



N OVA
NOVA SCHOOL OF
SCIENCE & TECHNOLOGY

DEPARTMENT OF
COMPUTER SCIENCE

JOÃO LUÍS GUERREIRO RAMALHO
Master in Computer Science

FROM APP BUILDERS TO APP EDITORS
BIDIRECTIONAL TRANSFORMATIONS OF LOW-CODE MODELS

MASTER IN COMPUTER SCIENCE
NOVA University Lisbon
November, 2021



FROM APP BUILDERS TO APP EDITORS

BIDIRECTIONAL TRANSFORMATIONS OF LOW-CODE MODELS

JOÃO LUÍS GUERREIRO RAMALHO

Master in Computer Science

Adviser: João Ricardo Viegas da Costa Seco
Associate Professor, NOVA University Lisbon

Co-adviser: Hugo Miguel Ramos Lourenço
Principal Research Engineer, OutSystems

From App Builders to App Editors

Copyright © João Luís Guerreiro Ramalho, NOVA School of Science and Technology, NOVA University Lisbon.

The NOVA School of Science 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.

To my dear parents and brother.

ACKNOWLEDGEMENTS

I would like to thank the NOVA School of Sciences and Technology, the Department of Computer Science, and every professor who supported me in this five-year journey. Your dedication and commitment helped me build the foundations of what I hope to be a successful career.

Secondly, I would like to thank my advisers for their devotion to this project. This wouldn't have been possible without you. Professor João Costa Seco, I can not speak too highly of your dedication and professionalism. Hugo Lourenço, I thank you for all the knowledge and expertise you provided. Your serene and well-pondered attitude taught me a lot.

Thank you OutSystems for this opportunity, and for having embarked on this research project. Everyone I talked to showed an amazing willingness to help and for that, I am very grateful. A special thanks to the Experience Builder team, particularly Gonçalo Martins and Tarik Ayoub for enduring my never-ending questions and requests.

Furthermore, I must address all the support given by my family, in particular my parents and my brother. During this journey, you encouraged and inspired me to keep going, always supporting me when I needed it. I hope you understand how important you were.

I had the good fortune of having two colleagues who walked this long path with me, Francisco Delgado and João Santos. I never thought I would miss all those restless nights working on our assignments. It is with great honor and appreciation I can say I have you as my friends.

A special thanks to my best friend André Coutinho for your wisdom and support. Your inherent drive to succeed motivated me in more ways than you can imagine.

A word of appreciation for NOVA-LINCS and the GOLEM project ¹ for partially supporting this work with the provided grant ², and last but not least Professor João Lourenço and everyone behind the this template [10] for their relentless service to every student who used it. I cannot stress enough how much your work facilitated mine.

¹Golem Project, REF Lisboa-01-0247-Feder-045917

²Grant Reference, UIDB/04516/2020

“Start with what is right rather than what is acceptable.”
(Franz Kafka)

ABSTRACT

OutSystems provides a model-driven development and delivery platform aided by a rich visual environment, allowing developers to create enterprise-grade web and mobile applications. Until recently, most of this development capability came from Service Studio, the platform's [Integrated Development Environment \(IDE\)](#), with which developers can quickly design a fully-fledged application. Nevertheless, in recent times the company has strived to offer a new collection of tools more focused on specific aspects of application development. The builders are tools that allow for non-IT related users to generate complete software solutions, with a small number of interactions, therefore reducing complexities correlated with the assembly of multiple-layer applications. Currently, there are two builders generally available: the Experience Builder providing greater focus towards the initial [User Experience \(UX\)](#) development, and the Workflow Builder, associated with the design of task management and automation applications.

Even though the OutSystems platform allows the combined use of the builders with Service Studio, currently this compatibility is unidirectional. That is, an application created using the Experience Builder, for instance, can be edited in the [IDE](#) but the inverse process is not possible. More specifically this shortcoming precludes a builder to update an application created or edited with any OutSystems tool. This substantially damages the chance for collaboration between different types of users employing different OutSystems tools. The present dissertation sets as its paramount objective the enabling of different [personas](#), both business and tech-oriented, to collaborate on the development of an enterprise-level application, employing the entire set of tools provided by the platform. From a more detailed standpoint, this work consists in the development of necessary model transformations proficient in supporting continuous and collaborative interoperability.

Hence, this dissertation aims to expand the reach of the OutSystems product line, but also, from an academic standpoint, it hopes to provide a useful contribution to [Model Driven Engineering](#) and model transformations, advancing the state-of-the-art.

Keywords: OutSystems, Builders, Editors, Model Transformations, BPT, MDE, Continuous Collaboration

RESUMO

A OutSystems fornece uma plataforma de desenvolvimento e entrega, orientada ao modelo e suportada num ambiente visual rico, permitindo aos seus programadores criarem aplicações móveis e web de nível empresarial. Durante vários anos, grande parte desta capacidade de desenvolvimento resultava do uso do Service Studio, o [Integrated Development Environment](#) da plataforma, capaz de rapidamente construir uma aplicação na sua plenitude. Ainda assim, nos últimos tempos a companhia empenhou-se em oferecer uma nova coleção de ferramentas, mais focadas em aspetos específicos do desenvolvimento aplicacional. Os builders, são então ferramentas que possibilitam que utilizadores não associados a áreas de [IT](#) possam gerar soluções completas de software através de um número reduzido de interações, conseguindo assim minorar complexidades correlacionadas com a montagem de múltiplas camadas aplicacionais. Atualmente, existem dois builders disponíveis para o público: o Experience Builder orientado ao desenvolvimento da [User Experience \(UX\)](#), o Workflow Builder, associado ao design de aplicações associadas a processos de gestão e automação de tarefas.

Embora a plataforma OutSystems permita o uso combinado dos builders com o Service Studio, atualmente esta compatibilidade verifica-se como unidirecional. Isto é, uma aplicação criada no Experience Builder, por exemplo, pode ser editada no [IDE](#), mas o processo inverso não é possível. Esta limitação impede um builder de atualizar uma aplicação criada ou editada utilizando uma qualquer ferramenta OutSystems. Isto prejudica substancialmente a colaboração entre diferentes tipos de utilizadores que empreguem diferentes ferramentas OutSystems.

A presente tese define como objectivo principal capacitar diferentes [personas](#) não só orientadas ao negócio como às tecnologias, com a possibilidade de colaborar no desenvolvimento de uma aplicação de nível empresarial, podendo, para isso, utilizar a totalidade de ferramentas fornecidas pela plataforma. De um ponto de vista mais detalhado, este trabalho irá consistir no desenvolvimento das transformações de modelo necessárias para o suporte do desenvolvimento contínuo e colaborativo que se pretende.

Deste modo, esta tese não só tem como propósito a expansão do alcance da linha de

produtos OutSystems, como de um ponto de vista académico, pretende contribuir utilmente para o paradigma da [Model Driven Engineering](#) e das transformações de modelos, avançando assim o estado da arte.

