a whole was previously explained in chapter 4.

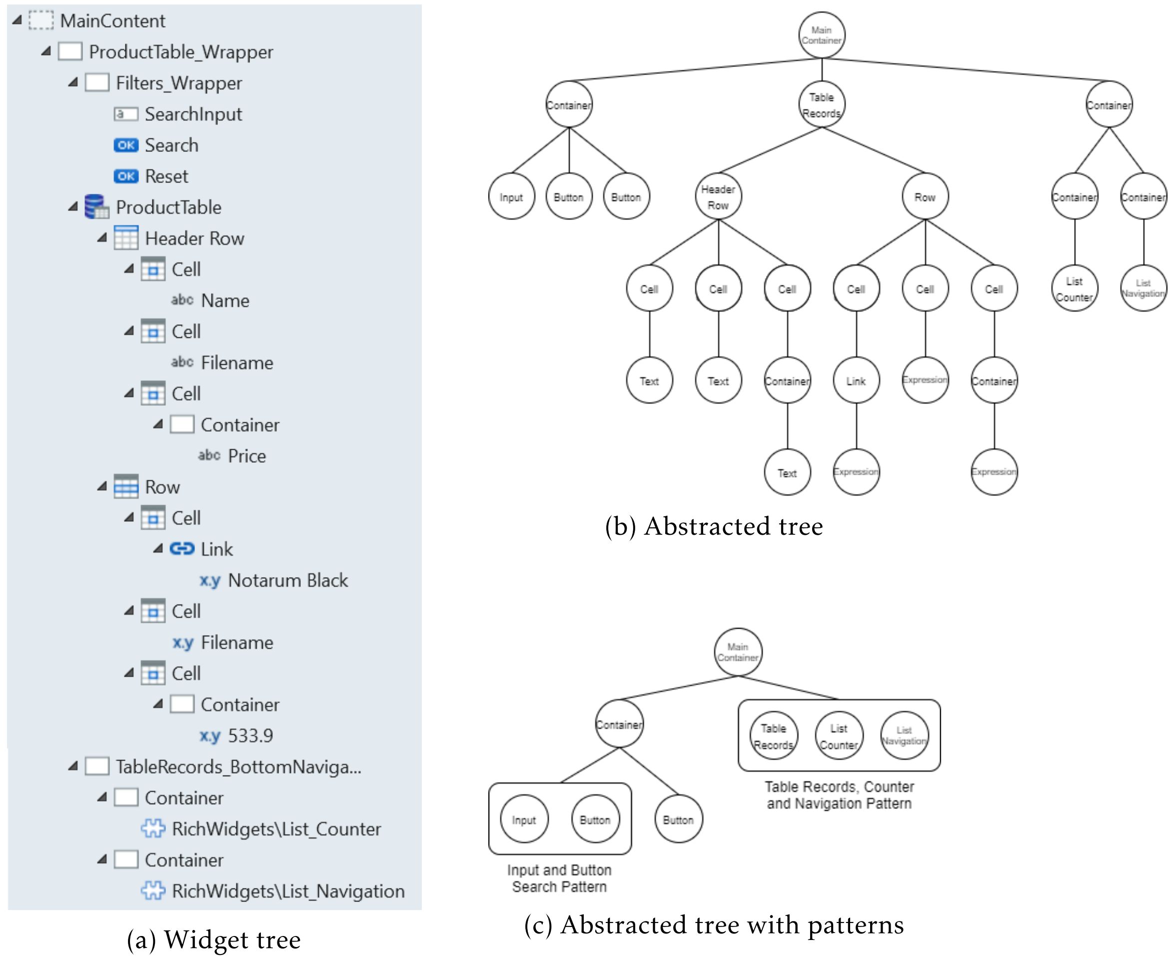
Therefore, a different way was proposed and consists of migrating the UI elements by searching and recreating them as known patterns in the new paradigm according to its best practices. This helps the migration approach accomplishing the best implementation in accordance with the Reactive Web guidelines, as the objective is to automate the best possible migration, and not simply reconstructing the widgets. This process consists of defining the patterns as C# classes and performing a search for instances of those classes in the abstracted tree.

Be that as it may, before explaining the functioning of the pattern definition and search, we ought to examine when and why the patterns should be used.

5.1.3.1 Patterns Motivations

Thus, a pattern needs to be defined and searched in the abstracted tree, and there are multiple reasons to support that. Some of which are:

- The individual migration (simply recreating the widgets in the Reactive Web application) would not produce a result according to the Reactive Web paradigm's best practices.



(a) Widget tree

(b) Abstracted tree

(c) Abstracted tree with patterns

Figure 5.2: Widget Tree representations

- The widget being migrated is from a library that does not exist in the Reactive Web paradigm (e.g. the library's widgets are now implemented as regular widgets).
- The widget migrated must be used differently in a Reactive Web application. This is the case where the usage of the widget has changed and is now implemented, for example, as a child of another widget, or it has different properties and/or values.
- Two widgets of the old paradigm being migrated are merged into a Reactive Web widget. If the functionality of two widgets of the Traditional Web paradigm is implemented by a single widget in the new paradigm, the properties of those widgets will have to be converted to replicate the UI components.
- Other specific situations. E.g. a Traditional Web widget's property which relates to a sibling widget is implemented in a sibling widget in the Reactive Web paradigm.

To define these particular scenarios where the patterns present a better way to perform a migration, an analysis of the different paradigms was made. The scenarios were detailed alongside some of the people responsible for developing the new paradigm, as well as

people in direct contact with multiple clients performing manual migrations (some of which mentioned in sections 4.1.2 and 4.3).

To illustrate this and prove the viability and application of these patterns, many were defined and some were implemented in the migration tool as a proof of concept.

5.1.3.2 Solution

A pattern is defined by its type, which contains the methods to search it, insert it in the abstracted tree (abstracting the tree nodes that compose it), and migrating it. Each type corresponds to a pattern and has to be defined by an internal OutSystems' Software Engineer with knowledge of the migration tool and the pattern to be created. This is necessary to enable the operations of the models' objects using direct model engineering and manipulation.

When a pattern type is created, 3 things must be set as functions:

- How is the pattern searched?

  This function receives a node (model object) and can access its parent (and other ascendant nodes), its children and respective subtree, and different properties to verify if the node belongs to this pattern type. If, for example, the pattern depends on whether a widget has a property with a certain value, and that value corresponds to the key of a sibling widget, that is checked in this function. If and only if the pattern is matched to this node and associated values, the next function is called and the result returns true. Detailed in section 5.2.2.

- How is the pattern represented in the abstracted tree?

  When a pattern type has a match, changes must be done in the abstracted tree to abstract the nodes constituting the pattern. This function is called in that eventuality, and performs a set of changes in the abstracted tree: creates a pattern node with the necessary data (such as the nodes to be abstracted), removes the nodes to be abstracted from the abstracted tree, and inserts the pattern node in the abstracted tree making the necessary rewiring of the nodes' connections. Detailed in section 5.2.3.

- How is the pattern migrated?

  When migrating a pattern instance, it may contain multiple widgets with important properties. To migrate these, the conversion may use the individual migration functions, while adding extra operations and complexity to the migration (variables setting, new widgets creation, etc.). Detailed in section 5.2.4.

By defining the first function (responsible for returning whether or not the node and associated values are a match for the pattern type), it is possible to loosen or tighten its requirements at will. In other words, the matching can be as restricted as the developer

of the pattern wants, since it is possible to change the function to increase the correlation needed to return a match. As an example, a pattern can be a match simply if the node is a container and has a child link widget. Or, as an alternative, to return a match the link widget needs to have a property with a certain value, or even a sibling widget detailed in the function. This is specified by the pattern type used and allows the developer to create different patterns with different levels of granularity.

Additionally, the definition of a pattern type is done separated from the migration tool implementation, with that type being searched and migrated. In fact, the migration tool does not need to be changed every time a pattern is added or removed from the search algorithm. This is possible using a file containing which pattern types will be used in the current migration process and using reflection in the tool implementation to develop the search for the types listed. These reflection-based calls of the pattern types guarantee that there is no need to modify the tool code due to changes in the list of patterns to be migrated.

After the first function identifies a match for the pattern type, the second function is responsible for representing the pattern in the abstracted tree. For that to happen, an instance of the pattern type is created as a pattern node object and associated with the pattern type. So, as stated in 5.1.2, an instance of the pattern has the nodes it abstracts, and since it is associated with the pattern type, it can use the type's migration function (third function) to perform the necessary conversions (model transformations) using the pattern node's saved data. Besides, the pattern node contains the abstracted node's subtree and consequently, its children. Not only that, but a pattern node can have as an abstracted node another pattern node, allowing for patterns to be nested (stored inside other patterns), which lets the program support the structuring and organization of complex patterns. Other metadata is stored inside the pattern to be later used in the migration logging (section 5.4.1).

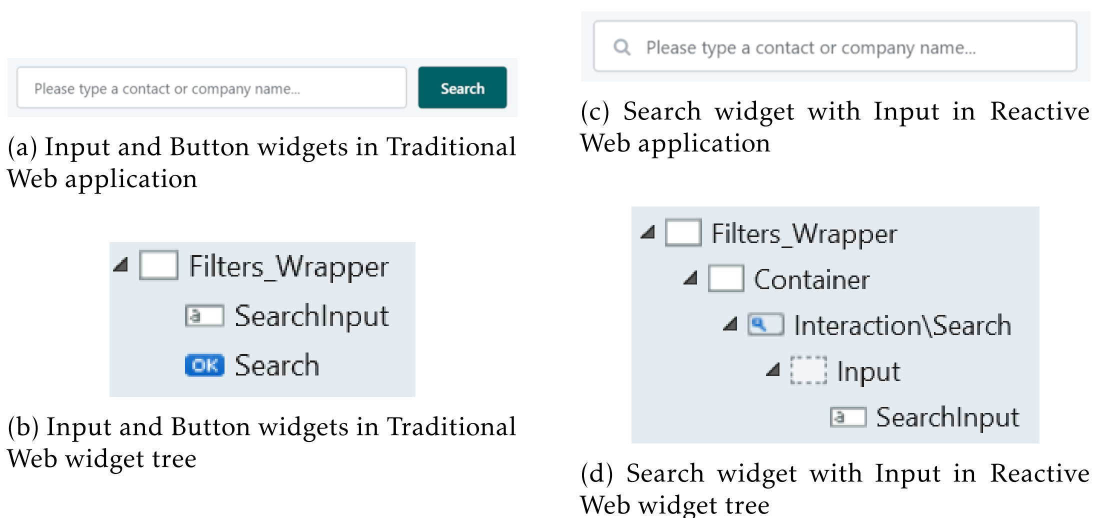
These details and specification of both the pattern types and pattern instances (represented as pattern nodes which are C# objects that can be manipulated using the same language), capacitates the pattern usage and makes it a rich and useful functionality of the migration.

5.1.3.3 Examples

A UI pattern can be as simple or as complex as we want. To sum up what was previously explained, patterns are used to migrate certain sets of Traditional Web widgets with relations among them which, if migrated via a direct transformation, would not present a result conforming to Reactive Web best practices. In section 5.1.3.1, the multiple motives that support the patterns' implementation and migration were detailed, and to better illustrate what the patterns are (and when they are needed) some examples will be presented.

An example of a pattern, perhaps the most elementary, is the combination of an Input

widget and a Button widget used for search. This can be used for multiple purposes, like searching elements in a table given a string. As depicted in figure 5.3a, in the Traditional Web paradigm, the user writes something in the input and clicks the search button to search the elements based on the keyword inserted, which refreshes the screen accordingly. In the Reactive Web paradigm, there is no need for a Search button, due to its rendering based on React. When the user writes something on the input, the necessary screen elements are rendered according to the keyword inserted, and each time the user changes the string, the elements are rendered automatically (without the need to click a button, as seen in figure 5.3c). So, if migrated directly as implemented, the migration would create an input and button widget, but according to Reactive Web best practices, the same functionality should be implemented with an Interaction/Search widget, with an Input child widget (figure 5.3).



(a) Input and Button widgets in Traditional Web application

(b) Input and Button widgets in Traditional Web widget tree

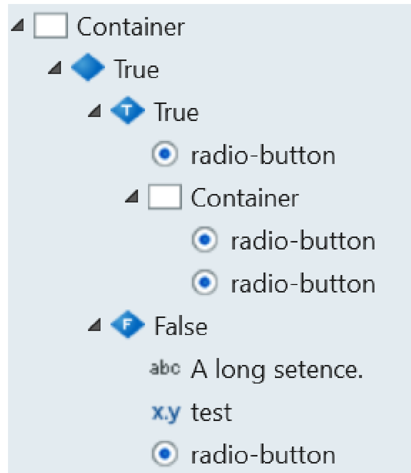
(c) Search widget with Input in Reactive Web application

(d) Search widget with Input in Reactive Web widget tree

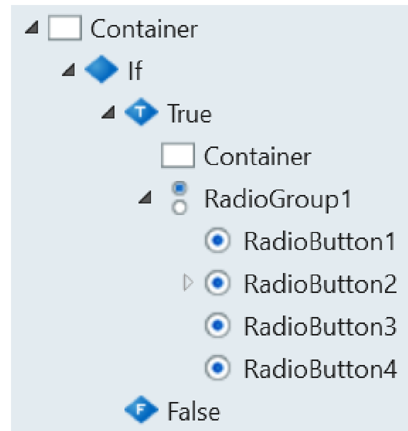
Figure 5.3: Differences in Search widgets implementation between paradigms

Another example of widgets that may constitute a pattern is a set of widgets of type Radio Button. In the Traditional Web paradigm, the Radio Button widget has a property where the associated variable is specified (a Radio Button alters the value of a certain variable). In that paradigm, a widget tree can have multiple Radio Buttons in different positions, each with their variable, and changing the variable's value when interacted with (figure 5.4a and 5.4b). In Reactive Web, however, there is a change in the widget's behavior and implementation. In this paradigm, the Radio Button widget does not have a property to specify the associated variable, but instead, all of the Radio Buttons related to a certain variable are inside a Radio Button Group, which has a property to specify the variable (figure 5.4c and 5.4d). If migrated directly, each Radio Button would be migrated to the new paradigm as an individual widget. Nonetheless, its variable property would not be able to be migrated. Hence, according to the Reactive Web paradigm, a pattern can be defined and searched, allowing for all of the occurrences of a Radio Button with the same variable to be migrated as a pattern, and into a Radio Button group with the

same variable specified as a property.



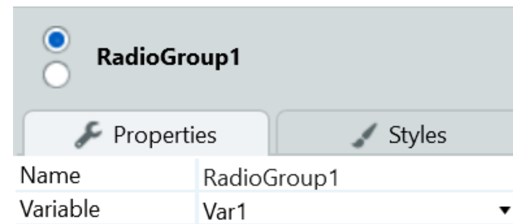
(a) Radio Button widgets across the Traditional Web widget tree



(c) Radio Button widgets grouped inside a Radio Button Group in the Reactive Web widget tree

Variable	Var1
Variable	Var1
Variable	Var1
Variable	Var1

(b) Variable values in the different Radio Button widgets



(d) Radio Button Group's variable value

Figure 5.4: Differences in Radio Button implementation between paradigms

In section 5.1.3.1, the scenario where the functionalities of two widgets are implemented by a single widget in the new Paradigm was referred to as a motivation for creating a pattern. This is the case of a table Counter and Navigation. In Traditional Web, a Table widget could have associated two types of widgets: a Counter widget and a Navigation widget (figure 5.5a). These widgets were part of a UI library and extended the functionalities of a table to facilitate its interaction, understanding, and navigation. In Reactive Web, the functionality of these two widgets is implemented by a single widget: the Pagination widget from a UI library (figure 5.5c). Thus, if migrated individually, it would not be possible to migrate the Counter and Navigation widgets. However, as a pattern, one can find the set of elements containing the Table, the Counter, and Navigation widgets, and migrate them as a pattern. This results in the implementation of the Pagination widget, its relation to the migrated Table widget, and the mapping of the properties from the Counter and Navigation to the properties of the Pagination Widget.

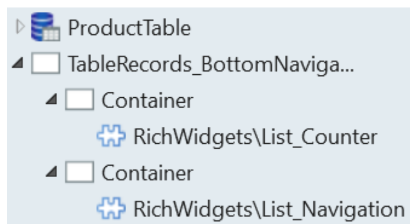
5.1.3.4 Potential

The theoretical work regarding the patterns and its possibilities allows us to consider its potential beyond what was presented so far. In the patterns' motivation section (5.1.3.1),

Name	Description	Price
Notarum Black	Powerful and reliable, this 15" HD laptop will not let you down. 256GB SSD storage, latest gen.	533.9
Notarum Black	Powerful and reliable, this 15" HD laptop will not let you down. 256GB SSD storage, latest gen.	533.9
Notarum Black	Powerful and reliable, this 15" HD laptop will not let you down. 256GB SSD storage, latest gen.	533.9

51 to 100 of 721 records ‹ ... 8 9 10 11 12 ›

(a) Table, Counter and Navigation in Traditional Web widget tree

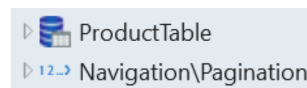


(b) Table, Counter and Navigation in Traditional Web application

Name	Description	Price
Notarum Black	Powerful and reliable, this 15" HD laptop will not let you down. 256GB SSD storage, latest gen.	533.9
Notarum Black	Powerful and reliable, this 15" HD laptop will not let you down. 256GB SSD storage, latest gen.	533.9
Notarum Black	Powerful and reliable, this 15" HD laptop will not let you down. 256GB SSD storage, latest gen.	533.9

1 to 10 of 15 items ‹ 1 ... 50 ›

(c) Table and Pagination in Reactive Web application



(d) Table and Pagination in Reactive Web widget tree

Figure 5.5: Differences in Table and Pagination implementation between paradigms

the cases presented were common to all of the Traditional Web applications, regardless of the client. However, since the patterns' search uses an external file and reflection (the implementation does not use the patterns' definition directly, but instead creates the pattern types when present in the file), different patterns can be defined separated from the migration tool and later searched and migrated. This provides the opportunity to define different patterns for specific cases where a recurring structure is identified, and an associated pattern is created and used by the migration tool. As an example, if a client wants to migrate complex applications where there is a recurring UI elements' combination, that can be defined as a pattern to further automate the migration. This can be used in multiple situations, such as if the objective is to represent a combination of elements in a distinct manner than the result of the individual migration, or if that same combination cannot be migrated to the Reactive Web paradigm with an element-by-element approach.

Another possible usage of patterns besides the standard use cases, is the opportunity to implement more than one pattern type for the same pattern, with different levels of demand for there to be a match (more property values compared, more widgets compared, etc.). This could be useful in a scenario where different levels of matching to a pattern could produce different results in the migration, with certain automatic migration decisions varying accordingly.

Also, despite not being implemented in the proof of concept, the possibility of giving the developer using the tool, the possibility to choose how the migration is conducted when a pattern is found in some widgets (individually or via pattern-driven migration)

was studied. This could be done when a pattern type matches to a certain node or set of nodes, by giving the developer the choice to abstract the nodes composing the pattern or deciding to keep representing the widgets as individual objects (in the first function, giving the user the choice whether or not the second function is called). In case the user chooses to use the pattern-driven migration when there is a match, an instance of the pattern is created as a pattern node. Otherwise, the tree nodes representing widgets are maintained in the widget tree and migrated sequentially. As previously mentioned, the tool as it is does not support this functionality, but the algorithms are prepared to easily integrate it into future works.

5.2 Alghorithms

The tool consists of a set of algorithms responsible for the different segments of the migration. In this section, we will detail those algorithms and their operations step by step. To better understand how it works, it is advised the reading of section 5.1, which explains the functioning of the auxiliary structures used to perform the procedures.

In general terms, the migration is comprised of the following steps (some of which depicted in figure 5.6), implemented as algorithms:

1. Tree Abstraction
2. Pattern Type Search
    a) Pattern Instance Creation
    b) Pattern Instance Insertion
3. Model Transformations
    a) Direct Transformations
    b) Pattern Driven Transformations
4. Communicating the migration information

Step 2 is repeated for every pattern type indicated in the file previously mentioned, and steps 2a) and 2b) only execute when there is a match in 2. The migration (step 3) is performed for every node of the migrated tree, but only one of the steps 3a) or 3b) is executed for a certain node (they are alternatives depending on the node type). Step 4 consists of printing the information gathered during the algorithms' execution, such as performance metrics and operations' logs (what was successfully migrated, what was not, what is there left to check/manually fix, and why).

Figure 5.6 shows a diagram with the different processes that constitute the migration, as well as the initial and final state. The UI objects are in-memory representations of the OutSystems UI models which are accessible as a graph of C# objects when editing an app in Service Studio. The migration (and all of its phases) will be responsible for transforming the OutSystems Traditional Web UI Objects in Reactive Web Objects that represent the Reactive Web Models (and conform to the OutSystems Meta-Models). The

diagrams corresponding to the steps represented as boxes will be detailed in the next sections.

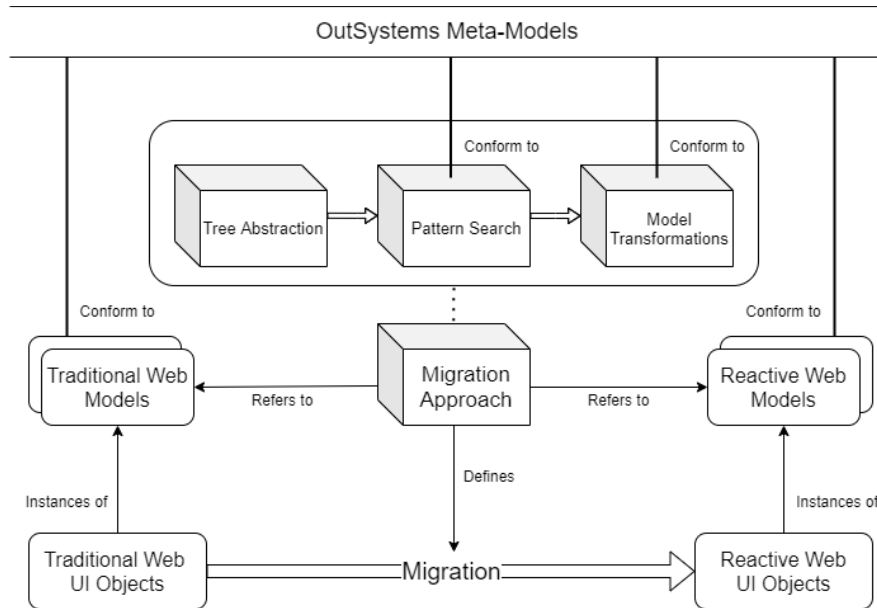


Figure 5.6: Overview of the migration architecture and its different processes.

Bellow, every step will be detailed to give the reader the knowledge to understand the tool's behavior. The algorithms' breakdown will not include the widget tree selection (that will be analyzed in section 5.6), instead, it will start by assuming the widget tree has already been selected and we have the root node.

5.2.1 Tree Abstraction

The tree abstraction algorithm receives as an input a Web Widget object from OutSystems' UI. From that object, it is possible to access the widget tree by navigating the widgets' children of type AbstractWidget from the OutSystems Traditional Web model. As formerly mentioned, these objects in memory are all represented as C# class instances, so the same language is used to access and manipulate the models.

The algorithm's objective is, for every widget of the selected tree, to create a Tree Node in the abstracted tree containing the following data: the widget object (to later access its properties and values), its parent tree node, its children tree nodes, the tree object where it is inserted, and an ID. When created, a tree node needs to be given its parent node, but the children can be created later and added to the node. Thus, to create every node of the abstracted tree, the widget tree is covered using pre-order traversal. This is because when creating a tree node from a widget, the tree node representing its parent has to be created and inserted in the abstracted tree. The algorithm presents a recursive behavior, where a node is created from a widget, and the function is called for its children. Figure 5.2 depicts a widget tree represented in an OutSystems application and the corresponding abstracted tree.

It was formerly stated that the OutSystems widget tree contains multiple values besides the widgets (Web Widgets and all of its descendants, which can be properties and values). So, the objective is to have the nodes only representing the widgets from a Traditional Web widget tree, making the abstracted tree easier to navigate and search. Through reverse engineering, the necessary values are extracted from the widgets and saved to later be encoded in the abstracted tree. When the abstracted tree is created, the nodes corresponding to the widgets are also created and store, for each widget, its value, its parent, its children, and its properties, among other values.

Alghorithm 1 details the tree abstraction process.

Algorithm 1: Tree Abstraction Alghorithm

```
   Input: A Widget widget and a Tree Node parent
1  Process AbstractTree(widget, parent):
2      node ← new TreeNode(widget, parent, tree);
3      if parent ≠ null then
4          parent.addChild(node);
5      foreach descendant in widget.DirectChildren WHERE descendant.type is
         ABSTRACTWIDGET) do
6          Call AbstractTree(descendant, node);
```

5.2.2 Pattern Type Search

The pattern search consists of looking for occurrences of a set of pattern types in the abstracted tree. These pattern types were detailed in section 5.1.3 and their functions will be used to search and migrate the pattern instances.

The pattern types to be searched are listed in an auxiliary file and are searched by the order they are listed in. This is important since the search order can influence the patterns found, for example, by searching more general patterns before more restricted patterns. Also, some patterns have other patterns nested and only match when those are in the tree, which affects the order in which the patterns are searched. To illustrate this need to order the patterns correctly, we can look at the case where we have two pattern types, one where it returns a match if a container node has as children an input widget and a label (1), and another where a container node has as children an input widget, a label and an expression widget (2). If there are two occurrences of the pattern (2) and one occurrence of the pattern (1) - without the expression widget - we want to identify those scenarios. However, if the types in the file are ordered so that pattern (1) is searched before pattern (2), we will have three matches for (1) and no match for (2). This is depicted in figures 5.7 and 5.8.

So, following the order of types listed in the file, an object corresponding to the pattern type is created for every pattern and it is searched in the abstracted tree. The creation of

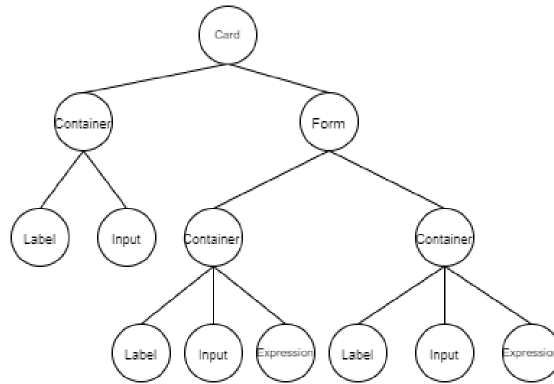


Figure 5.7: Example of an abstracted tree

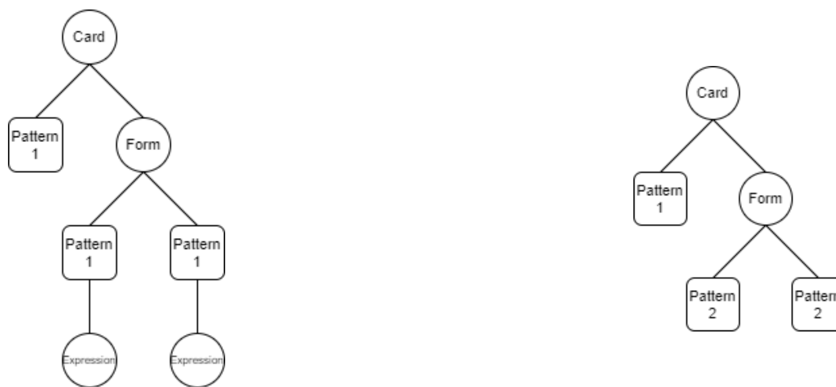


Figure 5.8: Abstracted tree with patterns varying according to the pattern search order

the object is implemented using a reflection-based call using the Activator[1] class from the .NET C# language, where the pattern type name is inferred from the string in the file. When created, the instances of a pattern type are searched using the function detailed in section 5.1.3.2. Algorithm 2 depicts the Pattern Types creation and search.

The search for a pattern type is implemented by a match function and searches it in every node of the tree using a post-order traversal. This means that before searching the pattern type in a node, the pattern type is searched in its children. This is because when a pattern is found, the nodes that are part of it are removed from the tree (and so are their subtrees) and a pattern node is inserted. So, if a pattern occurs in a node and one of its descendants, it is better to first find the pattern in the descendant node before removing the node and its subtree from the abstracted tree and inserting a pattern node. Thus, the match function from the pattern type is called for the root, but for every node, the function searches the pattern in its descendants before searching in it. Ergo, every node ends up covered by the function.

The pattern match function for the pattern containing an Input and Label widgets related to one another can be seen in algorithm 3. A function of this type is implemented for each pattern built, to make it possible to search it in the abstracted tree.

[1] Available at https://docs.microsoft.com/en-us/dotnet/api/system.activator

When a pattern type has a match in a node, an instance of that pattern is created and inserted in the widget tree as a pattern node, which will be detailed in the coming up section. After a pattern type is created, searched and all of its instances are inserted in the widget tree, the next pattern type follows: the next line of the file is read and the process is repeated for the next pattern type.

Bellow, in figure 5.9, a diagram illustrates the processes of abstracting the widget tree, and searching, creating, and inserting the patterns.

Algorithm 2: General Pattern Type Search Alghorithm

Input : The Abstracted Tree t representing the tree and a list of patterns lp indicating which patterns to search

```
1 Procedure PatternTypeSearch(t, lp):
2     foreach p in lp do
3         patternType ← Type.GetType(p);
4         pattern ← Activator.CreateInstance(patternType);
5         if pattern is instace of PatternMigrator then
6             Call pattern.Match(t.root));
```

5.2.3 Pattern Creation and Insertion

This operation is made after the pattern type is matched to a specific set of nodes and its properties. It is implemented by the object representing the pattern type instance and behaves accordingly, making the necessary transformations to remove the nodes and abstract them as a pattern node.

Thus, after the pattern type finds a match for the pattern in a set of nodes, a function responsible for creating the pattern instance is called. Each pattern type has a different implementation for that function depending on the nodes that make up the pattern and necessary transformations. The function has the following steps:

1. The nodes' information is obtained

   This function receives as parameters the nodes to be abstracted. The first step is to obtain the tree object they belong to, as well as the parent of every node, to perform the next operation.

2. A new pattern node is created

   An instance of the pattern is created receiving as arguments the tree, a tree node to be its parent, the associated pattern type, and a name. The tree will be used to store the migration logs and children nodes, and the node received will be set as the pattern node's parent node. Concerning the pattern type, it is stored inside the pattern node with the purpose that the node is correlated with a migration function characteristic of the pattern type.

Algorithm 3: Match function for Label and Input pattern

Result: true if the tree node matches the pattern type, **false** otherwise
Input : The Tree Node *node* representing the tree node where the pattern is searched

```
1  Function Match(node):
2      foreach child in node.children do
3          Call Match(child);
4      widget ← node.widget;
5      if widget ≠ null AND widget.type is LABEL then
6          if node.parent ≠ null then
7              inputSiblings ← new List;
8              foreach sibling in n.parent.children do
9                  if widget.type is INPUT then
10                     inputSiblings.add(sibling);
11             foreach s in inputSiblings do
12                 input ← s.widget;
13                 refs ← input.Referers;
14                 if refs.contains(x) WHERE x.parent.key = widget.key then
15                     Call insertPatternNode(node, s);
16                     return true;
17     return false;
```

3. The nodes are stored inside the pattern

   The nodes that are now represented by the pattern (the UI web widgets that compose the pattern), are saved inside the pattern in a structure with an associated ID. The nodes' values (correlated widgets) will be crucial when migrating the pattern, seeing that the values and properties will be necessary to create the equivalent implementations in the new paradigm.

4. The nodes are removed from the tree

   The nodes are removed since the pattern will be inserted in the tree. This is done in the interest that the combination of nodes is represented and migrated as a pattern, instead of as individual widgets. To remove the nodes, a function of its parents is called to remove the nodes (its children). This way, the next time the abstracted tree is covered, these nodes will not be covered since they are not any nodes' children. However, they will maintain their values and previous relations saved in the pattern node.

5. The tree connections are altered

After having the pattern node created, it is inserted in the tree as a child of a node (typically, the parent of one of the abstracted, and now removed, nodes). Furthermore, some of the children of the abstracted nodes may be kept in the tree and not abstracted. For this to happen, these children must be linked as children of the new pattern that now abstracts its original parent. The linking is done by setting the pattern node as its parent, and the nodes as children of the pattern.

After these steps are repeated for every pattern (with each node in the abstracted tree searched for each pattern type), the abstracted tree contains the pattern nodes abstracting the removed tree nodes. A simple example of this process is detailed in algorithm 4, which is called in algorithm 3 when there is a match.

Algorithm 4: Pattern Creation and Insertion for Label and Input pattern

Input : The Tree Node *label* representing the node abstracting the label widget and the Tree Node *input* representing the node abstracting the input widget

1 **Process** `InserPatternNode`(*label*, *input*)**:**
2 $parent \leftarrow label.parent$;
3 $tree \leftarrow label.tree$;
4 $patternNode \leftarrow new$ PatternNode($"Label\&Input", parent, Migrator, tree$);
5 $parent.addChild(patternNode)$;
6 $patternNode.encodeSpecialNode("Label", label)$;
7 $patternNode.encodeSpecialNode("Input", input)$;
8 $parent.removeChild(label)$;
9 $parent.removeChild(input)$;

Figure 5.9 shows the different steps involved in the tree abstraction, pattern search, and pattern creation (and insertion) in the widget tree. The elements selection will be approached in section 5.6.

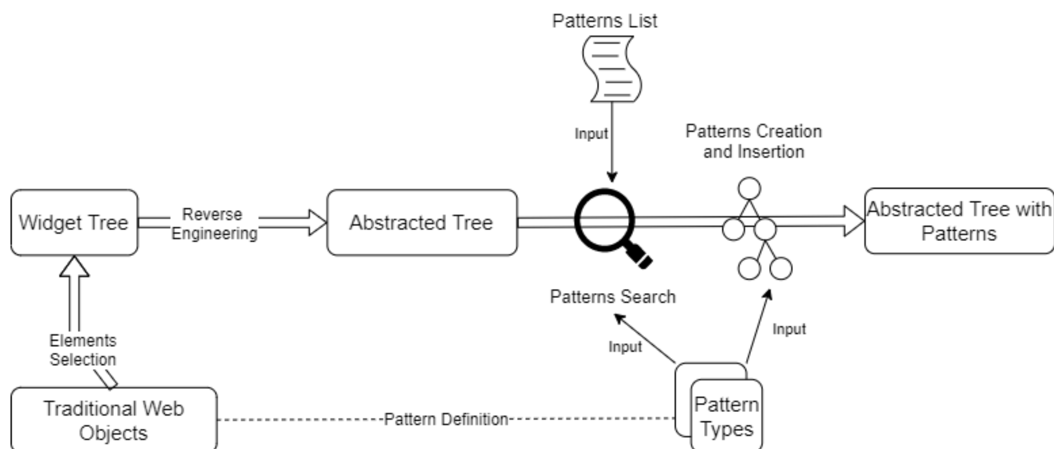


Figure 5.9: Tree Abstraction and Pattern Search processes.

5.2.4 Transformations

Succeeding the tree abstraction and pattern search (and subsequential insertion), the transformations can be applied to the abstracted widget tree elements.

The migration will proceed using the following approach: the process will be executed for the tree root node, where a function is responsible for identifying the node type and content, and calling the specific function responsible for migrating it. After the migration of the node to the tree in the new paradigm, the function responsible for identifying a node will be called for the (migrated) node's children with a placeholder of the (migrated) node as the target parent. Figure 5.10 shows a diagram to illustrate such process.

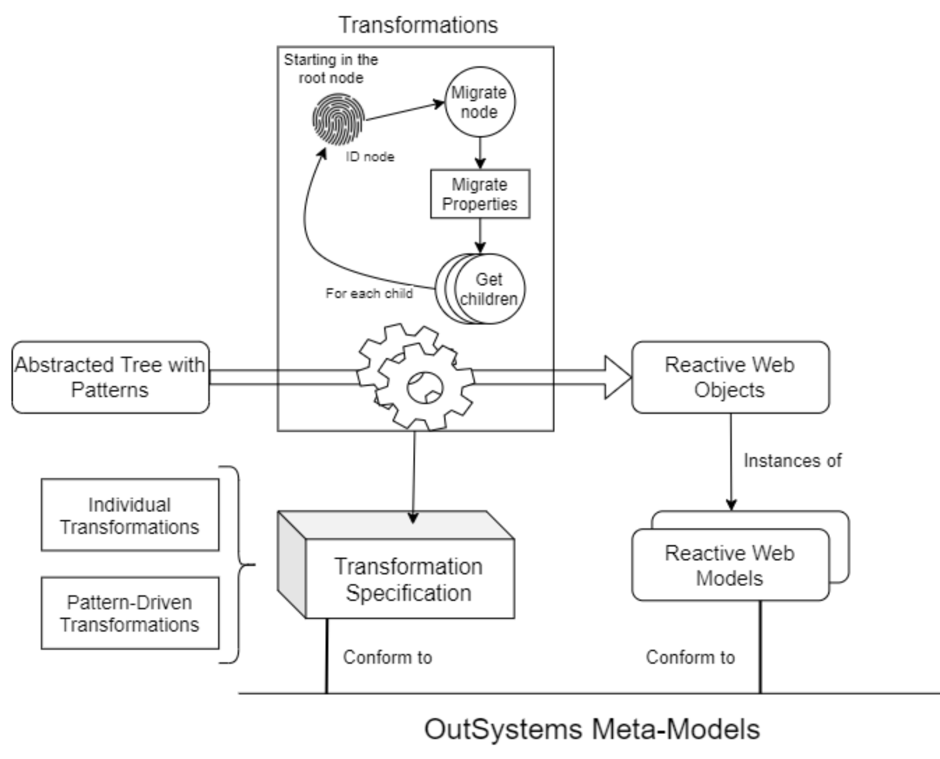


Figure 5.10: Transformations architecture

Depending on the node, the transformation for a Reactive Web element can differ in many aspects. Essentially, there are two types of transformations: Direct transformations, called for tree nodes representing a widget, and Pattern-Driven transformations, called for pattern nodes. These categories of transformations will be detailed in the coming up sections.