Palavras-chave: OutSystems, Builders, Editors, Model Transformations, BPT, MDE, Continuous Collaboration

CONTENTS

List of Figures	xxi
Glossary	xxv
Acronyms	xxvii
1 Introduction	1
1.1 Context	1
1.2 Motivation	2
1.3 Objectives	2
1.4 Expected Contributions	3
1.5 Outline	3
2 Background	5
2.1 Model Driven Engineering	5
2.1.1 Model	6
2.1.2 Metamodel	7
2.1.3 Meta-metamodel	7
2.1.4 Meta-Object Facility	7
2.2 Domain Specific Modeling Languages	8
2.3 Model Transformations	8
2.3.1 Kinds of Model Transformations	9
2.3.2 <i>Lenses</i>	11
3 OutSystems Platform	13
3.1 Architecture	13
3.1.1 Service Studio	13
3.1.2 Platform Server	14
3.1.3 Integration Studio	14
3.2 Builders	15

3.2.1	Experience Builder	16
3.2.2	Workflow Builder	17
3.2.3	Integration Builder	18
3.2.4	Example of the current development shortcomings	20
4	Model manipulation	21
4.1	OutSystems Language	21
4.2	OutSystems Meta-model	21
4.3	OutSystems Model	22
4.4	Builders	24
4.5	Builders Meta-model	25
4.6	Builders Model	26
4.7	Model Flow	29
4.8	ModelAPI	30
4.9	Builders' Key Management	30
4.9.1	Single-Shot Builders	31
4.9.2	MultiShot Builders	31
5	Related Work	33
5.1	Delta-based model transformations	33
5.1.1	Benefits from using a two-stage operation	34
5.2	Concurrent update propagation	35
5.2.1	Requirements for synchronizing concurrent updates	35
5.2.2	Algorithm	36
5.3	In summary	37
6	Design and Implementation	39
6.1	Design	39
6.1.1	Strategy Overview	39
6.2	Implementation	42
6.2.1	Prototype Plan	42
6.2.2	Console App Functionality	43
6.2.3	TinyApp	45
6.3	Backward Transformation	50
6.3.1	Serialization/Deserialization of the builder model	50
6.3.2	Delta Computation	51
6.3.3	Operation "Translations"	54
6.4	Forward Transformation	61
6.4.1	Metadata Injection	62
6.4.2	Projection	66
7	Results and Evaluation	71

7.1	Results	71
7.1.1	<i>Add Flow</i> operation	72
7.1.2	<i>Add Connection</i> operation	74
7.1.3	<i>Add Menu Item</i> operation	75
7.1.4	Backward and Forward Transformations	77
7.1.5	<i>Backward Transformation</i> operation	78
7.1.6	<i>Forward Transformation</i> operation	79
7.2	Evaluation	80
7.2.1	Validation	80
7.3	Unattained scenarios	86
7.3.1	Limitations	88
7.4	Running example	88
7.5	Final Remarks	89
8	Conclusion and Future Work	91
	Bibliography	93

LIST OF FIGURES

2.1	UML Java Class Models	6
2.2	Meta-model	7
2.3	MOF hierarchy	8
2.4	PIM to PSM model transformation	9
3.1	OutSystems Platform	14
3.2	Service Studio	15
3.3	Experience Builder - Template Screen	16
3.4	Experience Builder - <i>Add Flow</i> Screen	17
3.5	Experience Builder - Menu Customization	17
3.6	Workflow Builder - Template Screen	18
3.7	Workflow Builder - Form Customization	19
3.8	Workflow Builder - Business Process Customization	19
3.9	Integration Builder - <i>Choose Provider</i> Screen	19
3.10	Integration Builder - <i>My Integration</i> Screen	20
4.1	Service Studio Interface - <i>Action ProcessRequest</i>	22
4.2	OutSystems <i>Action</i> Meta-model (UML)	24
4.3	OutSystems <i>Action</i> Model (UML)	26
4.4	Experience Builder Interface - Screens and Flows	28
4.5	Experience Builder - Meta-model	28
4.6	Model Flow	29
5.1	Diskin Delta-based Model Transformation (Adapted from [4])	35
5.2	Takeshi <i>et al.</i> Algorithm [29]	37
6.1	Used Notation	40
6.2	Backward Transformation - Change Log	41
6.3	Backward Transformation - Builder-Log Coupling	41

LIST OF FIGURES

6.4	Backward Transformation - Difference Calculation	42
6.5	Prototype - Console Application	44
6.6	Service Studio - Application (Flows)	46
6.7	Service Studio - Application (<i>AnimatedOnboardingOption1</i> Screen)	49
6.8	Service Studio - Application (<i>BankingDashboard</i> Screen)	49
6.9	Service Studio - Application (<i>SimpleLogin</i>) Screen	49
6.10	Service Studio - Application (Bottom Bar Menu)	49
7.1	Experience Builder Interface - Add Flows button	72
7.2	Prototype Console - Add Flow use-case	73
7.3	Service Studio - Added Flow Metadata	73
7.4	Service Studio - Added Screen Metadata	74
7.5	Experience Builder Interface - Add Connection	74
7.6	Prototype Console - Add Connection	75
7.7	Service Studio - Added Connection Metadata	75
7.8	Experience Builder Interface - Add Menu Item dialogue window	76
7.9	Experience Builder Interface - Add Menu Item	76
7.10	Prototype Console - Add Menu Item	77
7.11	Service Studio - Added Menu Item Metadata	77
7.12	Experience Builder Interface - Publish button	78
7.13	Prototype Console - Backward Transformation	78
7.14	Prototype Console - Forward Transformation	80
7.15	Added <i>Users</i> entity	82
7.16	Category 1 - Ann's prototype commands	82
7.17	Service Studio - Backward Transformation Result	82
7.18	Service Studio - Bob's added text widget	83
7.19	Category 2 - Ann's prototype commands	84
7.20	Service Studio - Bob's added text widget (after Ann's changes)	84
7.21	Category 3 - Ann's prototype commands	85
7.22	Service Studio - Application flows after Ann's changes	86
7.23	Solution's behavior	86
7.24	Experience Builder Interface - Add Flows dialogue window with Empty Screen selected	87
7.25	Experience Builder Interface - Side Menu	87

LIST OF LISTINGS

1	OutSystems <i>Action</i> Meta-model	23
2	OutSystems Model Instance	25
3	Experience Builder - JSON Model (Abbreviated)	27
4	ModelAPI	30
5	Model with Key Management Mechanisms - MultiShot Operation Mode	32
6	JSON Payload - Basic Info	45
7	JSON Payload - Model (Screens)	47
8	JSON Payload - Model (Connections)	48
9	JSON Payload - Model (Menu)	48
10	Implementation - Experience Builder Model	50
11	Implementation - <i>LoadModel</i>	51
12	Implementation - <i>SaveModel</i>	52
13	Change Log	55
14	Abstract Operation Class	55
15	<i>Run</i> method - <i>UpdateFlow</i>	56
16	<i>UpdateFlow</i> method	57
17	Create connection - <i>Run</i> method	57
18	Create connection - <i>ConnectScreens</i> method	58
19	Create connection - <i>Connect</i> method	59
20	Create connection - <i>SetDestination</i> method	60
21	BackwardTransformation method	62
22	Metadata Injection - <i>AddFlowMetadata</i> method snippet	63
23	Metadata Injection - <i>AddScreenMetadata</i> method snippet	63
24	Metadata Injection - <i>CreateConnection</i> method snippet	64
25	Metadata Injection - <i>CreateMetadata</i> method snippet	65
26	Metadata Injection - <i>CreateMenuItem</i> method snippet	65
27	Projection - <i>Projection</i> method	67
28	Projection - <i>GetConnectionsFromLinks</i> method	68
29	Projection - <i>ForwardTransformation</i> method	69

LIST OF FIGURES

30 Change Log Example 79

GLOSSARY

Add-on	Generally, a small program that provides added capabilities to an existing software i , 14
App	An application software designed to satisfy a particular purpose i , 2 , 16 , 31 , 40 , 45 , 56 , 82
Artifact	Any tangible or concrete element produced from software development i
Connection	(OutSystems) A link connecting two application screens i , 42 , 45 , 50 , 52 , 74
DevOps	A collection of practises belonging to the realm of software development ("Dev") and operations ("Ops") i , 13
End-User	Person or entity who ultimately uses a (software) product i , 14 , 18
Flow	(OutSystems) Sequences of screens carried out as the user interacts with the application i , 16 , 42 , 45 , 50 , 52
IT	Information Technologies i , xxv
Log	Track record of events occurring in the context of a software i , 40 , 41
Persona	A fictional character representing a user type that might use a site, brand, or product i , xiii , xv , 2
S3 Bucket	A public cloud storage resource available in Amazon Web Services' (AWS) Simple Storage Service (S3) i , 46
Server Action	OutSystems action that runs logic on the server side i , 20

widget Usually referring to a user's interface button, link, bar, or any other application element the user can interact with [i](#), [66](#)