The transformations, both direct and pattern-driven, follow transformation specifications conforming to the OutSystems meta-models. These transformations are implemented in C# since this was the language chosen to perform the model manipulations due to the nature of the OutSystems objects when being processed/edited. To create an equivalent implementation of a widget or element, a process of reverse engineering is necessary where the property values are extracted and its values will be attributed to

the properties of the equivalent implementation (with or without changes). After that, forward engineering will be used to create the equivalent implementation, which highly depends on the node type and content.

5.2.4.1 Direct Transformations

The direct transformations are the transformations called for tree nodes representing Traditional Web widgets. These transformations are called direct because for a certain widget as input, the output is the Reactive Web equivalent implementation. This process includes the steps:

1. Identifying the Traditional Web widget in the tree node.

   The first step in the migration of a node is the identification of the associated Web Widget (since different widgets are migrated differently). When that information is retrieved using the widget type, the program calls the function responsible for migrating the widget, which receives the node and where to migrate the widget to (its Reactive Web parent).

2. Creating an equivalent implementation in the Reactive Web application.

   In a widget's migration function, the Web Widget's type is identified and an equivalent implementation (both in aspect and functionality) is created on the target of the migration (its parent, a Reactive Web object received as a parameter). The implementation can vary from a single widget to a set of widgets (when, for example, a similar widget does not exist in the Reactive Web paradigm).

3. Extracting the properties and values of the original Traditional Web widget.

   This is a process of reverse engineering where the widget's properties and CSS values are obtained to be later replicated. Since every widget has different properties and CSS attributes, each function must be suited to extract the necessary widget information.

4. Assigning values to the Reactive Web widget's properties

   With the objective of migrating both the functionality and visual appearance of the original widget, it is not enough to create a similar widget in the Reactive Web application, its properties and values must also replicate the original widget. To do so, we must use the values extracted in the previous steps and assign them using objects called Descriptors (defined in the OutSystems meta-models and models) which allow the function to change and assign the desired values.

5. The current node's children (whether a tree or pattern nodes) are migrated

   After migrating a node and respective widget, the algorithm presents a recursive behavior and calls the general migration function (step 1) for each children node

with the respective parent (e.g. a placeholder of the node) as a parameter. This makes the reconstruction of the widget tree possible.

So, in other words, step 1 is a general function responsible for receiving the tree node and calling the specific function responsible for migrating it. Steps 2 to 5 compose the algorithm to migrate a widget saved in the node covered at the moment. The process is repeated until the entirety of the tree is migrated.

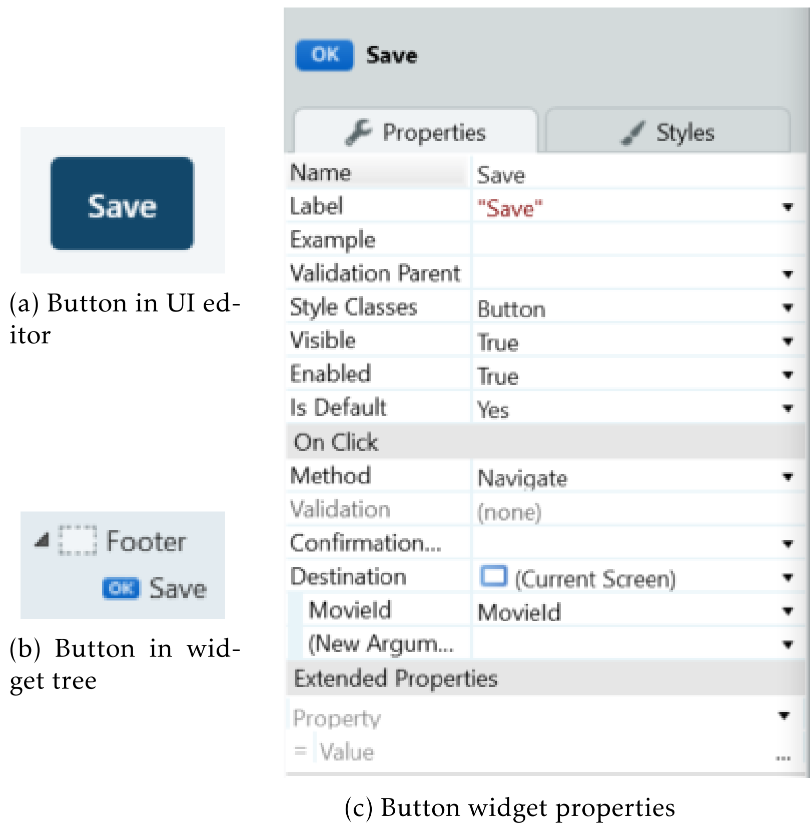
The functions responsible for migrating the widgets are suited for each widget and receive as parameter the tree node to be migrated and the parent of the new Widget (created in the migration). The extracting and assigning of property values are performed in this function and uses the previously mentioned objects called Descriptors. These objects are defined in the OutSystems models and provide operations to alter some of the properties and values of a widget (one Descriptor for each widget type). This is one of the reasons to make a migration function for each widget type: to know which Descriptor to use. This makes the migration functions conform to the OutSystems meta-models. Algorithm 5 shows the function responsible for migrating a Link Widget abstracted in a tree node.

The majority of the Traditional Web widgets have a corresponding implementation in the Reactive Web paradigm. A simple example is the button widget, which is implemented in both paradigms, and where a migration consists of creating a Reactive Web button and migrating the properties and CSS values. However, the different paradigms may have differences in the implementation of the widgets, as is the case of the button widget: In the Traditional Web paradigm, the button has an associated text as a property, but in the Reactive Web, the button must have a child widget of type text or expression to represent the button text. Small differences happen in almost every widget and must be taken into account during the migration. Figures 5.11 and 5.12 show the details of the button widget in the different paradigms.

However, some Traditional Web widgets do not exist in the Reactive Web paradigm. This may happen for one of many reasons: the widget was discontinued, the widget's functionality is now implemented by another widget, the widget has no use in Reactive Web applications, etc. So, to perform the best possible migration, the process varies to better replicate the widget's aspect and functionalities. Examples of this occurrence are: The Traditional Web widget Input Password is now implemented by the Input widget in Reactive Web paradigm, by selecting the input type; the Show Record widget does not exist in the Reactive Web paradigm, so it must be implemented using HTML tags; the Edit Record widget, like the previous example, does not exist in the Reactive Web paradigm, and it must be implemented using a Form widget with HTML tags. These examples, along with other specific cases, make the migration result in the implementation of certain widgets with different elements to replicate the aspect and functionality. However, the aspect may end up slightly different than the original widget. Some of these occurrences can be observed in figures 5.13 and 5.14.

Algorithm 5: Migration function for node representing Link Widget

```
Result: Reactive Web Widget representing the migrated Traditional Web Widget
Input :The Tree Node node representing the Link widget and a Reactive Web
       widget parent representing the migration target (the parent of the
       migrated widget)
1  Function MigrateLink(node, parent):
2      oldLink ← node.Widget as LINK;
3      newLink ← WidgetFactory.CreateWidget(parent, LINK);
4      descriptor ← newLink.Descriptor;
5      Call MapExtendedProperties(oldLink, newLink);
6      if oldLink.Name ≠ null then
7          newLink.Name ← oldLink.Name;
8      if oldLink.Style ≠ null then
9          descriptor.setStyle(newLink, oldLink.Style);
10     descriptor.setHeight(newLink, oldLink.Height);
11     descriptor.setWidth(newLink, oldLink.Width);
12     if oldLink.Title ≠ null then
13         titleProperty ← new ExtendedProperty(newLink);
14         titleProperty.AttributeName ← "Title";
15         titleProperty.Value ← oldLink.Title.DisplayName
16     if oldLink.Enabled ≠ null then
17         newLink.Enabled ← oldLink.Enabled.Value;
18     if oldLink.Visible ≠ null then
19         newLink.Visible ← oldLink.Visible.Value;
20     if oldLink.OnClick.ConfirmationMessage ≠ null then
21         newLink.ConfirmationMessage ←
             oldLink.OnClick.ConfirmationMessage.Value;
22     textChild ← newLink.GetDescendantsOfType(TEXT).Single();
23     textChild.Delete()
24     foreach child in node.Children do
25         Call Migrate(child, newLink.Placeholders.First().AsAbstractObject());
26     return newLink.AsAbstractObject();
```

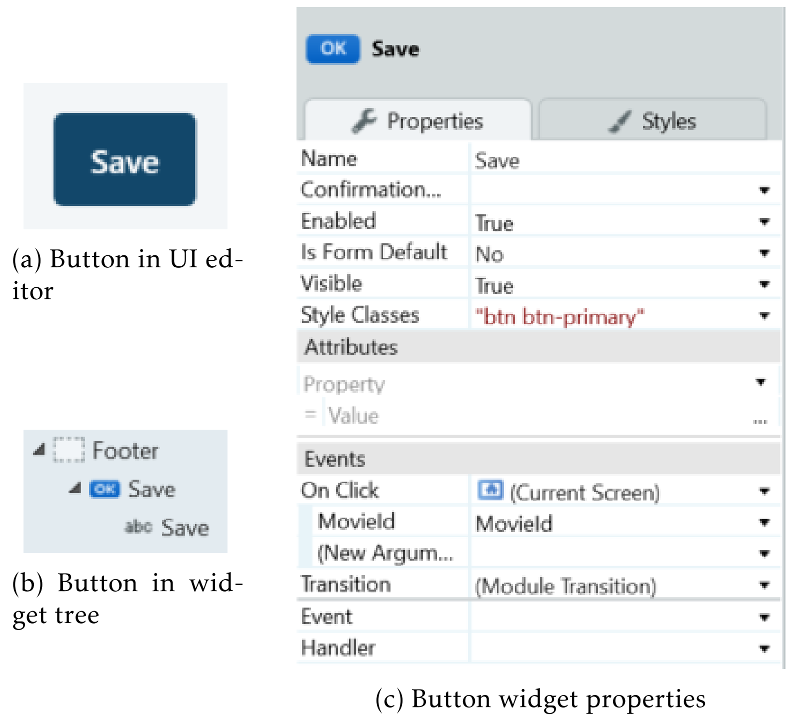


(a) Button in UI editor

(b) Button in widget tree

(c) Button widget properties

Figure 5.11: Button Widget in Traditional Web application

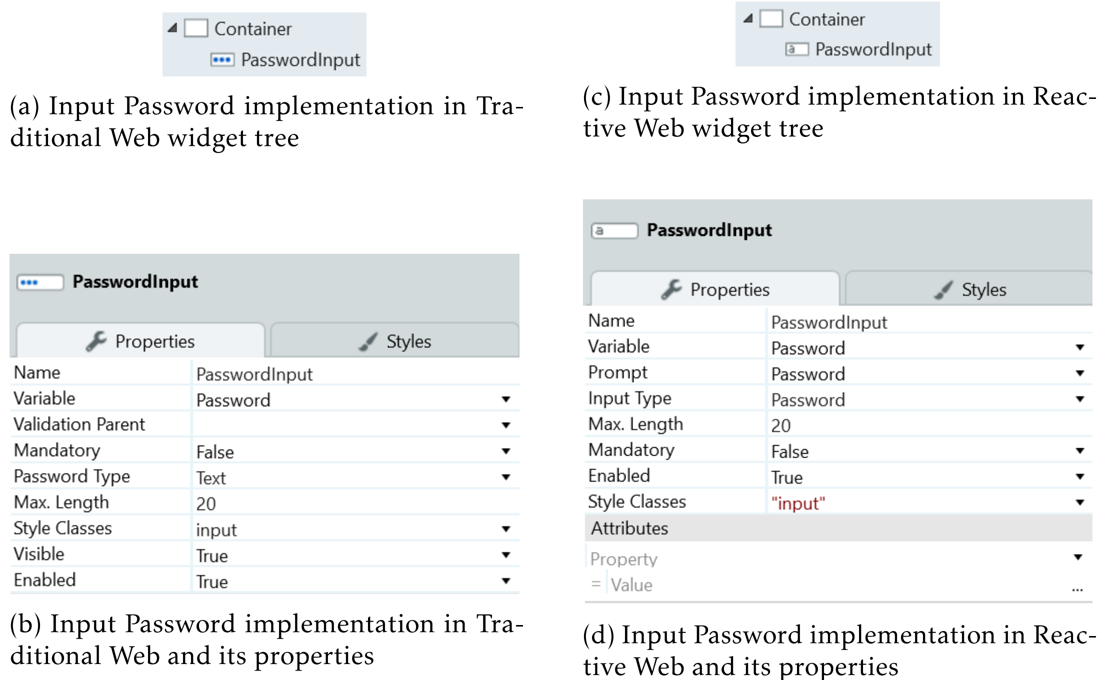


(a) Button in UI editor

(b) Button in widget tree

(c) Button widget properties

Figure 5.12: Button Widget in Reactive Web application



(a) Input Password implementation in Traditional Web widget tree

(b) Input Password implementation in Traditional Web and its properties

(c) Input Password implementation in Reactive Web widget tree

(d) Input Password implementation in Reactive Web and its properties

Figure 5.13: Differences in Input Password implementation between paradigms

Due to all of these differences (in properties and style values), as well as the widget's possible inexistence in the new paradigm, each widget has a specific algorithm where the migration to the Reactive Web paradigm is executed.

5.2.4.2 Pattern-Driven Transformations

The pattern-driven transformations are the transformations called for pattern nodes representing a pattern (a set of widgets and the relations amongst them). Unlike the direct transformations, these do not follow a set of steps, but instead, vary depending on the pattern instance to be migrated. In this type of conversion, the input is a node abstracting one or more Traditional Web widgets and the output is a set of Reactive Web widgets, with the same functionality and overall aspect, but following the paradigm's best practices.

The algorithm responsible for migrating a pattern node uses the patterns type's migration function mentioned in 5.1.3.2, and each pattern has a different migration algorithm. However, despite not existing a "migration formula" for every pattern like on the direct transformations, some common operations can be identified in all of the pattern conversion functions. These are:

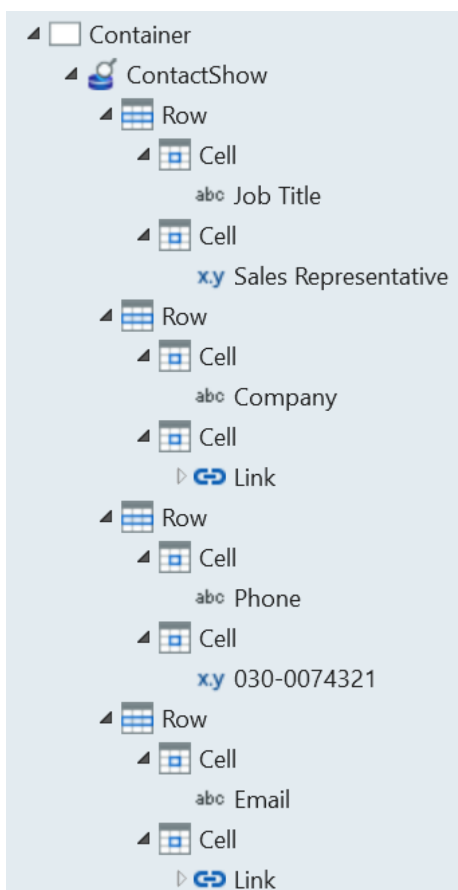
- Identifying the pattern node type
- Obtaining the stored values of the abstracted widgets
- Creating the equivalent implementation of the pattern to be migrated
- Assigning property values according to the pattern node's elements

Job Title	Sales Representative
Company	Acme, inc.
Phone	030-0074321
Email	maria.anders@acmeinc.example

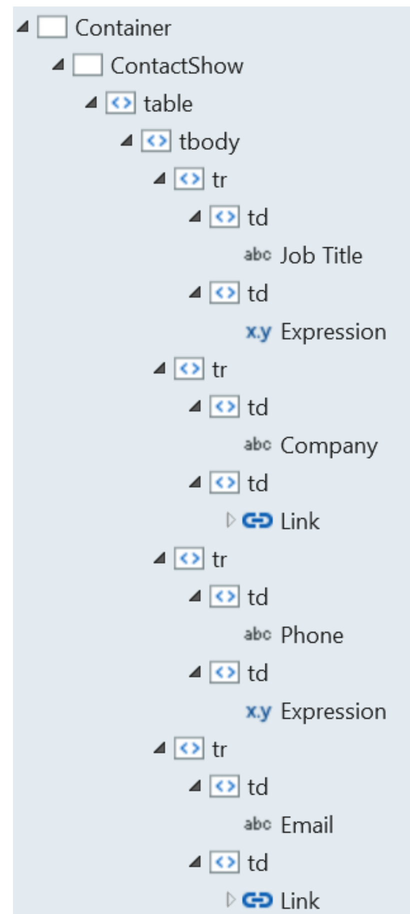
(a) Show Record implementation in Traditional Web UI editor

Job Title	Sales Representative
Company	Acme, inc.
Phone	030-0074321
Email	maria.anders@acmeinc.example

(c) Show Record implementation in Reactive Web UI editor



(b) Show Record implementation in Traditional Web widget tree



(d) Show Record implementation in Reactive Web widget tree

Figure 5.14: Differences in Show Record implementation between paradigms

A migration function for a pattern is detailed in its pattern type and receives as parameters the pattern node (representing an instance of the pattern in the widget tree) and the parent of the new Widgets. Once again, the assignment of property values uses the so-called Descriptors, making the pattern-driven transformations conform to the OutSystems meta-models. Besides, some of the direct transformation functions may be called in this process to migrate a certain widget abstracted by the pattern. After that, its children are migrated, but since a pattern transformation can create multiple widgets, the migration function must detail which of them will be the parent of the children to be migrated.

As seen in figure 5.2, a pattern encodes the relevant nodes (abstracting relevant widgets). From these nodes, its subtrees can be obtained and their descendants migrated, whether in the pattern migration function, or a direct transformation function called in the process.

Comparing to the direct transformations, creating the patterns' equivalent implementations in the Reactive Web paradigm is not as linear. This somewhat hinders the mapping of the properties and CSS values, since the properties and values of the original widgets may not exist in the target widgets (exemplified by the examples in section 5.1.3.3). So, the migration function defined in the pattern type must be as specific and precise as possible, defining the mapping of each property to its target property or attribute. In the eventuality of two very similar sets of widgets (which would be classified as the same pattern) having their properties mapped differently, two different patterns must be created and implemented accordingly, so as to maintain the strict property mapping representative of these structures. Algorithm 6 details the migration of the pattern seen in figure 5.3.