ACRONYMS

API	Application Programming Interface 30
AWS	Amazon Web Services 29, 45, 46
B2C	Business-to-Client 16
BX	Bidirectional Transformations 1, 2
CRM	Customer Relationship Management 18
CRUD	Create Read Update Delete 20
DSL	Domain-Specific Language 13
DSML	Domain Specific Modeling Language 8
ERP	Enterprise Resource Planning 18
GUI	Graphical User Interface 2, 43, 72
GUID	Globally Unique Identifier 31, 32
IDE	Integrated Development Environment xiii , xv , 2, 13, 29, 39, 40, 41, 61, 79, 81, 83, 88, 91, 92
ISO	International Organization for Standardization 5
IT	Information Technologies xiii , xv , 2, 15, 17, 91
MBE	Model Based Engineering 5
MDA	Model Driven Architecture 5, 9
MDD	Model Driven Development 5
MDE	Model Driven Engineering xiii , xvi , 1, 3, 4, 5, 6, 8
MDS	Model Driven Software Development 5, 10
MOF	Meta-Object Facility 7

ACRONYMS

OAP	OutSystems Application (Package) 15, 18, 29, 46, 61
OMG	Object Management Group 5, 7
OML	OutSystems Markup Language 30, 46
SaaS	Software-as-a-Service 15, 17, 18
UI	User Interface 14, 45, 46, 61, 83, 89
UML	Unified Modeling Language 5, 6, 8
UX	User Experience xiii, xv, 14, 15, 16, 89

INTRODUCTION

This dissertation was developed under a successful partnership between the OutSystems Research & Development (R&D) team, the NOVA Lincs Research Center, and the Department of Informatics of the School of Sciences and Technology of NOVA University of Lisbon (NOVA SST). The investigation aimed for the development of a viable solution regarding the subjects of [Model Driven Engineering \(MDE\)](#) and model transformations. This work intended not only to enrich the OutSystems Platform together with its product line but also hoped to provide a valuable contribution to the mentioned paradigms.

1.1 Context

The inadvertent increase of complexity has infamously stood as one of the greatest enemies of software engineering, notably claiming responsibility for crippling the development and management distinctively in the realm of enterprise-level applications. This complexity comes off as a by-product of frequent increases in the amount of data to be managed by said enterprises as well as the constant need to fulfill market and/or enterprises internal requirements [23].

To mitigate the adverse effects of this problem, companies have integrated several approaches and methodologies in their software development process. An example of this is [MDE](#) - a paradigm where the artifacts used in the traditional software development lifecycle are **models**, that is, views of the system to be constructed, using higher-level abstractions. But in this approach, models relocate themselves away from their classical use in design and documentation phases and acquire a more central role being used to produce actual code. Enterprises adopting approaches related to [MDE](#) paradigm have experienced considerable benefits regarding the increase in productivity by working with human-friendly abstractions, focused on the problem domain at hand, rather than wasting effort on technical aspects [26].

From a more thorough use of multiple interrelated models in software engineering, emerged the necessity of simplification. Hence, the research regarding **model transformations** was prompted, as well as the study towards the specific branch of [Bidirectional](#)

Transformations (BX). These concepts relate to the automation of processes responsible for taking one model and transforming it into another. The particular case of bidirectional transformations bears a great concern towards mechanisms of consistency maintenance across models, and its scope has a more straightforward relationship with the present work [28].

In addition to model-based development methodologies, **low-code tools** have also gained prevalence in the enterprise application industry, standing as a valuable resource against volatile changes in market requirements. These tools provide facilitating mechanisms, supported by a **Graphical User Interface (GUI)**, standing as a considerable alternative to traditional computer programming, allowing for their users to rapidly design, develop and deliver software. In addition, due to their emphasis on their visual development, these tools commonly do not require professional nor trained developers, hence, users of a wide specter of backgrounds can put their expertise to better use without needing **Information Technologies (IT)** related knowledge [23].

1.2 Motivation

OutSystems provides a model-driven development and delivery platform, based on allowing developers to create enterprise-grade web and mobile applications. Most of this building capability has come from Service Studio, the platform's **IDE**, capable of quickly delivering a full-fledged application in a singular place.

Nevertheless, in recent years, OutSystems has strived to develop the "Application Builders", a set of tools that lower the learning curve even more and provide an inclusive entrance to non-IT developers. These builders aim to eliminate the complexities of assembling multiple layers of an application that demand more technical knowledge and experienced developers and can be used together with Service Studio.

However, the use of builders has some drawbacks, such as the impossibility of a builder to edit an **app** created by other builders, damaging interoperability. In addition, an application that is first created with a builder, and then changed in Service Studio, can no longer be edited with a builder, which hinder the collaboration between business users, professional developers, and other **personas** such as designers, front-end specialists, etc.

Consequently, these shortcomings lay the foundation for the present dissertation, whose fundamental objective resided on the research and creation of a prototype capable of broadening the reach of the OutSystems **Low-Code** Platform along with delivering a continuous collaborative development experience to all their users.

1.3 Objectives

As previously mentioned, the present thesis set as its paramount objective the enabling of different personas, both business and tech-oriented, to collaborate on the development of an enterprise-level application, employing the entire set of Low-Code tools provided

by the OutSystems Platform. From a more detailed standpoint, this dissertation focused in the research and development of the necessary model transformations envisioned to support the development across multiple builders in a continuous and collaborative fashion. To attain this objective, the following steps were taken:

- Research and comprehension of the life cycle regarding the software development artifacts (models, meta-models, configurations, etc.) associated with Service Studio and the builders;
- Study of the current product functionality and current collaborative use between the OutSystems Platform tools;
- Assess of the desired behavior with stakeholders, as a way of defining the use cases to be supported;
- Analysis concerning possible infrastructural changes regarding the current collaboration between tools, in regard to application generation;
- Establishment of a feasible strategy detailing how can the builders perform application development and generation while adopting an editor related mode of operation;
- Development and implementation of a working prototype, capable of propagating changes made in a builder to the Service Studio and vice versa;
- Validation of the resulting prototype with thorough testing in a simulated collaborative environment;

1.4 Expected Contributions

This dissertation has a significant potential for expanding the reach of the OutSystems product line, particularly in the Low-Code development environment. Therefore, the main objective of this work lied upon in the successful development of a working prototype instantiating a bidirectional flow of interaction between the builders and Service Studio, allowing for different users to edit the same application, using the OutSystems development environment in a continuous and collaborative manner.

Moreover, from an academic standpoint, this dissertation's approach towards handling interrelated models aims to provide a useful contribution to [Model Driven Engineering](#) and model transformations, therefore aspiring to advance the state-of-the-art regarding these paradigms.

1.5 Outline

The structure of this document is organized as follows:

- **Chapter 2. Background:**

This chapter addresses relevant concepts that lay the foundations for the present thesis and establish the framework associated with the discussed problem. Thereby, it introduces key aspects related to [Model Driven Engineering](#) as well as model transformations.

- **Chapter 3. OutSystems Platform:**

This chapter presents an overview of the distinct elements that comprise the OutSystems Platform providing further emphasis on the components dedicated to the application development, namely the builders.

- **Chapter 4. Model manipulation:**

This chapter provides a detailed view of the underlying representation of the application models and respective meta-models, describing how these evolve in the OutSystems development platform and respective infrastructure.

- **Chapter 5. Related Work:**

This chapter illustrates the vast literature research that was made with the purpose of contextualizing and justifying the defined approach towards this problem.

- **Chapter 6. Design and Implementation:**

This chapter elaborates on the delineated strategy deemed as more viable and better suited to solve the issue at hand. It describes the reasoning behind the design decisions and depicts the implementation process in thorough detail.

- **Chapter 7. Results and Evaluation:**

This chapter presents the features of the developed prototype and compares them with the current behavior of the Experience Builder. In addition, this chapter also elaborates on the performed evaluation procedures and the prototype's limitations.

- **Chapter 8. Conclusion and Future Work:**

This chapter comprises the final remarks concerning the results obtained in this dissertation, along with several points that could be addressed in the future.