The precise transformations and mapping of the patterns migration function removes one of the obstacles faced in the direct transformations: when a Traditional Web widget did not have an equivalent implementation in the Reactive Web paradigm. That is the main reason for the UI library widgets being migrated as a pattern (as most of them do not have an equivalent implementation, so must be migrated with a clear and defined mapping between properties). In some cases, by implementing the Reactive Web equivalent, the aspect may end up different than the original widget, as seen in figure 5.3.

5.3 Progress Beyond UI

After the tool implemented allowed the migration of any given widget tree, an effort was made to identify how it could impact the migration problem. With the help of stakeholders in direct contact with OutSystems' clients interested in migrating applications from Traditional Web to Reactive Web (section 4.3), it was possible to understand the following: the majority of costumers would be interested in migrating entire screens along with the possibility of migrating a widget tree. This would result in increasing the level of

Algorithm 6: Migration function for pattern node of type Input & Button

Result: Reactive Web Widget representing the parent of the Reactive Web Widgets corresponding to the migrated pattern instance

Input : The Pattern Node *pattern* representing the Input & Button Pattern node and a Reactive Web widget *parent* representing the migration target (the parent of the migrated widgets)

```
1  Function MigrateInputButtonPattern(node, parent):
2      oldInput ← pattern.InputWidget;
3      oldButton ← pattern.ButtonWidget;
4      newContainer ← WidgetFactory.CreateWidget(parent, CONTAINER);
5      descriptor ← newContainer.Descriptor;
6      search ← WidgetFactory.CreateWidget(parent, WEBBLOCK);
7      if eSpace contains Library OutSystemsUI then
8          searchReference ← OutSystemsUI.SEARCH;
9          search.SourceWebBlock ← searchReference;
10     if oldInput ≠ null then
11         if oldInput.Width ≠ null then
12             descriptor.SetWidth(newContainer, oldInput.Width);
13         if oldInput.MarginTop ≠ null then
14             descriptor.SetMarginTop(newContainer, oldInput.MarginTop);
15         if oldInput.MarginLeft ≠ null then
16             descriptor.SetMarginLeft(newContainer, oldInput.MarginLeft);
17         newInput ←
             search.GetDescendantWithDescriptor(IInputWidgetDescriptor);
18         Call MapExtendedProperties(oldInput, newInput);
19         if oldInput.Name ≠ null then
20             newInput.Name ← oldInput.Name;
21         newInput.Descriptor.SetInputType(newInput, Mappers.GetInputValue(oldInput.Type));
22         if oldInput.Prompt ≠ null then
23             newInput.Prompt ← oldInput.Prompt.DisplayName;
24         newInput.MaxLength ← oldInput.MaxLength;
25         if oldInput.Mandatory ≠ null then
26             newInput.Mandatory ← oldInput.Mandatory.DisplayName;
27         if oldInput.Enabled ≠ null then
28             newInput.Enabled ← oldInput.Enabled.DisplayName;
29         newInput.Descriptor.SetCustomStyle(newInput, oldInput.CustomStyle;
30         if oldInput.Variable ≠ null then
31             newInput.Variable ← oldInput.Variable.DisplayName;
32     return parent.AsAbstractObject();
```

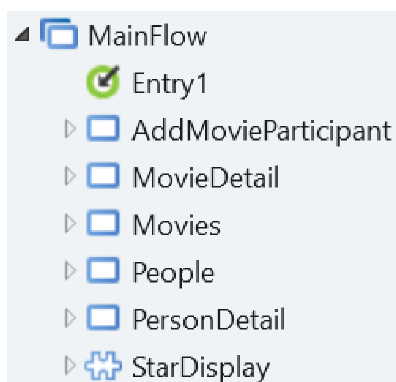
abstraction of the UI elements possible to migrate, from a widget tree (which contains multiple widgets) to a screen (which may contain multiple widget trees).

Be that as it may, an OutSystems Traditional Web screen contains other objects besides the widget trees, and those elements must be taken into account when migrating the screen. The problems encountered when approaching a full-screen migration and its respective solutions will be detailed below, culminating in the migration of more screen objects besides the UI widgets.

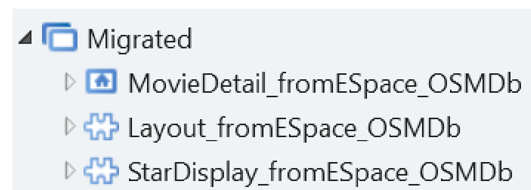
5.3.1 Full Screen Migration

The migration of a full screen consists of converting the Screen object, and for each of its widgets trees, migrate it to the new Screen in the Reactive Web paradigm. If, for example, the screen uses a Web Block (a block with UI to be reused in multiple screens), such Web Block also needs to be migrated along with the Screen.

So, as a solution, in the Reactive Web application, a new UI Flow is created and the UI elements (Screen and its Web Blocks) are migrated into that new UI Flow. This is exemplified in figure 5.15, where the *MovieDetail* Screen is migrated to the new UI Flow and so are the Web Blocks it uses.



(a) UI Screens and Web Block in Traditional Web application UI Flow



(b) Migrated Screens and Web Blocks to new UI Flow in Reactive Web application

Figure 5.15: Migrated Screen in the different applications' UI Flows

In its content, a Web Block is similar to a Screen despite being used differently: it is used to group certain widgets to be reutilized in Screens or other Web Blocks. So, a similar approach was followed, and it became possible to migrate full Web Blocks using the migration tool developed. When a new Screen is migrated and has an element which is an instance of a Web Block already migrated (individually or as part of a Screen), the tool uses the migrated Web Block, instead of migrating it again.

The full-Screen migration removes the necessity of, when a user is interested in migrating an entire screen, creating a new Screen and copying and pasting all of its UI components. The same is valid for a Web Block migration.

5.3.1.1 Problems Encountered

As formerly declared, a Screen contains more elements besides the widgets (UI elements), and when these elements are not migrated, the bindings between them and the widgets are broken, causing multiple errors in the migration result. These elements are:

- Input Parameters

  The input parameters of a screen are sometimes used in Expressions and other widgets with its values as properties. Without these parameters in the migrated screen, some UI elements have their bindings broken, and its values cannot be obtained, leading to errors in those widgets.

- Local Variables

  The local variables of the screen, like the input parameters, are an intrinsic part of the screen. As such, many UI widgets are bound to those variables and their inexistence in the migrated screen can lead to some errors due to broken bindings.

- Aggregates in Preparation

  The Preparation is a Data Action that takes place before the rendering of the associated Traditional Web screen. This Data Action is used to fetch the data to be presented on the screen, as well as to do some necessary preprocessing and variables assignment. In the Preparation, there are some nodes called Aggregates, responsible for fetching the data from the database by abstracting SQL queries. Since the Aggregates are not migrated, the database cannot be queried and multiple widgets cannot obtain the necessary data, creating errors on multiple properties of UI widgets. Figure 5.16 shows an error in an Expression widget due to the Aggregate *GetProductionPeople* not being migrated with the Screen (the value represents an Aggregate that does not exist).

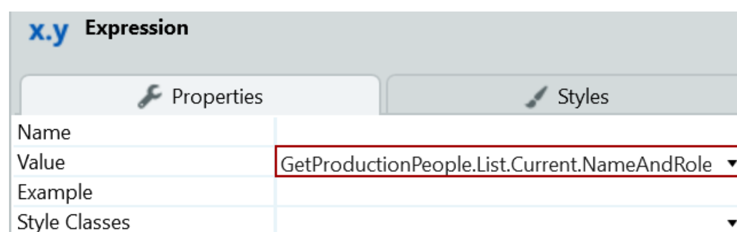


Figure 5.16: Error in Expression widget due to Aggregate not existing.

So, a screen and its widgets are dependent on the abovementioned elements, causing a poor migration experience when those are not migrated. Figure 5.17 shows the errors caused by the broken bindings between the UI components (such as widgets and their properties) and the screen elements not migrated.

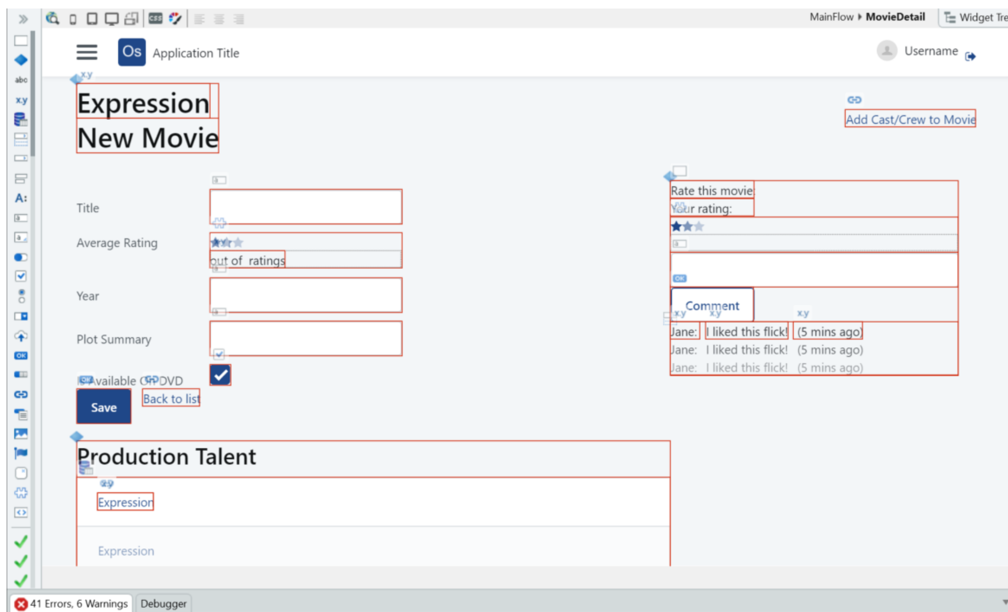


Figure 5.17: (41) Errors in a Screen where the Input Parameters, Local Variables, and Aggregates from the Preparation were not migrated.

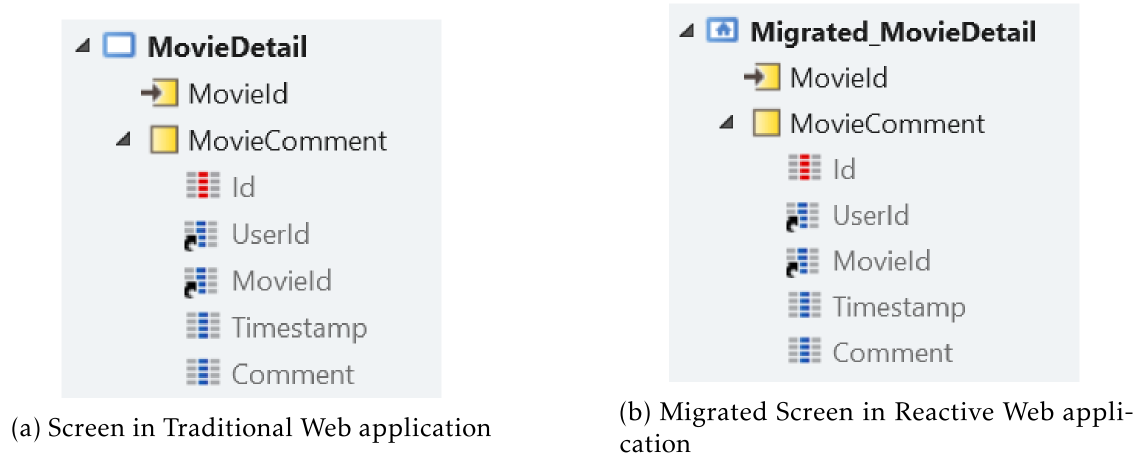
Hence, despite not being UI elements (the focus of this dissertation), we concluded that the Screen UI migration would highly benefit from having the Input Parameters, Local Variables, and Aggregates from the Preparation migrated.

Also, since the developed tool undertakes the automatic migration of a screen at a time, a limitation in the screen bindings migration arose. This is because some properties of widgets contain pointers to other screens, which, before such screens are migrated, cannot be assigned (the properties must be assigned a reference to such screens, which have not been migrated when the first screen is converted to the new paradigm). This can be seen in appendix A. Although this problem could not be tackled (unlike the migration of the abovementioned elements), this was already a step in manual migration. So, the automatic migration did not complicate such a step, but was simply not able to automate it.

5.3.2 Inputs and Variables Migration

To solve some of the previous errors, the automation of the migration of Input Parameters and Local Variables was undertaken. As a result, when a Screen is migrated, so are its Input Parameters and Local Variables, maintaining its designation and data types. Figure 5.18 shows the original and the migrated screen with its Input Parameter and Local Variable (with associated data).

A Local Variable can have as data type a standard value such as Text, Integer, and Boolean. However, it can also have as data type an Entity and, in that case, the reference for said Entity is imported to the Reactive Web application. This results in a database migration since the used entities and its values can then be used in the Reactive Web



(a) Screen in Traditional Web application

(b) Migrated Screen in Reactive Web application

Figure 5.18: Migrated Screen result

application as exemplified in figure 5.19.

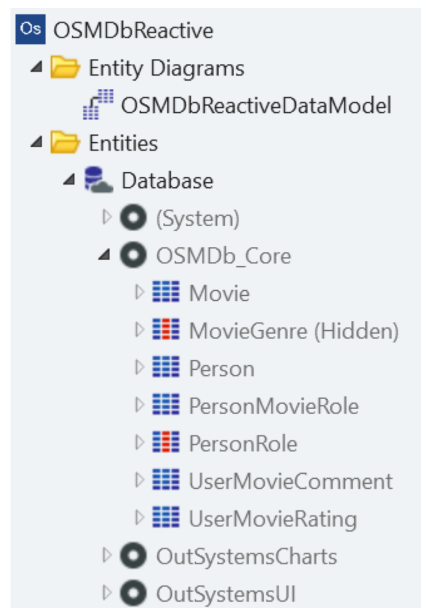


Figure 5.19: Entities migrated to the new paradigm as references.

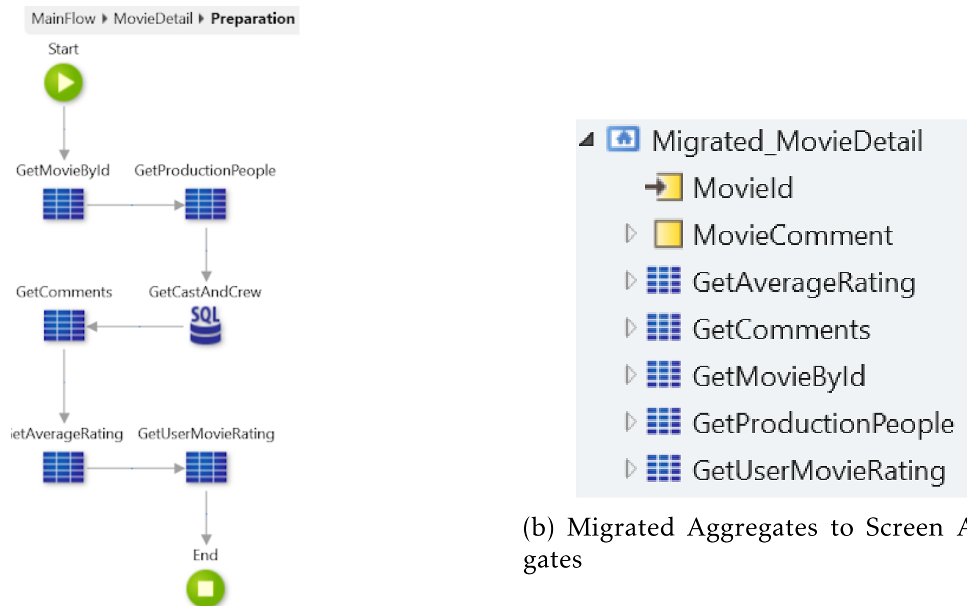
5.3.3 Aggregates Migration

As already explained, Aggregates abstract SQL queries and represent database requests to obtain data. However, unlike Input Parameters and Local Variables, these cannot simply be transformed into Reactive Web Preparation Aggregates due to a significant difference between paradigms: The Reactive Web Screens and Web Blocks do not have a Preparation data action.

In the Reactive Web paradigm, the Screens (and Web Blocks) have what are called the Screen (and Web Block) Aggregates, instead of having the Aggregates in the Preparation

(which does not exist in this paradigm). A Screen Aggregate fetches the data before the Screen is rendered and any preprocessing is done using data actions.

Thus, to correctly migrate the Aggregates from one paradigm to the other, the migration tool must convert the Preparation Aggregates from the Traditional Web paradigm in Screen Aggregates from the Reactive Web paradigm. That is accomplished by manipulating the model objects (the Preparation Aggregates and Screen Aggregates conform to the same meta-model objects), and an example of the result can be seen in figure 5.20.



(a) Aggregates in Preparation of a Traditional Web Screen

(b) Migrated Aggregates to Screen Aggregates

Figure 5.20: Preparation Aggregates before and after being migrated (in the different Web paradigms)

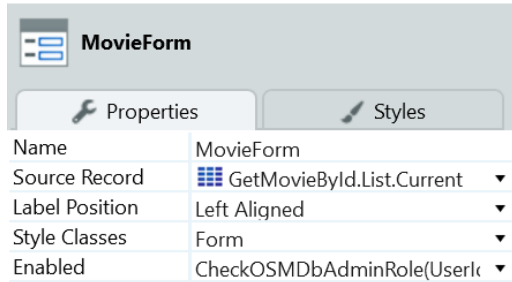
Like with the Input Parameters and Local Variables, the entities fetched by aggregates are imported to the Reactive Web application as references.

5.3.4 References Repairing

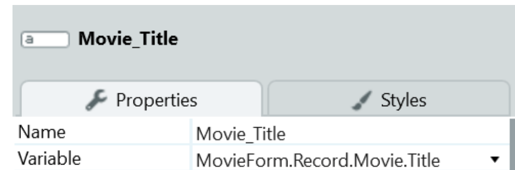
However, migrating the Preparation Aggregates by itself does not solve a considerable amount of errors caused by the broken bindings between widgets and Aggregates. As a result of the implementation of the paradigms, the Aggregates values are referenced differently in the Traditional and Reactive Web applications. That leads to some of the widgets not being able to obtain and use the migrated Aggregates, considering the properties' values as they were in the previous paradigm.

In other words, the Aggregates' usage in properties of Traditional Web widgets is migrated to the properties of the Reactive Web widgets, where they should be used in a distinct manner. This is a case of widgets like Forms, Show Records, List Records, and others, which have a property named *Source Record* in Traditional Web, assigned to an

Aggregate value. So, when other widgets want to reference the Form's Aggregate (for example), they can reference the Form's Record instead of directly using the Aggregate value as seen in figure 5.21. In Reactive Web, no widget has a *Source Record* property, thus, if a widget wants to use a certain Aggregate, it must reference it directly as seen in figure 5.22.

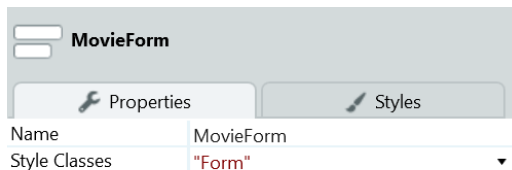


(a) Form widget in Traditional Web application

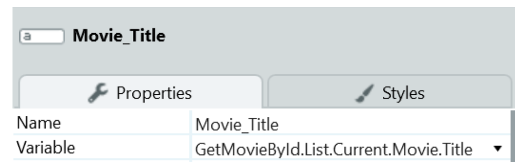


(b) Aggregate reference in Input inside Traditional Web Form

Figure 5.21: Form and Input widget with Form Agreggate reference in Traditional Web application



(a) Form widget in Reactive Web application



(b) Aggregate reference in Input inside Reactive Web Form

Figure 5.22: Migrated Form and Input in Reactive Web application with correct reference

Considering this, the migrated references according to the old paradigm must be changed to match the Reactive Web paradigm's correct notation. To fix the reference values, a mapping must be executed during the migration, where widgets which have a source record, have its value and the source record value added to a collection. After the migration, the screen's widget trees are covered and for every widget, if an Aggregate reference exists for another widget's source record in the collection, the source record value is obtained from that collection and referenced directly. In other words, for every widget referencing another widgets' source record, the referenced is fixed to use the precise value of the Aggregate.

So, as an example, an Expression widget with value *MovieForm.Record.Movie.IsAvailable OnDVD* becomes *GetMovieById.List.Current.Movie.IsAvailableOnDVD*.

5.4 Migration Information

Due to discrepancies between the Web paradigms and their respective models, some parts of the migration may not be possible or may require manual verification. These

transformations may be of elements, relations, or sets of elements and relations, as is the case of the pattern-driven transformations. Not only that, but the migration is a complex project, sometimes involving abstractions and operations across multiple Screens and Web Blocks. Hence, the users would benefit from having information about the process and transformations' details documented and presented to them. This information is presented via logs and metrics, and includes details not only about what was not possible to transform, but also migration context, choices, and performance data.

However, this migration tool allows a developer to migrate different sets of elements that can range from a widget tree with few elements in a copy-paste interaction (or even one single widget), to an entire screen (see sections 5.3.1 and 5.6). So, depending on the different use cases, there may not be a need to communicate the migration information to the user, since the storage and presentation of such information can be costly (as explained in section 2.6.2). As an example, in the scenario where the user copies a container with an expression widget from a Traditional Web application and pastes it in a Reactive Web application, a migration log and metrics would be adding unnecessary complexity and cost to a simple operation. However, when a user chooses to migrate an entire screen, the operations transform multiple elements and involve different processes and migration choices, creating a scenario where the migration information is essential to the user.

So, taking into account the different use cases, it was possible to conclude that when the user copies some widgets, a migration log and metrics are not necessary due to the goal of such operation (quickly reutilize certain Traditional Web elements when implementing a Reactive Web Screen). On the other hand, when a user migrates an entire screen, the objective is to migrate that screen and all of its elements to a different paradigm, making a migration log necessary to better understand the processes involved. To sum up, information about the migration process will be presented whenever a user decides to migrate an entire screen (or web block).

5.4.1 Migration Logs

One way of communicating the migration information is through execution logs, strings of text with useful data recorded during system execution (see section 2.6.2). As previously stated, the decision was to present the migration information, such as the logs, when a screen or web block is migrated. Via a textual representation, the following information contextualizing and detailing the processes is provided:

1. Screen and eSpace identification

   The original Screen and its eSpace are detailed, as well as the target eSpace (where the screen will be migrated to)

2. Local Variables, Input Parameters, and Aggregates migrated

   The migrated screen elements of these types are listed, and, in case any of its properties suffer any change, that is detailed. As an example, we have the *Max Records*

property in aggregates, which is not mandatory in Traditional Web aggregates but is in Reactive Web aggregates. So, when an aggregate that does not have a *Max Records* value assigned is migrated, the default value (advised by OutSystems) is assigned by the migration tool, and that information is communicated to the user. Exemplified in figure 5.23.

3. Screen Content identification

   The migration information communicates the widget tree to be migrated (with every widget having an assigned ID). This is done by printing the widget tree and the total number of widgets.

4. Patterns found in widget tree

   The instances of patterns types found in the widget tree are listed and the total number of patterns is indicated. Exemplified in figure 5.24.

5. Migration Logs per se

   The migration logs specify the migration of each tree node (whether a pattern or a node representing a widget). In the specification of the migration operations of an element, the conversion of its properties, CSS values, and attributes are detailed and a success, failure, or warning symbol is presented. Whenever necessary, further detail is provided (e.g. recommendations on how to proceed). Exemplified in figure 5.25.

6. References resolved

   The references repairing detailed in the previous section is also presented to the user for them to know what was changed in the migration. For each referenced changed, the widget type is specified, as well as the old and new values.

```
Migrating Aggregates, Input Parameters and Local Variables:

    Aggregate GetComments migrated to the new screen.
    Aggregate GetAverageRating migrated to the new screen.
        ⚠ The aggregate did not have a Max Records value, set to the default value 50.
    Aggregate GetUserMovieRating migrated to the new screen.

    Input Parameter MovieId migrated to the new screen.
    Local Variable MovieComment migrated to the new screen.
```

Figure 5.23: Aggregates, Input Parameters and Local Variables migration logs example.

The information mentioned is stored during the migration process (and its multiple phases) and presented after it is completed. In appendix A, a screen migration result is detailed and its metrics can be consulted.

```
Pattern Search:

    LayoutMenuTop from OSUI pattern found.
    Label & Input pattern found.
    Icon pattern found.
    Label & Input pattern found.

    4 patterns found.
```

Figure 5.24: Patterns found migration logs example.

```
Expression widget conversion (widget30)
    ✔Name property migrated.
    ✔Value property migrated.
    ✔Example property migrated.
    🔍Style property migrated.
      Manual verification advised.
    ⚠Visible property not migrated.
      Property does no exist in Reactive Web paradigm.
    ⚠Enabled property not migrated.
      Property does no exist in Reactive Web paradigm.
```

```
Link widget conversion (widget59)
    ✔Name property migrated.
    🔍Style property migrated.
      Manual verification advised.
    🔍Title property migrated as attribute.
      Manual verification advised.
    ✔Enabled property migrated.
    ✔Visible property migrated.
    ✔On Click Confirmation Message migrated.
    ⚠On Click Destination not migrated.
      Manual setting required.
```

Figure 5.25: Widget migration logs examples.

5.4.2 Migration Metrics

The migration is a complex set of processes and the log information can be quite lengthy. To give the user a sum up of the migrated elements and other information (such as the elapsed time), performance metrics (section 2.6.1) were used as an information summary.

These metrics present the following information about the migration of a screen (or web block): the elapsed time (duration); the number of Aggregates migrated (from Preparation Aggregates to screen Aggregates); the number of Input Parameters migrated; the number of Local Variables migrated; the total number of widgets in the screen; the number of widgets migrated; the total number of pattern instances identified; the number of pattern instances migrated; the number of web blocks migrated to the Reactive Web eSpace (used in the screen and not present in the new eSpace). With this data, the user has an overview of what was migrated and how long it took. In case they need further information, the migration logs detail the process in a more extensive approach.

Figure 5.26 exemplifies the performance metrics shown for a Screen migration.

```
Duration: 1.46 seconds

Number of Aggregates migrated: 5
Number of Input Parameters migrated: 1
Number of Local Variables migrated: 1

Total number of Widgets: 90
Number of Widgtes migrated: 90

Total number of Patterns identified: 4
Number of Patterns migrated: 4

Number of Web Blocks migrated to Reactive ESpace: 2
```

Figure 5.26: Screen migration's performance metrics example.

5.5 Overview

So, the developed approach comprises the migration of a Traditional Web Screen, involving the migration of UI elements such as widgets, as well as other screen elements. To sum up, using the developed tool (comprising the preprocessing, processes, and algorithms) the following manual migration steps are now automated:

- The creation of a Screen in the Reactive Web application
- The migration of the Screen UI content (widgets)
- The migration of the Web Blocks present in the Screen
- The migration of Input Parameters and Local Variables
- The migration of the Preparation Aggregates to Screen Aggregates
- The new Screen content is covered to solve the broken references to the Aggregates

A diagram depicting the UI migration of a Traditional Web Screen and its multiple processes may be found in figure 5.27, in which every process was detailed in the previous sections. One example of a migration execution and result can be found in appendix A.

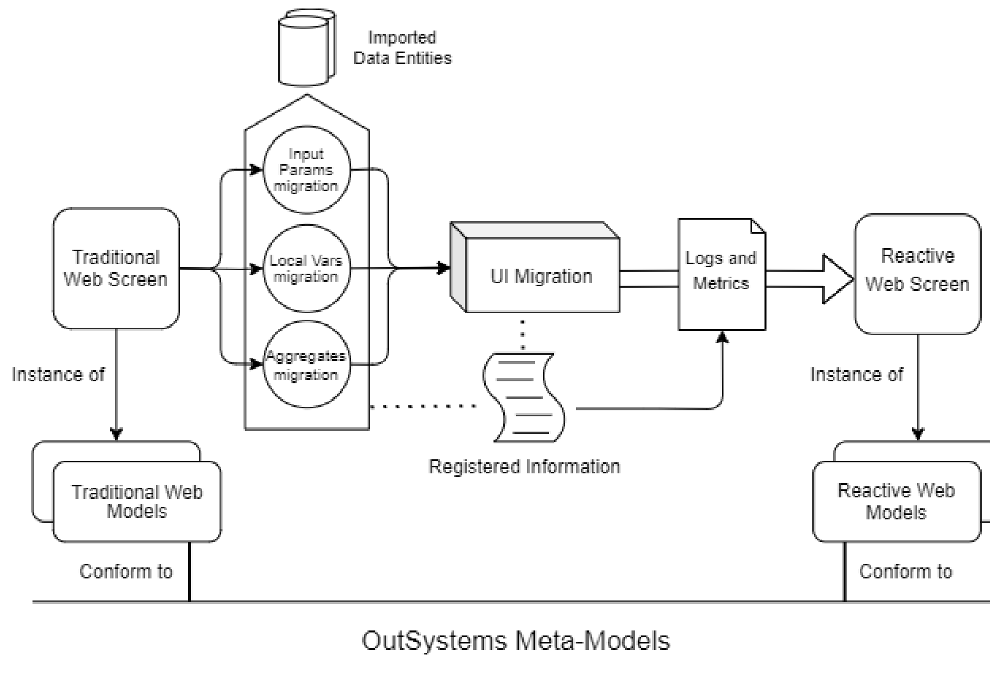


Figure 5.27: Screen migration processes and operations.

5.6 Integration in the OutSystems Platform

After the migration tool was developed, another goal of the dissertation was to integrate it into the OutSystems platform to allow users to easily migrate their applications. In the

previous sections, the migration process was extensively detailed, however, for something to be migrated, the user must specify two things: what will be migrated, and where it will be migrated to.

To specify these two inputs for the migration, the automatic migration purpose was revisited, namely, the fact that its goal is to serve as an accelerator and provide some automation to the manual migration (section 3.1.2). Taking into account that the manual migration already has some accelerators, we noticed that those were mostly implemented with a copy-paste approach (e.g. copying data actions from a Traditional Web application and pasting them in a Reactive Web application). At first glance, a copy-paste interaction seems to fit the migration tool: the copy allows the user to choose what to migrate, and the paste lets them choose where to migrate.

That being said, to analyze what should be the approach when integrating this tool in the platform, its use cases must also be considered, as different use cases may require different integrations. The migration process, as it is, can support two purposes:

1. Migrate a set of widgets (defined by a selected widget tree)

2. Migrate an entire screen (with its variables, aggregates, and UI elements)

For the first use case (1.), the copy/paste interaction suits its demands, since it is a scenario where the user has two screens (one implemented in each paradigm) and wants to reutilize the content of the Traditional Web screen in the Reactive Web screen. This gives the user a quick and effective way of choosing the elements of the screen they want to migrate, and specify the precise target where they intend to migrate to.

On the other hand (use case 2.), when migrating a full screen, it must be given as input and will be migrated to a UI flow in the Reactive Web Application. In this case, the copy-paste interaction also allows the user to choose which screen or set of screens to migrate (by selecting and copying them), as well as choosing the application where it will be migrated to. To clearly identify which screens were migrated, they are inserted in a specific UI flow in the target application.

So, a copy-paste interaction not only follows the interaction principles of the already implemented accelerators on the manual migration, but also fits both scenarios of the automatic migration. To sum up, it allows the user to choose what to migrate with a simple to use interaction. So, when a set of widgets is copied from a Traditional Web widget tree/screen and pasted in a Reactive Web widget tree/screen, the elements are migrated and the conversion result is pasted (the original elements could not be pasted as they do not conform to the target models). This can be seen in figure 5.28.

The same happens for a screen or set of screens: when copied from a Traditional Web application and pasted in a Reactive Web application, the screen is migrated and the conversion result is pasted (depicted in figure 5.29).

For this to work, the Service Studio (OutSystems' platform) builtin behaviors had to be modified. Now, when a user copies a Traditional Web widget or screen and pastes

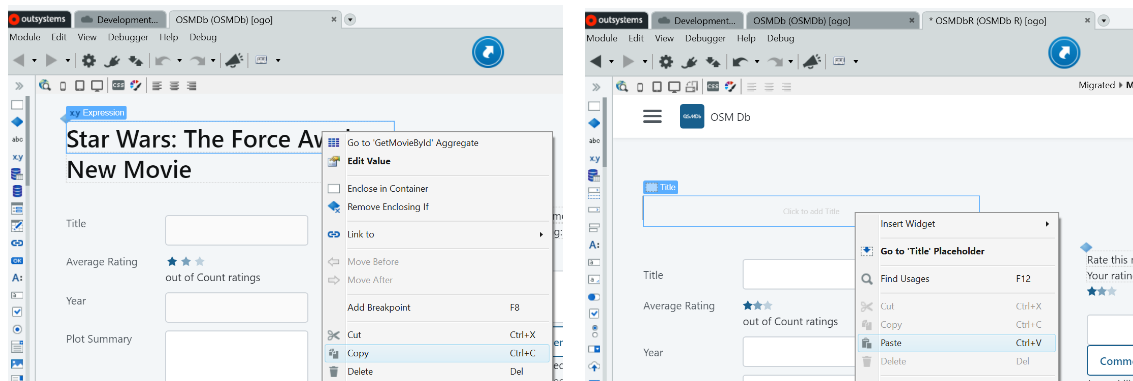


Figure 5.28: Copy of widget from Traditional Web application and paste in Reactive Web application.

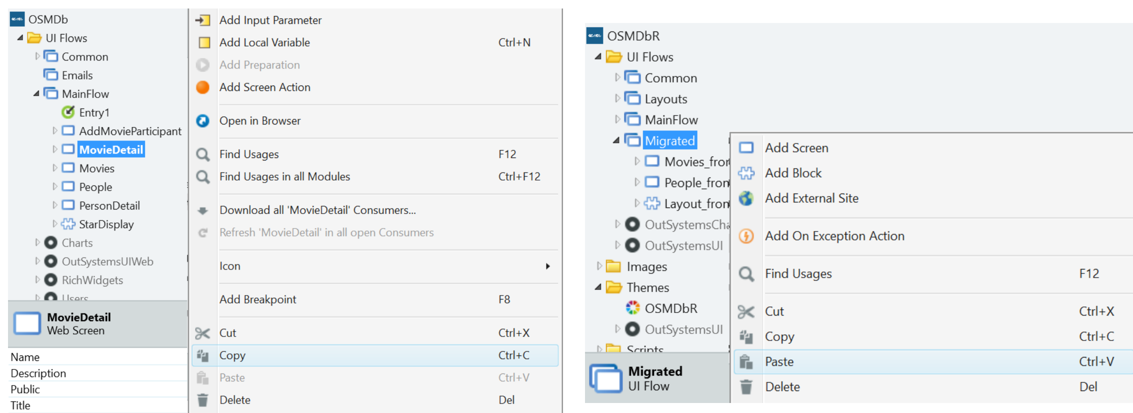


Figure 5.29: Copy of screen from Traditional Web application and paste in Reactive Web application's UI Flow.

it in a Reactive Web application, the platform uses the migration tool to perform the transformations and paste the migration result. Thus, it becomes possible to copy those elements from the old paradigm and paste them in the new paradigm, something that was unattainable before due to the differences in the paradigms' models.

Chapter 6

Evaluation

Once the tool was developed, it was evaluated and tested with two analyses and two types of tests, consisting of a total of four validations.

To understand its impact, the following analyses were performed: (1) A coverage analysis, where the percentage of elements possible to migrate was estimated and separated into different categories; (2) Queries run in the OutSystems clients' accounts, to better quantify how many systems, applications, and elements can benefit from the implemented automation.

After the analysis, the tool was tested in two different manners: (1) An average performance comparison between the manual and the automatic UI migration; (2) Usability tests carried out by users and evaluated using the System Usability Scale method.

6.1 Coverage Analysis

To better understand (and quantify) how much the tool helps a user in migrating an application, its coverage had to be evaluated. So, data concerning the total number of UI elements' types (widgets' types) in Traditional Web was gathered, as well as the number of elements that can now be migrated automatically. Three sets of widgets were evaluated:

- Traditional Web Widgets - The widgets of the Traditional Web paradigm
- OutSystems UI Widgets - Widgets of a library implemented by OutSystems, which contains more complex widgets to complement the Traditional Web paradigm and provide additional functionalities
- Rich Widgets - Widgets of another library implemented by OutSystems, which are commonly used in Traditional Web applications

Figure 6.1 details the number of widgets possible to migrate per category. As depicted in the figure, the standard widgets are all possible to migrate automatically, as well as the majority of the OutSystems UI widgets. However, some widgets from both libraries are not possible to migrate with the tool as it is.

A widgets' migration can be more complex than other widgets', for example, the widgets belonging to libraries may not have an equivalent implementation in the new paradigm. Due to the implementation consisting of a proof of concept and not the final version of the tool, some widgets are not automatically migrated at the moment, and thus were left for future work.

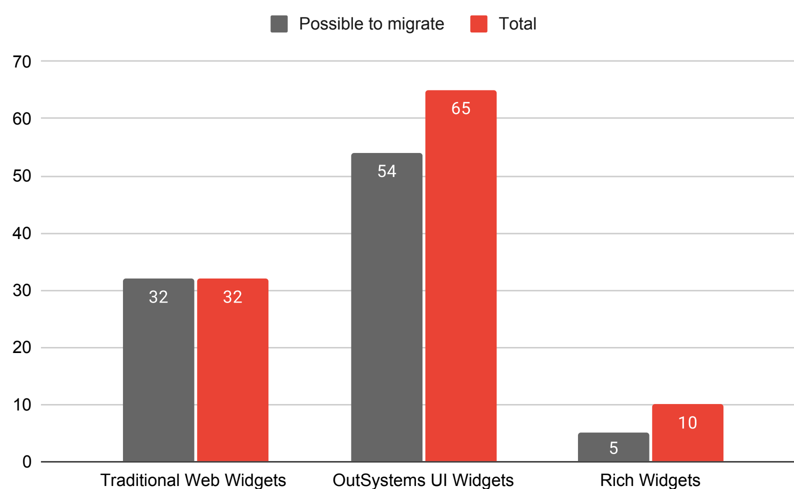


Figure 6.1: Total number of widgets and number of widgets possible to migrate by category.