BACKGROUND

2.1 Model Driven Engineering

In 2000, the [Object Management Group \(OMG\)](#), published a white paper about [Model Driven Architecture \(MDA\)](#) [13] which meant to define the standards and terminology for a new approach in the software development process, built upon the notion of assigning greater importance to object models. This led to the development of paradigms like [Model Driven Development \(MDD\)](#), [Model Based Engineering \(MBE\)](#), [Model Driven Software Development \(MDS\)](#), that in the last years have emerged as effective software development approaches in both academia and industry contexts. Despite each one bringing specific assets to the table, the mentioned approaches can all be clustered in the [Model Driven Engineering \(MDE\)](#) paradigm whose principles distinguish itself from more classical development methodologies intrinsically correlated with manually written and maintained code.

Although these approaches have been quite the buzzwords in recent times, they utilize a very endearing technique in computer science and engineering in general: abstraction. Nonetheless, these development techniques go beyond the traditional objectives regarding complexity reduction, generally achieved through the expressing of domain-specific concepts in a concise fashion. Thereby, the model assumes a pivotal role in the context of [Model Driven Engineering](#), overriding its classical use in design and documentation phases, and enabling a more coherent vision of the system amongst developers and stakeholders, therefore facilitating their communication and overall work. The concept of model transformations has direct application in the above-mentioned context, namely the realm of model-to-model transformations whose scope is associated with the representation of different views regarding the same object logic, consequently optimizing communication and comprehension. Whilst model-to-code transformations are relevant to the definition and management of automated software development since they enable code generation from model resources.

[OMG](#) currently manages the [Unified Modeling Language \(UML\)](#), an [ISO](#) standard [6] that help “*specify, visualize, and document models of software systems, including their*

structure and design, in a way that meets all of these requirements”[12], by providing a set of tools that focuses on standardizing the way software’s structure, behavior, and interaction is expressed. The following key concepts of **Model Driven Engineering (MDE)** abide the **UML** standard.

2.1.1 Model

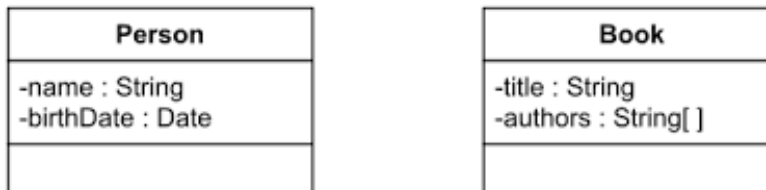


Figure 2.1: UML Java Class Models

The notion of a model has a cross-sectional meaning and relevance in various fields of knowledge, such as Engineering, Finance, Economics, Physics, etc. The modeling process consistently aims to develop a simplified perspective of what in reality is described in greater detail and complexity.

In the branch of Computer Science, and more particularly in **MDE**, the model is considered to be a “*first-class artifact*” [24] and the cornerstone of this software development paradigm, corresponding to an abstraction that aims to summarize the main characteristics regarding the structure, behavior and overall definition of a system under study. Furthermore, a model allows predictions and inferences to be made with a considerable degree of correctness and fidelity, empowering engineers and developers to resolve issues without directly considering the represented system.

Selic [25] claims that the struggle in accepting MDE as a viable and useful software production technique in the past was in great measure the result of adopting models lacking the following five characteristics:

- **Abstraction:** a model must describe the “essence” of the reality it represents;
- **Understandability:** it must clearly express its meaning, reducing the “intellectual effort” used to comprehend the model in question;
- **Accuracy:** it must provide a faithful representation of the system which it intends to model;
- **Predictiveness:** it must be capable of allowing correct predictions to be made, without taking into account the modeled system;
- **Cheapness:** it must be easier to produce and analyze than the modeled system.

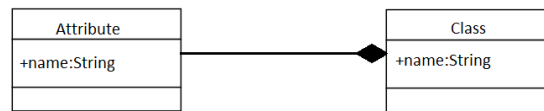


Figure 2.2: Meta-model

2.1.2 Metamodel

Another key concept in this methodology resides in metamodels, which can be considered as the “models of models”, defining the latter’s syntax, schemes, constraints, etc. In other words, a metamodel consists of a set of specifications that provide a higher level of abstraction and describe the domain of resources that can be used to construct models.

By defining the rules and constructs needed for creating models, metamodels are considered models of a modeling language and an interpretation of a given metamodel resolves to a direct mapping between the metamodel’s elements and the modeling language’s elements. Through the analysis of the metamodel’s specifications, the validity of the respective model can be assessed upon it following the determined "rules".

2.1.3 Meta-metamodel

As one can deduct from this rationale, the mentioned modeling language must also be described in some terms, hence the notion of meta-metamodels.

Similarly, one may also infer that a higher-level of abstraction should then be required to define those same meta-metamodels, and this type of multi-level architecture would never come to an end, with the higher layers always obliging specifications of even higher layers, and so on. In order to overcome this problem, it is generally employed, within a certain level of the hierarchy, a language that describes itself in its own language in a process referred to as *reflexive metamodel/meta-metamodel* (depending on the level of its use). This technique is frequently used in Computer Science, p.e. with *Bootstrapping* [27], where a compiler of a certain language (C, Lisp, etc.), is written in the source programming language it was designed to compile.

2.1.4 Meta-Object Facility

[Object Management Group](#) proposes a hierarchy layered architecture, that summarizes the multiple levels of abstraction, as well as the interactions between them, and addresses this higher abstraction problem with a reflexive meta-metamodel in the top layer.

This architecture is named [Meta-Object Facility](#)(MOF) [14], and at the top layer, designated as M3, lies the meta-metamodel, defining the language used by the [MOF](#) to build the metamodels. As mentioned this layer is instantiated from its metamodel, in the previously mentioned, reflexive fashion. Further down, in the M2 layer, we have the metamodels section, instantiated by the [MOF](#) defined above. An example of an element

belonging to this section is the [UML](#). Directly below, M1 defines the model layer, composed of instantiations of the layer up above. And finally, in the lowest layer, M0, we have the real element that may be modeled.

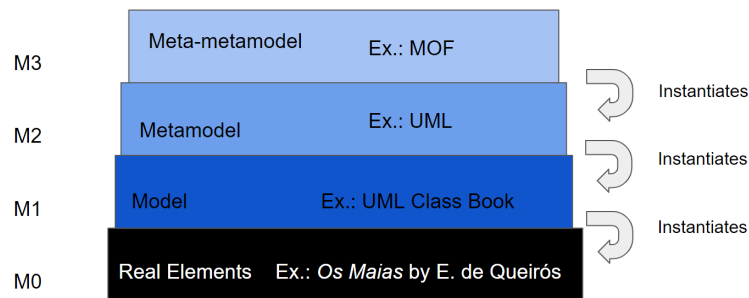


Figure 2.3: MOF hierarchy

2.2 Domain Specific Modeling Languages

As a natural consequence of having object models as the central piece in the [MDE](#) approach, comes the need of manipulating those same elements in a formal and context-specific manner, hence the existence of [Domain Specific Modeling Language \(DSML\)](#)s. These languages provide a closer interaction between the developers and the concepts of the issue at hand. This is accomplished by reducing the burden associated with the technical details of the underlying implementation.

These languages allow the formalization of the application elements, structure, its behavior, and requirements, and can be characterized by two main aspects: it is *abstract syntax*, usually supported by the metamodel component, defines language concepts, rules on how they relate to each other and determine relation constraints between them. While the *concrete syntax* maps those same concepts into their programmable representation, textual or visual. In the case of the OutSystems [DSML](#), the concrete aspect of the [language](#) refers to visual representations, like *Actions*, *Entities*, *Widgets*, etc.

2.3 Model Transformations

The mentioned approach, as it should be clear by now, calls for a recurrent and continuous use of different models. These, in turn, can either provide a more horizontal view, focusing on the description of different aspects and elements of the system, or they can have a more vertically-oriented approach, representing the same component(s) through different abstraction levels. And from this collective and interconnected use of models arises the importance of defining and automating processes capable of converting a model into a different one. Such process or set of processes constitute a *model transformation*,

and require the existence of one or more *Source Model(s)*(the input of the transformation), the corresponding existence of one or more *Target Model(s)* (the outcome of the process) and finally a set of *transformation rules*.

2.3.1 Kinds of Model Transformations

There are multiple kinds of model transformations, whose classification can be made according to a great variety of features [11], but in the scope of this work, it is important to distinguish between *unidirectional transformations* from *bidirectional transformations*. The former ones, as the name suggests, comprehends transformations with a consistent type of input, the source model, and a consistent type of output, the target model, which is not generally manipulated by a human. No other mode of operation is supported, and all the processing occurs in one direction only. These transformations are frequent in the domain of compilers where a model supported in a high-level programming language (like Java or C#) is transformed into a different model described in a lower-level language (like assembly or machine code). The bidirectional variant comes off as more relevant to the domain of this work, and in opposition to the previous kind of transformation, it accepts both the source model and the target model as its input. A classical example of the use of this type of transformation is the [Model Driven Architecture](#) problem that emerges from the transformation from *Platform Independent Models (PIM)* to *Platform Specific Models (PSM)*. In this scenario, we have a model that specifies structure, behavior, and functionality generically, without any direct association with a particular implementation, which is modified in order to comply with the constraints and implementation details of a specific platform, or vice-versa[2].

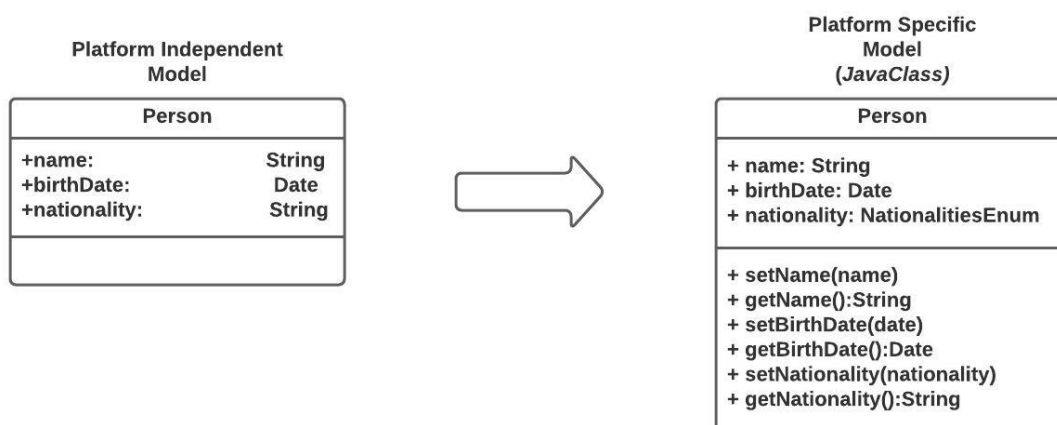


Figure 2.4: PIM to PSM model transformation

Another relevant way of grouping model transformations relates to the differences between the languages defining the models. Thereby, when all models comprising a transformation are represented in the same language, we are talking about *endogenous*

transformations. In the scenario where the source and the target models differ in their language of representation, it is a question of *exogenous transformations*. On the one hand, the endogenous type refers to transformations linked with optimization and refactoring processes, where an artifact is improved or simplified in order to acquire better understandability or reusability, etc. without giving rise to observable behavior changes, for instance. On the other hand, exogenous transformations comprise software migrations namely, where artifacts are defined in a different language even though no changes are made to the original level of provided abstraction.

2.3.1.1 Bidirectional Model Transformations

Bidirectional model transformations are enclosed in the bigger picture of bidirectional transformations (*bx*) [3], and sometimes these expressions are used interchangeably owing to the fact that both refer to mechanisms for *consistency maintenance* across two (or more) related sources of information, although the former concept perceives those sources as models. Due to its bidirectional character, the terms “source” and “target” model are less adequate in this paradigm, but generally speaking, the first model to be generated and the one usually more prone to changes, is designated as the source model. These transformations have gained momentum in multiple realms of computer science and engineering, being addressed in several fields such as programming languages, document and database engineering and naturally in [MDSD](#) to provide solutions for long-established problems such as:

- the mentioned **PIM-to-PSM** [2];
- the **database view** problem where changes made by a user in a view must be propagated to the actual database [8];

A relevant and distinctive characteristic of this category of transformations is its emphasis on consistency maintenance and change propagation concerns. This emerges from common scenarios in which the target model is very likely to be changed by some developer, hence the source model can potentially become outdated forcing restore actions to take place to guarantee both sides have an updated view of their models.

A *bx* between two sources/models - let's name them S and T - consists of a pair of unidirectional transformations one from S to T and another from T back to S. The process of going from S to T is called a *forward transformation*, while naturally, the reverse process from T to S goes by the name of *backward transformation*. These transformations can be further categorized according to the established consistency relation concerning the pair of models involved. We call a transformation *bijection* in the case the consistency relation corresponds to a bijection, that is to say, for any model there exists a unique corresponding model satisfying the consistency specifications. In particular, for two sets of models with different cardinalities, the referred bijection is impossible [28].

Considering a transformation that defines two functions between two models: $f : S \rightarrow T$, and the respective inverse function $f^{-1} : T \rightarrow S$. The consistency relation is bijective, therefore the bidirectional transformation is bijective if and only if, both functions are *surjective*, that is each model element in S can be part of the transformation to T and vice-versa. In addition, both functions are required to be *injective*, i.e. every element of a model must correspond to at most one element of the other model and conversely.

2.3.2 Lenses

The naive way of developing a bidirectional transformation is to build two unidirectional transformations and potentially define an explicit consistency relation between them. This naturally requires added technical effort concerning the definition of the transformations which can be expensive and error-prone. Furthermore, this naive approach requires continued maintenance since any change in a data format implies a redefinition of both transformations and a new consistency proof. [22].

To address this issue Foster et al. in [5] defined the concept of lenses, which can be considered one of the most prominent approaches in the realm of bidirectional transformations. Developed to tackle the *view-update problem*, a lens defines a forward transformation as a projection of concrete models into abstract views and a backward transformation as the reverse process associated with the translation of an abstract view into an update over concrete models. The first component of this approach consists of the function $get : C \rightarrow A$ responsible for abstracting details from the concrete model that come as unnecessary in the domain in question. Secondly, in order to re-establish the abstracted information, the lense requires the function $put : A \times C \rightarrow C$ that considers the original concrete instance in the interest of restoring information no longer present in the view. When this is not possible, a default concrete model can be restored by applying the function $create : A \rightarrow C$ to the view.

OUTSYSTEMS PLATFORM

OutSystems provides a model-driven development and delivery platform, allowing developers to quickly and easily build and deploy enterprise-grade web and mobile applications, that can be run in the cloud, on company premises, and in hybrid environments. This platform concentrates in a single place multiple tools and resources, that provide the user a thorough design and customization capability, in all the major aspects of their application, such as business logic and processes, data modeling, user interface, experience design, [DevOps](#), among others [19].

A huge part of this fast and uncomplicated application development has to do with the integrated visual [Domain-Specific Language \(DSL\)](#) the OutSystems Platform offers to its users. Its main goal, like other [DSLs](#), lies in providing an added distance, by means of abstraction, between the developers and the details and challenges that occur from building and changing an application. In order to achieve such a feat, users interact directly with an environment more aligned with the specific problem at hand. With this added simplicity, enterprises reduce substantially development and deployment time, as well as overall costs in the production of software.

3.1 Architecture

Several environments, tools, components, and services compose the OutSystems Platform (Figure 3.1), an infrastructure responsible for gathering in a single place all the required resources better fitted to address the entire lifecycle of an application, not only covering the development phase but also addressing concerns related to deployment, management, and monitoring. This section will provide a brief overview of the main components comprised in the platform's architecture.