So, as coverage analysis, we can conclude that all of the standard widgets can be migrated automatically. Regarding external libraries, those created and supported by OutSystems have the majority of its elements migrated. This makes the tool very useful and its impact significant, as proven in the succeeding sections.

6.2 Queries in OutSystems Accounts

Before knowing how the tool impacts the migration of the Traditional Web UI, it is important to quantify the need for this tool, as well as the impact it will have in production. Hence, several metrics were gathered to better understand the differences in paradigms' usage (to know how many Traditional Web applications will be benefit from the tool), as well as the average complexity of such applications. These metrics were measured using queries ran in OutSystems clients' accounts with their consent.

The total number of OutSystems' eSpaces is **132 117**, of which **83 224** are Traditional Web ESpaces (63%) and **9 638** are Reactive Web ESpaces (7%). So, the Traditional Web ESpaces considerably outnumber the Reactive ESpaces, but it would be beneficial for both

the users and OutSystems that the opposite happened. However, the manual migration is a cumbersome process, thus discouraging the migration of said applications and keeping the number of legacy systems undesirably high.

To see how many of these ESpaces could be impacted by the automatic migration, the number of ESpaces with UI components was evaluated: **57 105** Traditional Web ESpaces have UI components, as well as **8 073** Reactive Web ESpaces. Since the tool automates the UI migration (which is the greatest bottleneck in a migration, as explained in section 4.1), there are **57 105** applications which can greatly benefit from the provided automation.

However, the number of applications with UI by itself is not a clear indicator of how many elements can be migrated automatically with the developed accelerator. In more depth analysis, it was possible to obtain the following numbers:

- There are **528 875** Web Blocks in Traditional Web applications
- There are **489 600** Screens in Traditional Web applications
- There are **41 039 469** Traditional Web Widgets across all of the applications
- There are **1 195 711** Input Parameters across all of the Traditional Web Screens and Web Blocks
- There are **1 173 538** Local Variables across all of the Traditional Web Screens and Web Blocks
- There are **559 063** Preparation Aggregates across all of the Traditional Web Screens and Web Blocks

All of the abovementioned elements can now be automatically migrated. If we look at the Traditional Web widgets, some of them are fairly complex and can take a couple of minutes to migrate manually (with the properties mapping and style setting), while automatically, the migration is instantaneous. When analyzing the difference in a widget's migration, it is only a few minutes, but accumulated over forty-one million widgets, the difference is tremendous. To illustrate such a difference, if a widget takes (on average) 3 minutes to manually migrate (including its properties and style), the developed tool may save an accumulated amount of time equal to **123 118 407** minutes, or, in other words, **234.24** years. Considerable amounts of time saved can also be obtained for the Input Parameters, Local Variables, and Preparation Aggregates.

Screens and Web Blocks have various elements such as Widgets, Input Parameters, Local Variables, and Preparation Aggregates. So, the difference will be significant even when migrating a single Screen or Web Block, and even more when migrating 489 600 and 528 875 of them, respectively. A performance comparison will be undertaken in the next section, but before, other metrics were compiled to fundament such comparison. The data regarding the average number of elements per screen and web block (which are now possible to migrate) is detailed in table 6.1

Traditional Web	**Widgets**	**Input Parameters**	**Variables**	**Preparation Aggregates**
Screen	83.82	2.44	2.4	1.14
Web Block	77.6	2.26	2.22	1.06

Table 6.1: Average number of elements per Traditional Web Screen and Web Block.

An also important data to understand the effort of migrating a Traditional Web application is the average number of Screens and Web Blocks in it. This can be deducted with the data already presented: on average, a Traditional Web application contains **8.57** Screens and **9.26** Web Blocks.

6.3 Performance Comparison

To test the automatic Screen and UI migration and its performance, we must take into account the previous (and current) migration methodology: the manual migration (section 3.1.2). So, the performance of both approaches was tested to understand the impact this dissertation can have on the real-life complexity of migrating applications.

The initial comparison plan contemplated having users with at least one year of experience with the OutSystems' platform migrating the same screen manually and using the tool. Via this experiment, it would be possible to compare both performances, as well as the usability of the automatic migration tool (developed in this dissertation's context).

However, when the initial users began migrating the first screen, it became clear that it would not be possible to perform the planned validation: After one hour, all three of the users had migrated around 10% (between 9% and 11%) of the original screen. The screen in question had 91 widgets, 3 Aggregates in the Preparation, 1 Input Parameter, 0 Local Variables, and 3 instances of Web Blocks that had to be migrated (one of them containing 178 widgets). The same screen (as well as all of its elements and Web Blocks used) was automatically migrated by the developed tool in 5.1 seconds.

So, a quick evaluation allowed us to understand that a manual migration of that particular screen would take approximately 10 hours or more. Since such screen was not highly complex (the number of widgets was around the average per screen according to section 6.2), and the number of hours needed to finish the migration was incompatible with the volunteers' availability (also due to restrictions caused by COVID-19), a different approach to the performance comparison was undertaken.

As an alternative, the following approach was considered: evaluating the effort required to manually migrate a screen or web block with the Advanced Development Team and comparing its performance to the automatic UI migration. The Advanced Development Team belongs to the Customer Office Department and is responsible not only for migrating clients' applications but also for instructing clients on how to manually migrate according to OutSystems' best practices. Hence, the team in question has extensive experience in manual migrations, and those past migrations will be the data used to

Duration	Widgets	Aggregates	Input Params	Local Variables	Web Blocks
4.79 s	160	4	1	0	1
0.51 s	67	2	0	1	0
0.59 s	74	2	3	0	0
5.1 s	91	3	1	0	3
0.4 s	62	1	0	0	0
0.69 s	64	2	1	4	0
1.27 s	90	2	1	2	1
1.37 s	90	5	1	1	1
0.28 s	52	1	0	0	0
0.39 s	51	1	1	0	0

Table 6.2: Migration elapsed time (in seconds) and number of elements migrated for different Screens

Duration	Widgets	Aggregates	Input Params	Local Variables	Web Blocks
3.43 s	178	4	3	2	0
5.74 s	142	2	5	2	2
1.5 s	249	2	2	0	0
5.63 s	162	2	2	0	0
2.98 s	165	2	5	2	0

Table 6.3: Migration elapsed time (in seconds) and number of elements migrated for different Web Blocks

evaluate the automatic migration performance.

According to the team and based on past manual migrations, the processes automated by the tool in a screen migration (sections 5.5) would take from one to one and a half working days if migrated manually. In other words, for an average screen, its manual migration would take between 8 and 12 hours. As depicted in table 6.1, we can see the average number of elements per Screen and Web Block, so it is possible to understand the size of the average screen, for which we now have a manual migration time estimate.

Table 6.2 shows the execution times for the automatic migration of different screens, as well as the number of Aggregates, Input Parameters, Local Variables, and Web Block instances that needed to be migrated. Table 6.3 shows the same information for the automatic migration of different Web Blocks.

As we can observe in tables 6.2 and 6.3, the automatic migration time of both a Screen or Web Block takes merely seconds. Based on the experiments made using the automatic migration tool (some depicted in the abovementioned tables), we can affirm that the migration time is mostly influenced by the necessity of migrating web blocks to the new paradigm (due to being used in the migrated screen/web block).

However, despite the small differences in elapsed time observed in different screens and web blocks, the automatic migration process is considerably faster than the manual migration. So, an average screen that would take between 8 and 12 hours to migrate, now takes less than 10 seconds (the largest elapsed time recorded for a screen of average

complexity, according to the data retrieved in section 6.2, was around 9.38 seconds). Thus, while a screen is manually migrated, so can be between 2 880 and 4 320 screens if automatically migrated using the developed tool (considering a migration time of 10 seconds).

Even if the automatic migration result produces an incomplete screen or web block, the missing operations would also have to be undertaken in the manual migration. In other words, even when the migration cannot be completed, it is significantly accelerated (with some of the screen's elements being migrated almost instantly). Thus, it is possible to confirm that the developed tool produces a very efficient automation of the UI migration, or, in the worst-case scenario, a valuable accelerator to the manual migration.

6.4 Usability Experiment

Usability is a measure of how the user interacts with a product and is assessed by taking into account user performance, satisfaction, and acceptability [5].

To measure the developed system's usability, an experiment was conducted with 21 users, of which 66.66% were male and 33.33% were female, between the ages of 20 and 53. These participants, their identity, and responses are to remain anonymous under the protection of a verbal agreement made prior to the usability test.

All of the subjects had an engineering educational background and varying expertise concerning the OutSystems' platform. They can be separated into 3 groups: 7 users had no experience with the OutSystems platform, 6 users had between one month and one year of experience with the platform, and 8 users had more than one year of experience with the platform. The differences in experience allowed the usability test to evaluate the manual and automatic migration usage and results according to different perspectives. The procedure took between 45 and 60 minutes, and consisted of the following steps:

1. The user performed a manual migration of a segment of a screen, where some peculiarities and key scenarios of the manual migration could be experienced.
2. The user communicated his feedback and carried out the System Usability Scale (SUS) test for the manual UI migration.
3. The user did an automatic migration of the same screen (in its entirety), using the developed tool.
4. The user inspected the migration logs produced.
5. The user communicated his feedback and carried out the System Usability Scale (SUS) test for the automatic UI migration.

The goal was for the users to experiment using both UI migration processes (manual and automatic), and for the feedback to reflect not only the individual usabilities but also how they compare to one another.

Since the manual migration was executed before the automatic migration, a threat to the validity of this experiment arises. When performing the manual migration, the user may gain some context which later facilitates the automatic migration. Yet, the automatic migration does not need the manual operations used in the manual migration, making the learned operations irrelevant to the experiment, and only the minimal amount of context gained regarding both paradigms can cause a small vulnerability in the SUS result.

Another threat to the experiment's internal validity is the fact that subjects did not migrate entire screens (due to their availability and COVID-19 restrictions), making their experience influenced by the part of the screen migrated. To compensate for this circumstance, different users migrated separate parts of distinct screens, to obtain a sample representative of various types of screens.

6.4.1 SUS

The System Usability Scale (SUS) is a "simple, ten-item scale giving a global view of subjective assessments of usability"[11]. For each question, the user indicates its agreement or disagreement level on a five-point scale (1 representing "strongly disagree" and 5 representing "strongly agree"). Figure 6.2 presents an interpretation for the SUS test score obtained in [4].

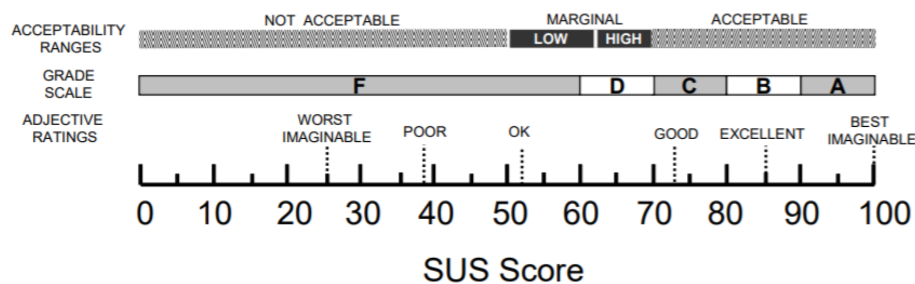


Figure 6.2: Adjective ratings, acceptability scores, and school grading scales, in relation to the average SUS score [4].

During the usability experiment, the participant answered the System Usability Scale (SUS) test on two occasions to evaluate both the manual and the automatic migration usability. Table 6.4 shows the mean answer per question for each migration approach. With the scores obtained, the descriptive statistics for the different SUS tests were calculated and can be seen in tables 6.5 and 6.6, as well as the boxplot in figure 6.3.

Through an analysis of the scores obtained for both approaches using in the table 6.2, we can confidently claim that the obtained mean value of 94.29 presents an extremely positive result and an improvement regarding the manual migration (which obtained a mean value of 22.86). It is possible to observe, looking at the metrics in tables 6.5 and 6.6, that the manual migration score varies considerably according to the user, while the automatic migration does not show a large difference between the tests' results. This is also seen in the boxplot in figure 6.3.

Question	Manual Mig. Mean Score	Automatic Mig. Mean Score
1. I think that I would like to use this system frequently.	1.33	4.9
2. I found the system unnecessarily complex.	4.57	1.1
3. I thought the system was easy to use.	2.29	4.86
4. I think that I would need the support of a technical person to be able to use this system.	4.05	1.43
5. I found the various functions in this system were well integrated.	2	4.86
6. I thought there was too much inconsistency in this system.	3.24	1.29
7. I would imagine that most people would learn to use this system very quickly.	1.71	4.76
8. I found the system very cumbersome to use.	4.76	1.05
9. I felt very confident using the system.	2.52	4.62
10. I needed to learn a lot of things before I could get going with this system.	4.1	1.43

Table 6.4: Mean SUS score for each UI migration approach per question.

N	Mean	Std. Dev.	Skew.	Kurt.
21	22.86	8.49	0.039	-1.274

Table 6.5: SUS descriptive statistics for the manual migration

N	Mean	Std. Dev.	Skew.	Kurt.
21	94.29	3.46	-0.168	-0.649

Table 6.6: SUS descriptive statistics for the developed tool

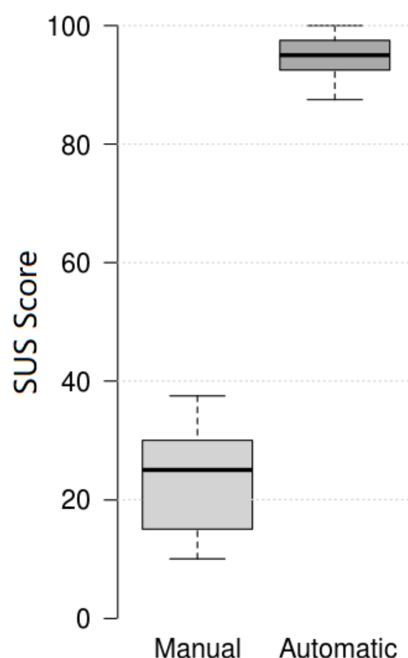


Figure 6.3: Boxplot for the SUS Score distribution.

6.4.2 Results and Analysis

For the manual migration, the following feedback was gathered during the validation experiment:

- The instinct of many users was to copy and paste elements, which is not possible in the manual migration. Hence, they were forced to manually replicate each widget.
- When the mapping of a certain widget was not as linear, the inexperienced participants had trouble migrating and understanding where to migrate each property value. Only the users experienced in both paradigms performed what could be classified as a smooth migration of these widgets.
- It was not uncommon for users to forget to verify if all of the properties of a certain widget had been migrated, as well as its style values. Despite happening to all types of users, it was proportional to their inexperience, and in some cases, the migration result was a screen with a considerably different aspect than the original screen.
- Some participants showed frustration in not being able to take advantage of the original Traditional Web Screen elements and classified the manual migration as unworkable.
- Several users considered the repetitive work of the manual migration to be prone to errors, struggling to make an application which they already had implemented in the legacy paradigm.
- Mostly, the manual migration was considered cumbersome, difficult, and prone to errors. As many users pointed out, it can be considered a reimplementation of the screen, making it a lengthy process.

For the automatic migration, regarding both possible interactions (copy-paste of elements or an entire screen migration), the feedback was:

- With the copy-paste of elements, a performance increase was noticeable on the overall result of a screen migration, both in its duration, as well as the quality of the final result (correctly mapped elements and property values).
- By using the tool to perform a full screen (and web block) migration, the performance increase was even more noticeable, with the screen being migrated almost instantly.
- However, some users mentioned that in a full-screen migration, some sense of control was lost, mostly due to many processes being performed instantly and without a manual interaction of the user. Be that as it may, when the same users analyzed the migration logs, that concern partially disappeared.

- 2 different users had problems navigating the logs. That being said, after a brief technical explanation, the same users seemed to have no further problems in the logs navigation.

- To sum up, the participants found the automatic migration very helpful and practical since it allowed them to take advantage of the Traditional Web Screen implementation. Furthermore, the process was classified as easy to use, convenient, and expeditious, producing a better result in multiple scenarios.

These experiments allowed us to conclude that a considerable performance improvement was achieved, both in speed and quality of the migration result. Nonetheless, some work must be done to give the users a similar sense of control to the manual migration. As for the usability, the SUS tests outcome presented a very satisfying result, reassuring the necessity of further automation for the manual migration.

The progress achieved proved to be a big step in the right direction: moving forward from the legacy paradigm with ease and comfort.

Chapter 7

Conclusions

In October 2019, OutSystems announced Reactive Web: a new paradigm to build reactive web applications. It was built to take advantage of modern web features, presenting multiple differences to the previous paradigm to develop Web Applications (Traditional Web). Because of this, OutSystems was confronted with a challenge: 63% of its applications were implemented in the old Web paradigm, thus becoming legacy systems.

Legacy Information Systems can be defined as “any information system that significantly resists modification and evolution” [10]. To solve this problem, migration is a type of system modernization and consists of moving a system from out-of-date languages or platforms to a more modernized environment [70].

The initial OutSystems’ approach to this problem was a manual migration, which involves rewriting legacy applications [82]. However, this type of migration uses modern architecture and tools but discards a considerable part of the effort previously made on the legacy system.

The automation of certain parts of the migration, all while maintaining quality standards, facilitates it, allows more systems to evolve more rapidly, and thus makes the ecosystem less dependant on the legacy technologies.

A well-founded case study took place with a platform and community analysis, as well as interviews with both OutSystems’ employees and developers. It allowed us to classify the UI as the most prioritized feature, but coincidently, the major bottleneck in migrations (needing the most investment).

This dissertation and the related project had the following objectives:

- The design and development of an automatic migration approach capable of converting UI elements to accelerate the manual migration.

- The creation and search of UI patterns to make the migration result according to the Reactive Web's best practices.

- The integration of the developed tool in the OutSystems platform (IDE) with an easy to use interaction.

The objectives were all fulfilled and in some cases, exceeded. A migration approach was implemented, allowing for a user to automatically migrate different granularities of UI components and when necessary, using predefined patterns to produce an upgraded migration result. Such a migration approach was integrated into the OutSystems platform with a copy-paste interaction, allowing the users to choose what and where to migrate. Besides, beyond the UI, some of the logic, data requests, and variables were migrated when used in migrated UI components. As a consequence of this migration, the patterns and additional structures to better abstract, search, and manipulate UI model elements were created and can constitute the foundation for future works.