3.1.1 Service Studio

Service Studio [20] is the platform's [IDE](#) (Figure 3.2), and therefore stands as a crucial component in the OutSystems developing environment, combining a visual low-code approach, supported through a [Domain-Specific Language](#), with the capability of creating

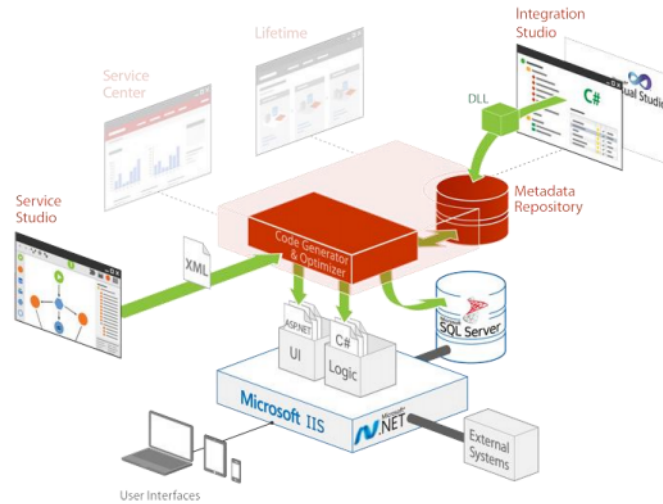


Figure 3.1: OutSystems Platform

all parts of a fully-fledged, enterprise-level application, in a single place. This development capacity correlates with the creation and customization of *eSpaces*, application-specific modules where processes, business logic, data models, and views can be defined, as well as custom [User Interface \(UI\)](#) and [User Experience \(UX\)](#).

Using Service Studio, developers can visually build all the necessary layers of an application and publish it to the Platform Server (Figure 3.1), which in turn will be responsible for generating, building, packaging, and finally deploying the created software to a standard application web server and database, allowing for [end-users](#) to access their designed applications using their devices.

3.1.2 Platform Server

This server is the core component in the OutSystems Platform, comprised of several servers dedicated to the generation, build, deployment, and packaging of web and mobile applications. Developers can connect to this component through Service Studio, where they can create and publish their applications, prompting the Platform Server to compile and generate optimized code to a standard application server [19].

3.1.3 Integration Studio

Integration Studio stands as a desktop tool, where developers can create and manage their used extensions, i.e. custom components to integrate with pre-existing systems, databases, or even code. These extensions can then be consumed in Service Studio, as [add-ons](#) [17].

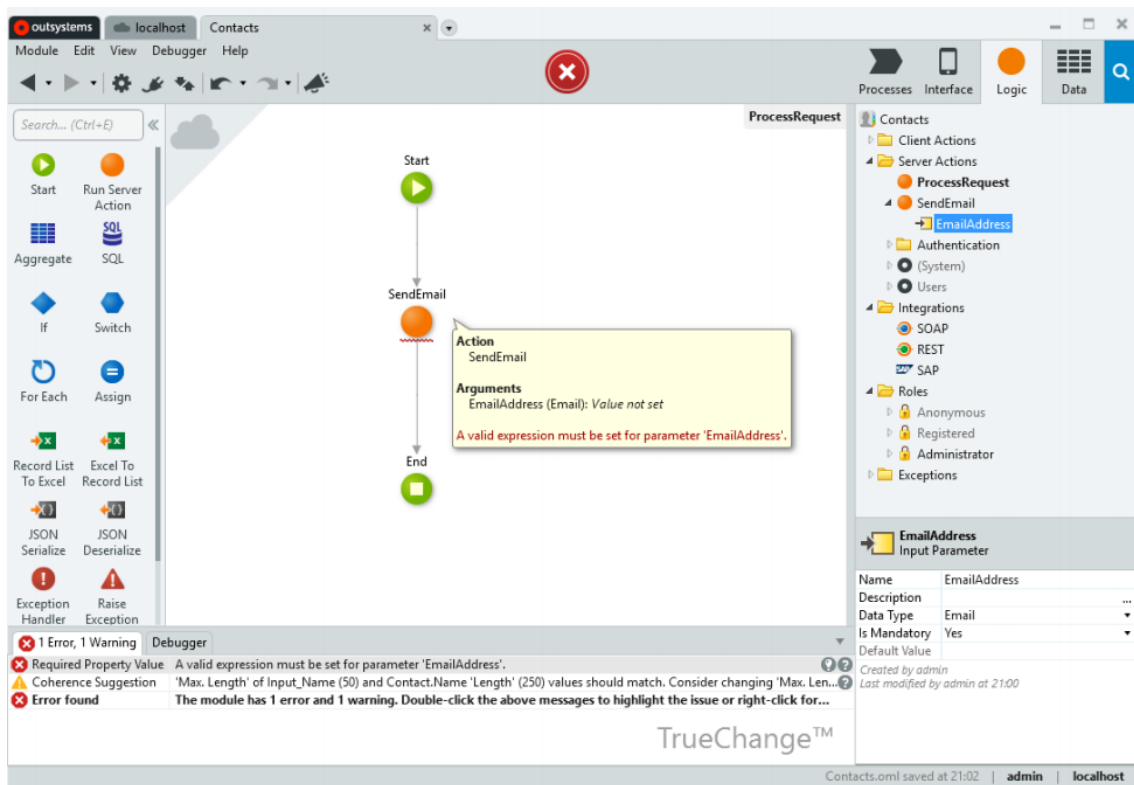


Figure 3.2: Service Studio

3.2 Builders

In the last few years, OutSystems has developed and released a set of visual development tools based on the [Software-as-a-Service \(SaaS\)](#) model, named builders [18]. These tools optimized for particular aspects of the development cycle and for team collaboration, by bridging the gap between domain experts with non-IT related backgrounds and software engineers. The builders allow for users to generate complete software solutions, with a small number of interactions, therefore reducing complexities correlated with the assembly of multiple-application layers, that frequently require the help of professional developers.

With these tools, a domain expert can focus on their specific area of knowledge, and rapidly create a prototype or even a completely functional application that can be rapidly deployed and used, or furtherly developed by a user in greater technical detail using Service Studio.

Currently, there are two builders available to the public: the Experience Builder providing greater focus towards the [User Experience \(UX\)](#) development, and the Workflow Builder, associated with the design of task management and automation applications. The Integration Builder which has not been released concentrates on enabling the extension of existing systems, integrating an [OutSystems Application \(Package\) \(OAP\)](#) with external data resources.

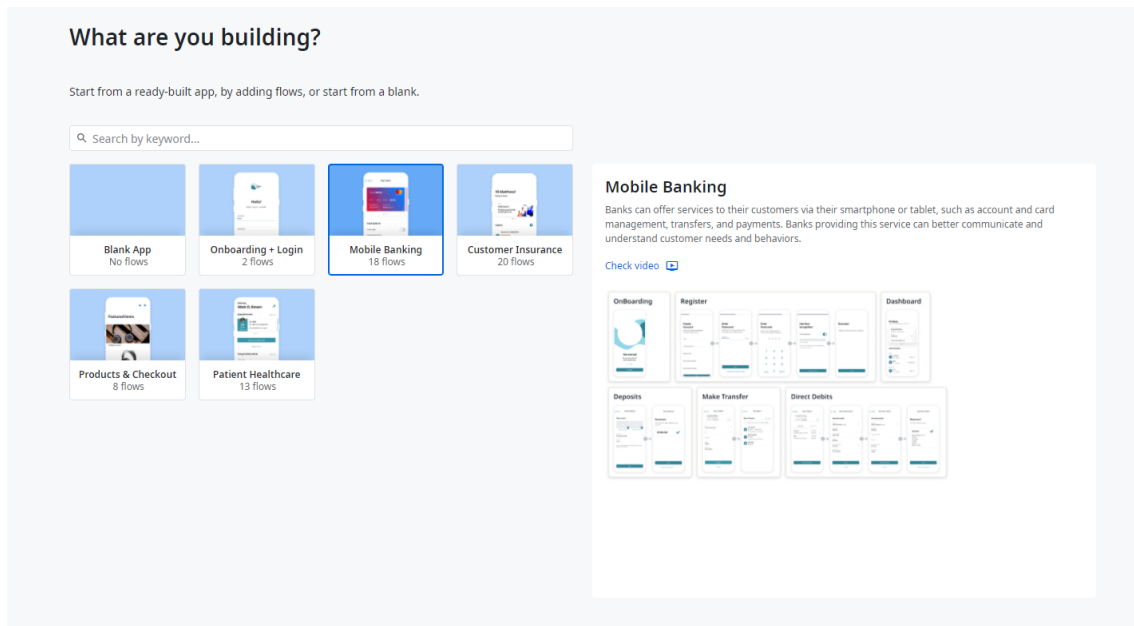


Figure 3.3: Experience Builder - Template Screen

3.2.1 Experience Builder

The Experience Builder [15] targets a faster development of B2C mobile applications while striving to provide a great User Experience (UX), without burdening the developer with technical comprehension of the back-end code. Developers can build a fully working prototype of a mobile app, selecting from a wide array of possible user journeys and pre-built application templates.