The performed validation experiments, as well as the user feedback, prove that the result of this dissertation is an accelerator for the manual migration. As an example, the automatic migration can convert up to 4 320 screens of average size to the new paradigm, all in the same time that an equivalent screen is migrated manually. There was a noticeable performance increase on the overall result of a UI migration, both in its duration, as well as the quality of the final result in multiple scenarios. Likewise, the migration experience became considerably more positive and convenient, motivating users to use it frequently when compared to the manual migration.

Accomplishing this automation impacts OutSystems and its community, as it may accelerate the migration of **57 105** applications, with a sum of **1 018 475** screens and web blocks. Also, it gives developers and customers interested in using the Reactive Web paradigm, the power to take advantage of the formerly produced efforts in the previous paradigm.

7.1 Contributions

This dissertation includes the contributions that follow:

- A set of techniques, algorithms, and associated structures to preprocess and abstract UI model objects, which can be used to search and execute operations over sets of elements that constitute predefined design patterns. Despite the target being the OutSystems models, the developed algorithms and techniques are adaptable to other models abstracting UI development and manipulation.

- A model-driven migration approach with source and target systems built using different OutSystems web development paradigms. For the different migration processes, the algorithms are capable of manipulating and transform the respective

model objects and domain-specific languages. The approach is adequate to convert not only UI elements of different sizes and complexities, but also variables and data correlated to the transformed interfaces.

- Integration of the migration process in the OutSystems platform with an easy to use interaction. This has the potential to allow for millions of objects to be migrated automatically, hence paving the path for multiple applications to evolve into using state-of-the-art technologies.

- Performance and usability tests for the manual migration process used for migrating legacy OutSystems Web Applications, as well as for the developed process automation.

7.2 Future Work

The objectives set for the dissertation were met and in some cases, surpassed. However, due to the complexity of the OutSystems model and the intricacies of migrations, there are some possible aspects which could be covered in future works:

- The work implemented in this dissertation undertakes the automatic migration of a screen at a time, causing a problem in the screens' bindings. This is due to some properties containing pointers to other screens, which, before such screens are migrated, are not able to be assigned. A solution for this problem would be an implementation of a command, that after the migration of all the screens, reassigns the properties responsible for the navigation between them.

- With the patterns' implementation as it is, the patterns must be defined internally by OutSystems. Allowing for the patterns to be defined by the developers via some sort of low-code interaction, would bring great value and help the patterns reach their potential (mentioned in section 5.1.3.4).

- During the automatic migration, some decisions are automatically made by the algorithms. Examples of these decisions are: migrating a set of widgets as a pattern instead of individual widgets, converting certain properties into others, migrating widgets that do not exist in the new paradigm to equivalent (but different) objects, among others. However, there may exist some scenarios where the user does not want for those decisions to happen, so, these choices should be optional and, when intended, made by the users. Despite the developed algorithms already supporting this interaction, an interface must be developed to give the users such capability.

- An automation of the logic migration between paradigms, expanding the scope of OutSystems models and DSLs possible to migrate automatically. This automation of the entire logic conversion to the new paradigm would provide further acceleration

to the migration process. Also, as implemented for the UI, it would be important to have an automation of the migration decisions to create the best result for the new paradigm (e.g. the former server actions now have to be separated into client and server actions).

- After the logic automatic migration, the abstraction level could be increased, allowing for users to migrate entire applications. Even more, a case study could be undertaken to approach the possibility of OutSystems migrating all of the applications, thus removing the usage of the legacy web paradigm.

Bibliography

[1] A. Ahmad and M. A. Babar. "A Framework for Architecture-Driven Migration of Legacy Systems to Cloud-Enabled Software." In: *Proceedings of the WICSA 2014 Companion Volume*. WICSA '14 Companion. Sydney, Australia: Association for Computing Machinery, 2014. ISBN: 9781450325233. DOI: 10.1145/2578128.2578232. URL: https://doi.org/10.1145/2578128.2578232.

[2] A. A. Almonaies, J. R. Cordy, and T. R. Dean. "Legacy system evolution towards service-oriented architecture." In: *International Workshop on SOA Migration and Evolution*. 2010, pp. 53–62.

[3] S. Balasubramaniam, G. A. Lewis, E. Morris, S. Simanta, and D. Smith. "SMART: Application of a Method for Migration of Legacy Systems to SOA Environments." In: *Service-Oriented Computing – ICSOC 2007*. Springer Berlin Heidelberg, 2008, 678–690. DOI: 10.1007/978-3-540-89652-4_60. URL: http://dx.doi.org/10.1007/978-3-540-89652-4_60.

[4] A. Bangor, P. Kortum, and J. Miller. "Determining what individual SUS scores mean: Adding an adjective rating scale." In: *Journal of usability studies* 4.3 (2009), pp. 114–123.

[5] N. Bevan, J. Kirakowski, and J. Maissel. "What is Usability?" In: *Proceedings of the 4th International Conference on HCI*. Elsevier, 1991.

[6] J. Bézivin. "On the unification power of models." In: *Software & Systems Modeling* 4.2 (May 2005), pp. 171–188. DOI: 10.1007/s10270-005-0079-0. URL: https://doi.org/10.1007%2Fs10270-005-0079-0.

[7] P. Bille and I. Li Gørtz. "The Tree Inclusion Problem: In Optimal Space and Faster." In: *Automata, Languages and Programming*. Springer Berlin Heidelberg, 2005, 66–77. DOI: 10.1007/11523468_6. URL: http://dx.doi.org/10.1007/11523468_6.

[8] J. Bisbal, D. Lawless, Bing Wu, and J. Grimson. "Legacy information systems: issues and directions." In: *IEEE Software* 16.5 (Sept. 1999), pp. 103–111. ISSN: 1937-4194. DOI: 10.1109/52.795108.

[9] J. Bisbal, D. Lawless, Bing Wu, J. Grimson, V. Wade, R. Richardson, and D. O'Sullivan. "An overview of legacy information system migration." In: *Proceedings of Joint 4th International Computer Science Conference and 4th Asia Pacific Software Engineering Conference*. Dec. 1997, pp. 529–530. DOI: 10.1109/APSEC.1997.640219.

[10] M. L. Brodie and M. Stonebraker. *Legacy Information Systems Migration: Gateways, Interfaces, and the Incremental Approach*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1995. ISBN: 1558603301.

[11] J. Brooke. "System usability scale (SUS): a quick-and-dirty method of system evaluation user information." In: *Reading, UK: Digital Equipment Co Ltd* 43 (1986).

[12] H. Bunke and B. T. Messmer. "Recent Advances in Graph Matching." In: *International Journal of Pattern Recognition and Artificial Intelligence* 11.1 (Feb. 1997), 169–203. DOI: 10.1142/s0218001497000081. URL: http://dx.doi.org/10.1142/S0218001497000081.

[13] S. Cetin, N. I. Altintas, H. Oguztuzun, A. H. Dogru, O. Tufekci, and S. Suloglu. "A Mashup-Based Strategy for Migration to Service-Oriented Computing." In: *IEEE International Conference on Pervasive Services*. July 2007, pp. 169–172. DOI: 10.1109/PERSER.2007.4283910.

[14] W. Chen. "More Efficient Algorithm for Ordered Tree Inclusion." In: *J. Algorithms* 26 (Feb. 1998), pp. 370–385. DOI: 10.1006/jagm.1997.0899.

[15] M. J. Chung. "O(N2.5) Time Algorithms for the Subgraph Homeomorphism Problem on Trees." In: *J. Algorithms* 8.1 (Mar. 1987), 106–112. ISSN: 0196-6774. DOI: 10.1016/0196-6774(87)90030-7. URL: https://doi.org/10.1016/0196-6774(87)90030-7.

[16] R. Cole, R. Hariharan, and P. Indyk. "Tree Pattern Matching and Subset Matching in Deterministic O(n Log3 n)-Time." In: *Proceedings of the Tenth Annual ACM-SIAM Symposium on Discrete Algorithms*. SODA '99. Baltimore, Maryland, USA: Society for Industrial and Applied Mathematics, 1999, 245–254. ISBN: 0898714346.

[17] S. Comella-Dorda, K. Wallnau, R. Seacord, and J. Robert. "A Survey of Black-Box Modernization Approaches for Information Systems." In: *Proceedings of the International Conference on Software Maintenance (ICSM'00)*. ICSM '00. USA: IEEE Computer Society, 2000, p. 173. ISBN: 0769507530.

[18] S. Comella-Dorda, K. Wallnau, R. Seacord, and J. Robert. *A Survey of Legacy System Modernization Approaches*. Tech. rep. Carnegie-Mellon univ pittsburgh pa Software engineering inst, Apr. 2000, p. 30.

[19] D. Conte, P. Foggia, C. Sansone, and M. Vento. "Thirty Years of Graph Matching in Pattern Recognition." In: *International Journal of Pattern Recognition and Artificial Intelligence* 18.03 (2004), pp. 265–298. DOI: 10.1142/S0218001404003228. URL: https://doi.org/10.1142/S0218001404003228.

[20] L. P. Cordella, P. Foggia, C. Sansone, F. Tortorella, and M. Vento. "Graph matching: a fast algorithm and its evaluation." In: *Proceedings. Fourteenth International Conference on Pattern Recognition (Cat. No.98EX170)*. Vol. 2. 1998, 1582–1584 vol.2. DOI: 10.1109/ICPR.1998.712014.

[21] P. Cserkuti, T. Levendovszky, and H. Charaf. "Survey on Subtree Matching." In: *2006 International Conference on Intelligent Engineering Systems*. June 2006, pp. 216–221. DOI: 10.1109/INES.2006.1689372.

[22] K. Czarnecki. "Generative Programming: Methods, Techniques, and Applications." In: *Proceedings of the 7th International Conference on Software Reuse: Methods, Techniques, and Tools*. ICSR-7. Berlin, Heidelberg: Springer-Verlag, 2002, 351–352. ISBN: 3540434836.

[23] A. van Deursen, P. Klint, and J. Visser. "Domain-Specific Languages: An Annotated Bibliography." In: *SIGPLAN Not.* 35.6 (June 2000), 26–36. ISSN: 0362-1340. DOI: 10.1145/352029.352035. URL: https://doi.org/10.1145/352029.352035.

[24] W. Fan, J. Li, S. Ma, N. Tang, Y. Wu, and Y. Wu. "Graph Pattern Matching: From Intractable to Polynomial Time." In: *Proc. VLDB Endow.* 3.1–2 (Sept. 2010), 264–275. ISSN: 2150-8097. DOI: 10.14778/1920841.1920878. URL: https://doi.org/10.14778/1920841.1920878.

[25] J.-M. Favre. "Megamodelling and Etymology." In: *Transformation Techniques in Software Engineering*. 2005.

[26] J.-M. Favre and T. NGuyen. "Towards a Megamodel to Model Software Evolution Through Transformations." In: *Electronic Notes in Theoretical Computer Science* 127.3 (2005). Proceedings of the Workshop on Software Evolution through Transformations: Model-based vs. Implementation-level Solutions (SETra 2004), pp. 59 –74. ISSN: 1571-0661. DOI: https://doi.org/10.1016/j.entcs.2004.08.034. URL: http://www.sciencedirect.com/science/article/pii/S1571066105001398.

[27] T. A. S. Foundation. *Apache Cordova*. URL: https://cordova.apache.org/.

[28] B. Gallagher. "Matching structure and semantics: A survey on graph-based pattern matching." In: *AAAI Fall Symposium - Technical Report* 6 (Jan. 2006).

[29] A. S. Ganesan and T. Chithralekha. "A Survey on Survey of Migration of Legacy Systems." In: *Proceedings of the International Conference on Informatics and Analytics*. ICIA-16. Pondicherry, India: Association for Computing Machinery, 2016. ISBN: 9781450347563. DOI: 10.1145/2980258.2980409. URL: https://doi.org/10.1145/2980258.2980409.

[30] D. E. Ghahraman, A. K. C. Wong, and T. Au. "Graph Optimal Monomorphism Algorithms." In: *IEEE Transactions on Systems, Man, and Cybernetics* 10.4 (Apr. 1980), pp. 181–188. ISSN: 2168-2909. DOI: 10.1109/TSMC.1980.4308468.

[31] R. Gimnich and A Winter. "SOA migration: approaches and experience." In: *Softwaretechnik-Trends* 27.1 (2007), pp. 13–14.

[32] E. Guerra, J. de Lara, D. S. Kolovos, R. F. Paige, and O. M. dos Santos. "Engineering model transformations with transML." In: *Software & Systems Modeling* 12.3 (Sept. 2011), 555–577. DOI: 10.1007/s10270-011-0211-2. URL: http://dx.doi.org/10.1007/s10270-011-0211-2.

[33] H. Henriques, H. Lourenço, V. Amaral, and M. Goulão. "Improving the Developer Experience with a Low-Code Process Modelling Language." In: *Proceedings of the 21th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems*. MODELS '18. Copenhagen, Denmark: Association for Computing Machinery, 2018, 200–210. ISBN: 9781450349499. DOI: 10.1145/3239372.3239387. URL: https://doi.org/10.1145/3239372.3239387.

[34] F. Hermans, M. Pinzger, and A. Van Deursen. "Domain-specific languages in practice: A user study on the success factors." In: *International Conference on Model Driven Engineering Languages and Systems*. Springer. 2009, pp. 423–437.

[35] C. M. Hoffmann and M. J. O'Donnell. "Pattern Matching in Trees." In: *J. ACM* 29.1 (Jan. 1982), 68–95. ISSN: 0004-5411. DOI: 10.1145/322290.322295. URL: https://doi.org/10.1145/322290.322295.

[36] Y. Itokawa, W. Masanobu, I. Toshimitsu, and T. Uchida. "Tree Pattern Matching Algorithm Using a Succinct Data Structure." In: vol. 110. Mar. 2011. DOI: 10.1007/978-1-4614-1695-1_27.

[37] P. Jamshidi, A. Ahmad, and C. Pahl. "Cloud Migration Research: A Systematic Review." In: *IEEE Transactions on Cloud Computing* 1.2 (July 2013), pp. 142–157. ISSN: 2372-0018. DOI: 10.1109/TCC.2013.10.

[38] M. Julian. *Practical Monitoring: Effective Strategies for the Real World*. "O'Reilly Media, Inc.", 2017.

[39] S. Kabinna, C. Bezemer, W. Shang, and A. E. Hassan. "Logging Library Migrations: A Case Study for the Apache Software Foundation Projects." In: *2016 IEEE/ACM 13th Working Conference on Mining Software Repositories (MSR)*. 2016, pp. 154–164.

[40] R. Khadka, A. Saeidi, A. Idu, J. Hage, and S. Jansen. "Legacy to SOA Evolution." In: *Migrating Legacy Applications*. IGI Global, pp. 40–70. DOI: 10.4018/978-1-4666-2488-7.ch003. URL: https://doi.org/10.4018%2F978-1-4666-2488-7.ch003.

[41] P. Kilpelainen and H. Mannila. "Ordered and Unordered Tree Inclusion." In: *SIAM J. Comput.* 24.2 (Apr. 1995), 340–356. ISSN: 0097-5397. DOI: 10.1137/S0097539791218202. URL: https://doi.org/10.1137/S0097539791218202.

[42] S. R. Kosaraju. "Efficient tree pattern matching." In: *30th Annual Symposium on Foundations of Computer Science*. Oct. 1989, pp. 178–183. DOI: 10.1109/SFCS.1989.63475.

[43] T. Kühne. “What is a Model?” In: *Language Engineering for Model-Driven Software Development*. Ed. by J. Bezivin and R. Heckel. Dagstuhl Seminar Proceedings 04101. Dagstuhl, Germany: Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany, 2005. URL: http://drops.dagstuhl.de/opus/volltexte/2005/23.

[44] T. Kühne, G. Mezei, E. Syriani, H. Vangheluwe, and M. Wimmer. “Explicit transformation modeling.” In: *International Conference on Model Driven Engineering Languages and Systems*. Springer. 2009, pp. 240–255.

[45] T. Kühne. “Matters of (Meta-) Modeling.” In: *Software & Systems Modeling* 5 (Dec. 2006), pp. 369–385. DOI: 10.1007/s10270-006-0017-9.

[46] J. Larrosa and G. Valiente. “Constraint satisfaction algorithms for graph pattern matching.” In: *Mathematical Structures in Computer Science* 12.4 (Aug. 2002), 403–422. DOI: 10.1017/s0960129501003577. URL: http://dx.doi.org/10.1017/S0960129501003577.

[47] M. Lazarescu, H. Bunke, and S. Venkatesh. “Graph Matching: Fast Candidate Elimination Using Machine Learning Techniques.” In: *Advances in Pattern Recognition*. Springer Berlin Heidelberg, 2000, 236–245. DOI: 10.1007/3-540-44522-6_25. URL: http://dx.doi.org/10.1007/3-540-44522-6_25.

[48] H. Lourenço and R. Eugénio. “TrueChange (TM) Under the Hood: How We Check the Consistency of Large Models (Almost) Instantly.” In: *2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C)*. 2019, pp. 362–369.

[49] J. Ludewig. “Models in software engineering - an introduction.” In: *Software and Systems Modeling* 2.1 (Mar. 2003), 5–14. DOI: 10.1007/s10270-003-0020-3. URL: http://dx.doi.org/10.1007/s10270-003-0020-3.

[50] D. W. Matula. “Subtree Isomorphism in O(n5/2).” In: *Algorithmic Aspects of Combinatorics*. Elsevier, 2011, 91–106. DOI: 10.1016/s0167-5060(08)70324-8. URL: http://dx.doi.org/10.1016/S0167-5060(08)70324-8.

[51] B. D. McKay. “Practical Graph Isomorphism.” In: *Congr. Numerantium* 87 (Jan. 1981), pp. 30–45.

[52] M. Mernik, J. Heering, and A. M. Sloane. “When and How to Develop Domain-Specific Languages.” In: *ACM Comput. Surv.* 37.4 (Dec. 2005), 316–344. ISSN: 0360-0300. DOI: 10.1145/1118890.1118892. URL: https://doi.org/10.1145/1118890.1118892.

[53] P. Michailidis and K. G. Margaritis. “On-line string matching algorithms: Survey and experimental results.” In: *International Journal of Computer Mathematics* 76 (Feb. 2001), pp. 411–434. DOI: 10.1080/00207160108805036.

[54] M. Mukelabai. "Verification of Migrated Product Lines." In: *Proceedings of the 22nd International Systems and Software Product Line Conference - Volume 2*. SPLC '18. New York, NY, USA: Association for Computing Machinery, 2018, 87–89. ISBN: 9781450359450. DOI: 10.1145/3236405.3236428. URL: https://doi.org/10.1145/3236405.3236428.

[55] G. Mussbacher, D. Amyot, R. Breu, J.-M. Bruel, B. H. C. Cheng, P. Collet, B. Combemale, R. B. France, R. Heldal, J. Hill, and et al. "The Relevance of Model-Driven Engineering Thirty Years from Now." In: *Lecture Notes in Computer Science*. Springer International Publishing, 2014, 183–200. DOI: 10.1007/978-3-319-11653-2_12. URL: http://dx.doi.org/10.1007/978-3-319-11653-2_12.