In addition to enabling mobile app basic customization, by changing the icon and selecting a primary color, this tool allows users to start developing their application prototype by choosing from several pre-built application templates or starting from a blank canvas (figure 3.3). Each template is composed of a specific set of predetermined flows designed to address common user interface patterns such as *login* or *onboarding* processes. Moreover, Experience Builder does also provide built-in templates associated with industry-specific patterns, such as *Mobile Banking* or *Customer Insurance*.

Using this builder, developers can assemble their applications with custom journeys through the use of flows from different application templates. This means developers are not bound to flows (figure 3.4) from a specific template. For instance, they can combine an *Authentication* flow with a flow included in the *Mobile Banking* template. Furthermore, the Experience Builder also provides customization options for the application menu, as developers can change the button icons and link them to new or existing flows (figure 3.5).

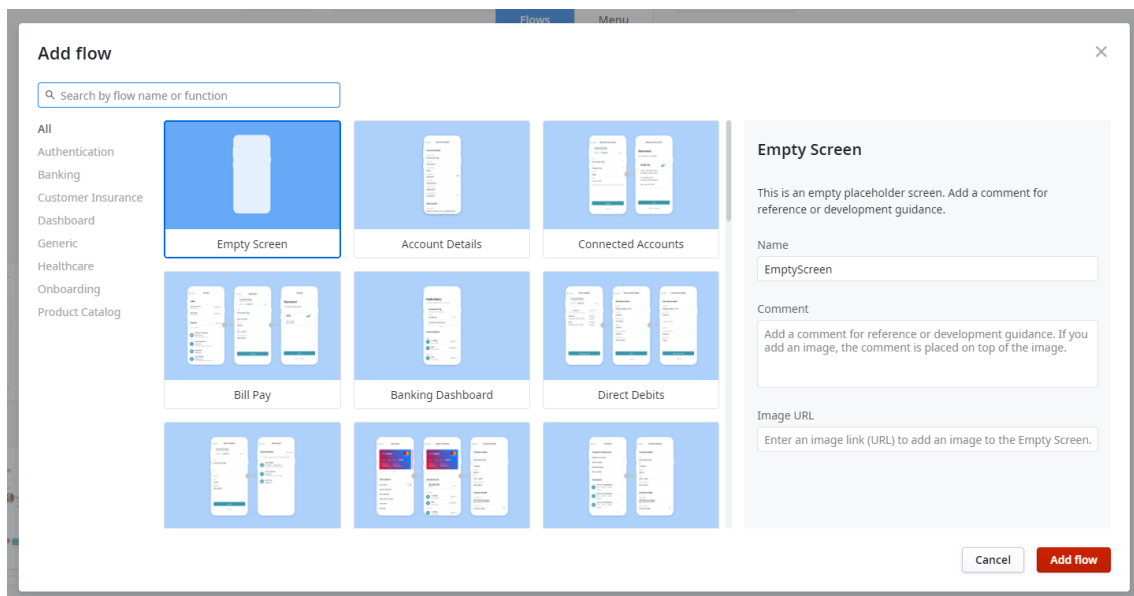
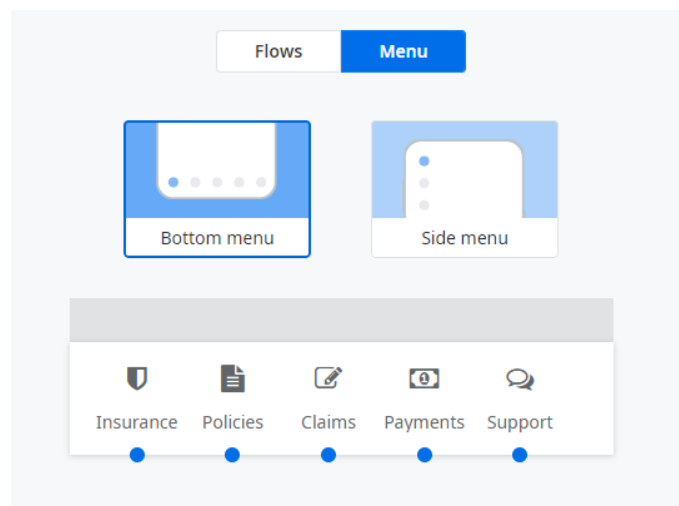
Figure 3.4: Experience Builder - *Add Flow* Screen

Figure 3.5: Experience Builder - Menu Customization

3.2.2 Workflow Builder

The second SaaS builder is the Workflow Builder [21]. This tool is coupled with business logic and processes, aiming to allow multidisciplinary teams to create their workflows and sequences of activities, leading to improvements in productivity and reducing IT load, as a result. Equipped with this builder, developers can define, organize and manage business processes automatizing frequent and predictable activities.

Similar to the Experience Builder, developers can start to build their applications from scratch or by selecting from one of three pre-built app templates available (figure 3.6) - *Project Request*, *Approval Request*, and *Issue Report*. *Project Request* relates to the automated processing of incoming projects, or requirements, requested within an organization, while

the *Approval Request* template handles requests concerned with granting resource access. Finally, the *Issue Report* case affiliates with the management of organization incidents, providing ticket creation and handling mechanisms.

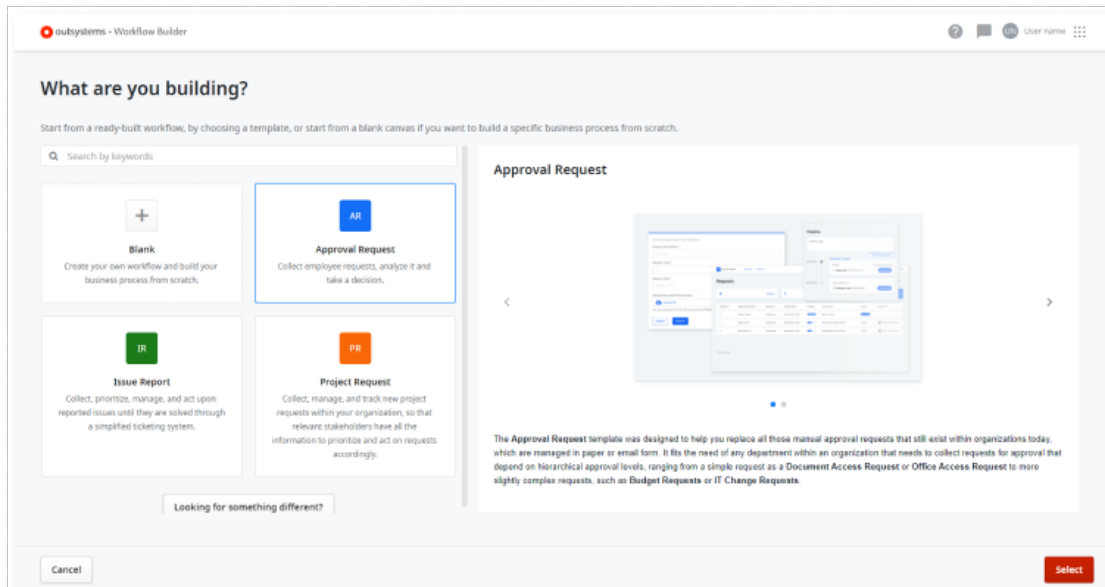


Figure 3.6: Workflow Builder - Template Screen

Every management case is associated with a distinct business process. These processes, independently of the selected management case, behave similarly and can be furtherly customized by setting up forms with which the *end-users* will interact (figure 3.7) and defining requirements that may trigger a particular action (figure 3.8). These actions generally take the shape of automated emails sent to business entities such as a finance group or a manager, whose answer might be evaluated and continue the defined business process.