[56] OutSystems. *Becoming a Traditional Web Developer*. URL: https://www.outsystems.com/learn/paths/2/becoming-a-traditional-web-developer/.

[57] OutSystems. *Forum Post - The Next Generation of Web Apps*. URL: https://www.outsystems.com/forums/discussion/52761/reactive-web-the-next-generation-of-web-apps/.

[58] OutSystems. *OutByNumbers - Benchmark Overview Report*. URL: http://www.outsystems.com/res/OutbyNumbers-DataSheet.

[59] OutSystems. *OutSystems 11 Documentation*. URL: https://success.outsystems.com/Documentation/11/.

[60] OutSystems. *OutSystems Platform Best Practices*. URL: https://success.outsystems.com/Documentation/Best_Practices/OutSystems_Platform_Best_Practices.

[61] OutSystems. *OutSystems Tools and Components*. URL: https://www.outsystems.com/evaluation-guide/outsystems-tools-and-components/.

[62] OutSystems. *Reactive Web Applications - The Next Generation of Web Apps*. URL: https://www.outsystems.com/blog/posts/reactive-web-applications/.

[63] OutSystems. *Scaffolding and Rich Widgets*. URL: https://www.outsystems.com/learn/lesson/867/scaffolding-and-richwidgets/.

[64] OutSystems. *Traditional to Reactive App Migration reference*. URL: https://success.outsystems.com/Support/Enterprise_Customers/Upgrading/Introduction_into_migrating_Traditional_Web_to_Reactive_Web_Apps/.

[65] OutSystems. *UI Patterns*. URL: https://outsystemsui.outsystems.com/OutSystemsUIWebsite/PatternOverview.

[66] OutSystems. *What Is Low-Code?* URL: https://www.outsystems.com/blog/what-is-low-code.html.

[67] R. Ramesh and I. V. Ramakrishnan. "Nonlinear Pattern Matching in Trees." In: *J. ACM* 39.2 (Apr. 1992), 295–316. ISSN: 0004-5411. DOI: 10.1145/128749.128752. URL: https://doi.org/10.1145/128749.128752.

[68] M. Razavian and P. Lago. "A systematic literature review on SOA migration." In: *Journal of Software: Evolution and Process* 27.5 (May 2015), 337–372. doi: 10.1002/smr.1712. url: http://dx.doi.org/10.1002/smr.1712.

[69] A. Rodrigues da Silva. "Model-driven engineering: A survey supported by the unified conceptual model." In: *Computer Languages, Systems & Structures* 43 (2015), pp. 139–155. issn: 1477-8424. doi: https://doi.org/10.1016/j.cl.2015.06.001.

[70] A. Rodríguez, A. Caro, and E. Fernández-Medina. "Towards Framework Definition to Obtain Secure Business Process from Legacy Information Systems." In: *Proceedings of the First International Workshop on Model Driven Service Engineering and Data Quality and Security*. MoSE+DQS '09. Hong Kong, China: Association for Computing Machinery, 2009, 17–24. isbn: 9781605588162. doi: 10.1145/1651415.1651419. url: https://doi.org/10.1145/1651415.1651419.

[71] E. Seidewitz. "What models mean." In: *IEEE Software* 20.5 (2003), pp. 26–32.

[72] B. Selic. "The pragmatics of model-driven development." In: *IEEE Software* 20.5 (2003), pp. 19–25.

[73] S. Sendall and W. Kozaczynski. "Model transformation: the heart and soul of model-driven software development." In: *IEEE Software* 20.5 (2003), pp. 42–45.

[74] R. Shamir and D. Tsur. "Faster subtree isomorphism." In: *Proceedings of the Fifth Israeli Symposium on Theory of Computing and Systems*. June 1997, pp. 126–131. doi: 10.1109/ISTCS.1997.595164.

[75] W. Shang, Z. M. Jiang, B. Adams, A. E. Hassan, M. W. Godfrey, M. Nasser, and P. Flora. "An Exploratory Study of the Evolution of Communicated Information about the Execution of Large Software Systems." In: *2011 18th Working Conference on Reverse Engineering*. 2011, pp. 335–344.

[76] D. Shasha, J. T. L. Wang, and R. Giugno. "Algorithmics and Applications of Tree and Graph Searching." In: *Proceedings of the Twenty-First ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*. PODS '02. Madison, Wisconsin, USA: Association for Computing Machinery, 2002, 39–52. isbn: 1581135076. doi: 10.1145/543613.543620. url: https://doi.org/10.1145/543613.543620.

[77] K. Shearer, H. Bunke, S. Venkatesh, and D. Kieronska. "Efficient Graph Matching for Video Indexing." In: *Computing Supplement*. Springer Vienna, 1998, 53–62. doi: 10.1007/978-3-7091-6487-7_6. url: http://dx.doi.org/10.1007/978-3-7091-6487-7_6.

[78] D. Smith. "Migration of Legacy Assets to Service-Oriented Architecture Environments." In: *Companion to the Proceedings of the 29th International Conference on Software Engineering*. ICSE COMPANION '07. USA: IEEE Computer Society, 2007, 174–175. ISBN: 0769528929. DOI: 10.1109/ICSECOMPANION.2007.48. URL: https://doi.org/10.1109/ICSECOMPANION.2007.48.

[79] H. M. Sneed. "Integrating legacy software into a service oriented architecture." In: *Conference on Software Maintenance and Reengineering (CSMR'06)*. Mar. 2006, 11 pp.–14. DOI: 10.1109/CSMR.2006.28.

[80] W. Souiou and N. Bounour. "Migration of Legacy Systems to Service Oriented Architecture." In: *The Second International Conference on Digital Enterprise and Information Systems (DEIS2013)*. Mar. 2013, pp. 166–173.

[81] C. Sridharan. *Distributed Systems Observability: A Guide to Building Robust Systems*. O'Reilly Media, 2018.

[82] E. Stehle, B. Piles, J. Max-Sohmer, and K. Lynch. "Migration of Legacy Software to Service Oriented Architecture." In: *Department of Computer Science Drexel University Philadelphia, PA* 19104 (2008), pp. 2–5.

[83] E. Syriani, J. Gray, and H. Vangheluwe. "Modeling a Model Transformation Language." In: May 2013, pp. 211–237. ISBN: 978-3-642-36653-6. DOI: 10.1007/978-3-642-36654-3_9.

[84] J. R. Ullmann. "An Algorithm for Subgraph Isomorphism." In: *J. ACM* 23.1 (Jan. 1976), 31–42. ISSN: 0004-5411. DOI: 10.1145/321921.321925. URL: https://doi.org/10.1145/321921.321925.

[85] A. Van Deursen, E. Visser, and J. Warmer. "Model-driven software evolution: A research agenda." In: *Technical Report Series TUD-SERG-2007-006* (2007).

[86] P. Vincent, Y. Natis, K. Iijima, J. Wong, S. Ray, A. Jain, and A. Leow. *Magic Quadrant for Enterprise Low-Code Application Platforms, September 30, 2020*. Tech. rep. Gartner, Inc., 2020.

[87] Z. Wan, F. J. Meng, J. M. Xu, and P. Wang. "Service Composition Pattern Generation for Cloud Migration: A Graph Similarity Analysis Approach." In: *2014 IEEE International Conference on Web Services*. June 2014, pp. 321–328. DOI: 10.1109/ICWS.2014.54.

[88] Z. Wan and P. Wang. "A Survey and Taxonomy of Cloud Migration." In: *2014 International Conference on Service Sciences*. May 2014, pp. 175–180. DOI: 10.1109/ICSS.2014.46.

[89] J. Whittle, J. Hutchinson, and M. Rouncefield. "The State of Practice in Model-Driven Engineering." In: *IEEE Software* 31.3 (2014), pp. 79–85.

[90] A. Winter and J. Ziemann. "Model-based migration to service-oriented architectures." In: *International Workshop on SOA Maintenance and Evolution*. 2007, pp. 107–110.

Appendix A

Migration Result

This appendix details the result of a Screen migration from a Traditional Web Application to a Reactive Web Application.

A.1 Original Screen

The development of the original screen to be migrated has the visual appearance found in figure A.1.

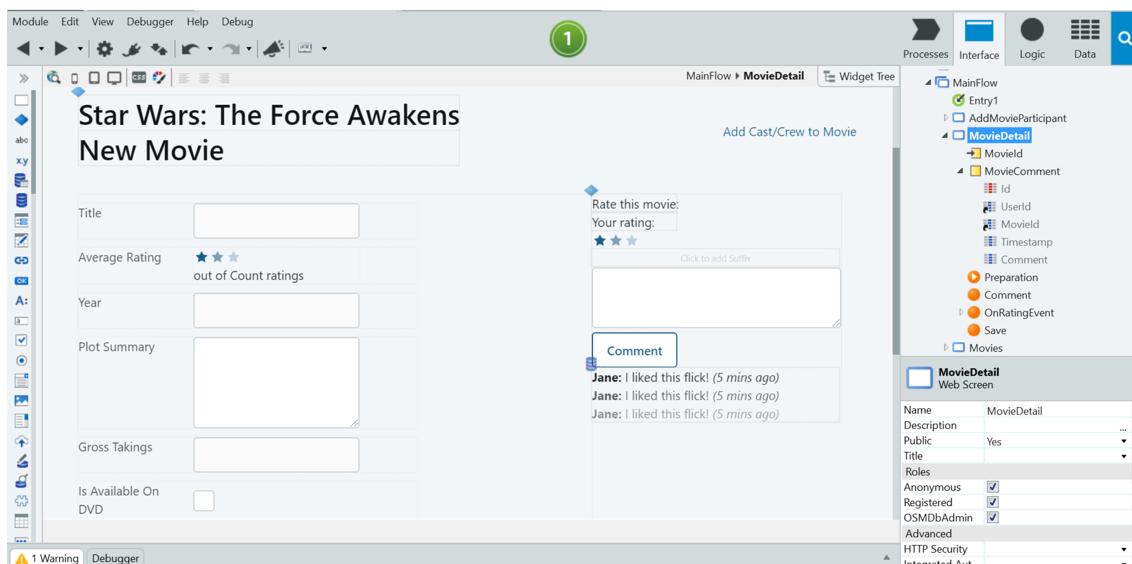


Figure A.1: Screen to be migrated.

The screen in question has one input parameter, one input variable, and five preparation aggregates. Besides, it contains two instances of two different web blocks. The

migration will be executed by copying this screen and pasting it in the Reactive Web application.

A.2 Migration Result

Depicted in figure A.2 is the result of the migration.

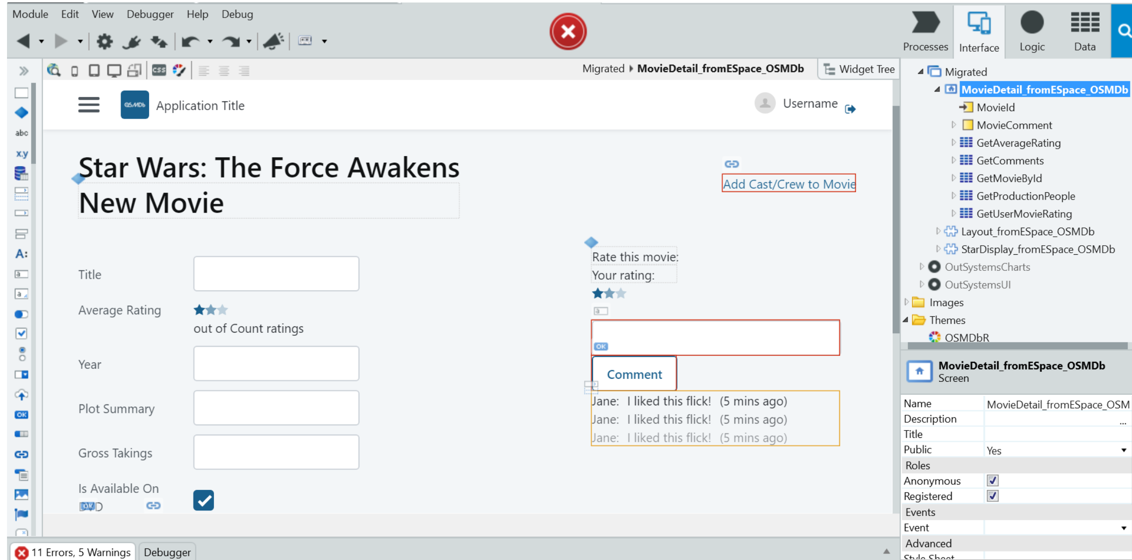


Figure A.2: Screen migration result.

As we can see, both the functionalities and aspects of the screen are migrated. Not only that but so are its input parameter, local variable, and preparation aggregates. The aggregates are migrated to screen aggregates. In the same figure it possible to observe that 2 web blocks were migrated. The screen in question had two instances of web blocks: one using the "Layout" web block, and another using the "Star Display" web block. These blocks were migrated in order to be used in the migrated screen.

On top of the screen content, the data from the Traditional Web Application is imported as a reference, as seen in figure A.3.

Nonetheless, it is noticeable that some parts of the migrated result appear signaled with a red margin. This is because the automatic migration caused the appearance of 11 errors, which can be seen in figure A.4. These errors have 2 reasons to be:

- On Click errors

  These errors are due to only this screen having been migrated, so the navigation to other screens (to be migrated), was not set. To fix these errors, the user must migrate the screens and assign the properties' values (responsible for the navigation between screens).

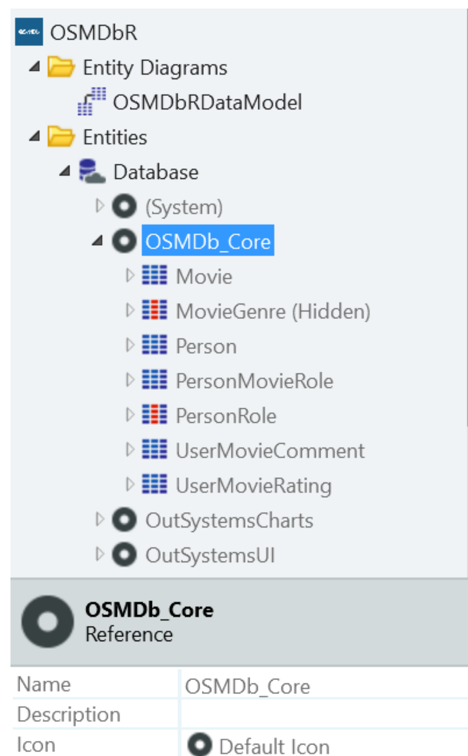


Figure A.3: Screen data imported as reference.

- Role related errors

  The roles in Traditional Web are defined and used in some properties. Since the automation does not migrate those roles, it must be migrated manually.

However, these errors are steps that would have to be done in the manual migration and which take only a few minutes. Thus, instead of having to manually migrate the entire screen, a developer has only to solve the errors (which also had to be tackled in the manual migration).

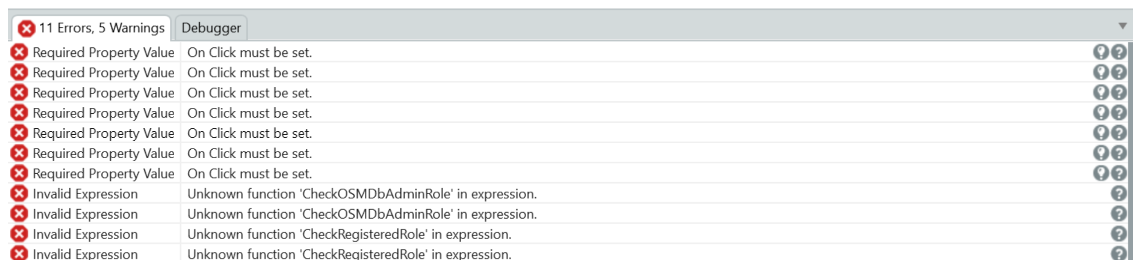


Figure A.4: Errors created by the automatic migration.

A.3 Migration Metrics

This section depicts the migration metrics of the screen and web blocks migrated. Due to the magnitude of the migration logs generated, those will not be presented.

Listing A.1: Screen Migration Metrics.

```
1  Duration: 2.15 seconds
2
3  Number of Aggregates migrated: 5
4  Number of Input Parameters migrated: 1
5  Number of Local Variables migrated: 1
6
7  Total number of Widgets: 77
8  Number of Widgets migrated: 77
9
10 Total number of Patterns identified: 4
11 Number of Patterns migrated: 4
12
13 Number of Web Blocks migrated to Reactive ESpace: 2
```

Listing A.2: "Layout" Web Block Migration Metrics.

```
1  Duration: 0.37 seconds
2
3  Number of Aggregates migrated: 0
4  Number of Input Parameters migrated: 0
5  Number of Local Variables migrated: 0
6
7  Total number of Widgets: 21
8  Number of Widgets migrated: 21
9
10 Total number of Patterns identified: 1
11 Number of Patterns migrated: 1
12
13 Number of Web Blocks migrated to Reactive ESpace: 0
```

Listing A.3: "Star Display" Web Block Migration Metrics.

```
1  Duration: 0.19 seconds
2
3  Number of Aggregates migrated: 0
4  Number of Input Parameters migrated: 2
5  Number of Local Variables migrated: 2
6
7  Total number of Widgets: 9
8  Number of Widgets migrated: 9
9
10 Total number of Patterns identified: 2
11 Number of Patterns migrated: 2
12
13 Number of Web Blocks migrated to Reactive ESpace: 0
```

ANNEX I

Comparison Of Migration Approaches

Table I.1, obtained in [29], expresses and compares the advantages and disadvantages of each of the migration approaches mentioned in section 3.2.

Table I.1: Comparison of migration approaches [29]

Approach	Type	Advantages	Disadvantages
Database First approach	Gateway	- Reuse of legacy system is possible - Target system development could be incremental - Suitable for migration of fully decomposable systems	- The Information system is not operational during data migration - Migration of data takes significant time
Database Last approach	Gateway	- Reuse of legacy system is possible - Suitable for fully decomposable systems - Target system development could be incremental	- The Information system is not operational during data migration - Migration of data takes significant time
Composite database approach	Gateway	- Reuse of legacy system is possible - Eliminates requirement for a single large migration of data as required in the above two approaches - Suitable for migration of fully decomposable, semi decomposable and non decomposable systems	- Suffers from the overhead of the Database First and Database Last approaches with added complexity due to introduction of co-coordinator
Chicken Little Strategy	Gateway	- Operational Information System will be a composite of target and legacy information system - Suitable for migration of fully decomposable systems, semi decomposable and non decomposable systems	- Employs complex gateways - Need for interoperation of legacy and target systems through gateways add greater complexity to the already existing complex process
Big Bang Methodologies	Non-Gateway	- Improvement over the existing legacy system is possible as the development is from scratch	- Huge cost is involved - Takes longer development time - Legacy system reuse is not possible
Butterfly Methodology	Non-Gateway	- No Gateways used - Eliminates the need for simultaneous access of both legacy and target systems - Testing can be carried out against the already migrated data - Legacy system to be shut only for minimal time	- Target system is not in production while the system is being migrated