3.2.3 Integration Builder

The final builder is the Integration Builder [16]. This *SaaS* stands as a tool capable of creating integrations that connect an OutSystems application to external enterprise systems or records. Those integrations correspond to generated *OAPs* that support interaction with the external system as configured by the user. The main purpose of this builder is to enhance applications with data originated from external providers (figure 3.9) related with from *Enterprise Resource Planning* (ERP) and *Customer Relationship Management* (CRM) software like SAP, and Salesforce.

In the context of the Integration Builder, a connection to data sources from an external system (like the ones above-mentioned) is referred to as an *Integration*. Using this tool, developers start by selecting the service provider that's the data source of their desired integration (e.g. SAP). Consequently, developers can then determine which data source

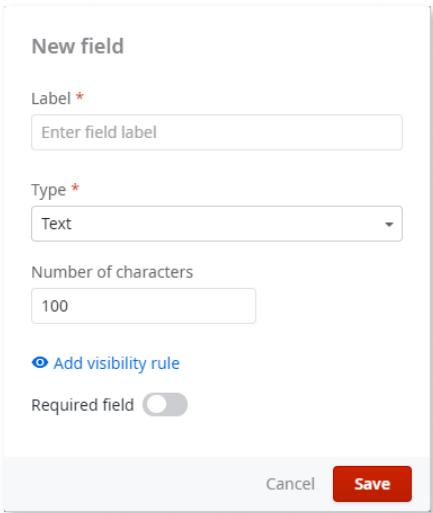


Figure 3.7: Workflow Builder - Form Customization

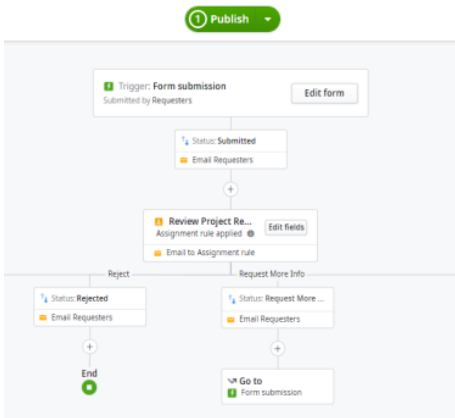


Figure 3.8: Workflow Builder - Business Process Customization

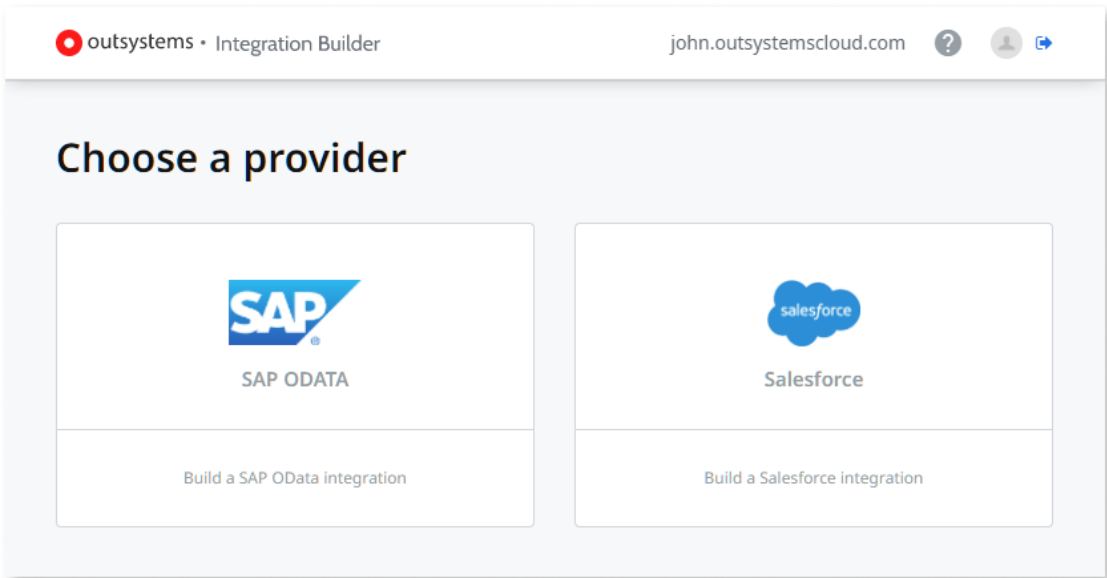
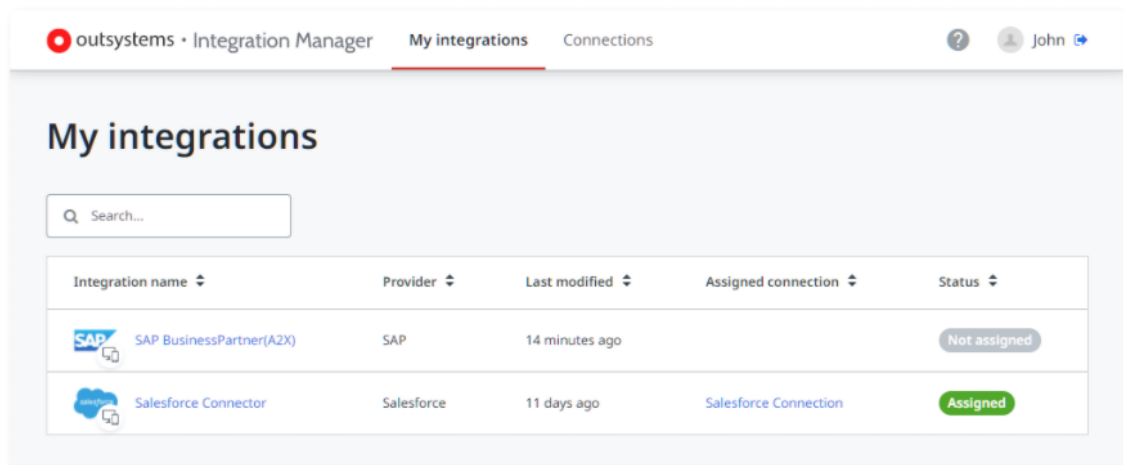


Figure 3.9: Integration Builder - Choose Provider Screen

Figure 3.10: Integration Builder - *My Integration* Screen

objects they will interact with, using OutSystems applications. This interaction will be made possible using **CRUD** and other operations created for each source object selected (figure 3.10).

After publishing this integration, developers can use it in Service Studio through a set of **server actions** that will expose the previously created **CRUD** operations, furthermore, they can also filter, define the sort order, and paginate results from invoking such operations.

3.2.4 Example of the current development shortcomings

The following example illustrates the current builders shortcomings in a scenario where two users work cooperatively on the construction of the same app. The example will motivate and later on be used to describe the developed algorithm:

Ann and Bob are two developers working on a simple application. Ann is using Experience Builder, while Bob is using Service Studio. Ann starts by defining and publishing an application with two flows, *AnimatedOnboardingOption1* and the *LoginAndPasscode*. The publishing process ends with the generation of an app ready to be used. However, Bob is not completely satisfied with the name Ann chose for a screen in the second flow. He opens the application in Service Studio and changes the name from *SetFaceID* to *ActivateFaceRecognition*. In addition, Bob defines the database entities and their corresponding attributes required to store the users of the app, something that Ann couldn't have done in Experience Builder. He proceeds to setup a *Users* entity and several basic entity attributes and publishes the new version of the application. Unaware of these changes, Ann returns to Experience Builder where she adds a *GoogleLogin* flow, changes the *AnimatedOnboardingOption1* flow name to *BasicOnboarding* and re-publishes the app. As described, this process will fully regenerate the application, consequently discarding all the changes made by Bob.

MODEL MANIPULATION

This chapter concentrates on the current management and manipulation of an application's models and meta-models, in both the **builders'** environment, as well as in **Service Studio**. It provides a detailed view of the underlying representation of the application artifacts, describing how these evolve in the OutSystems development platform and respective infrastructure.

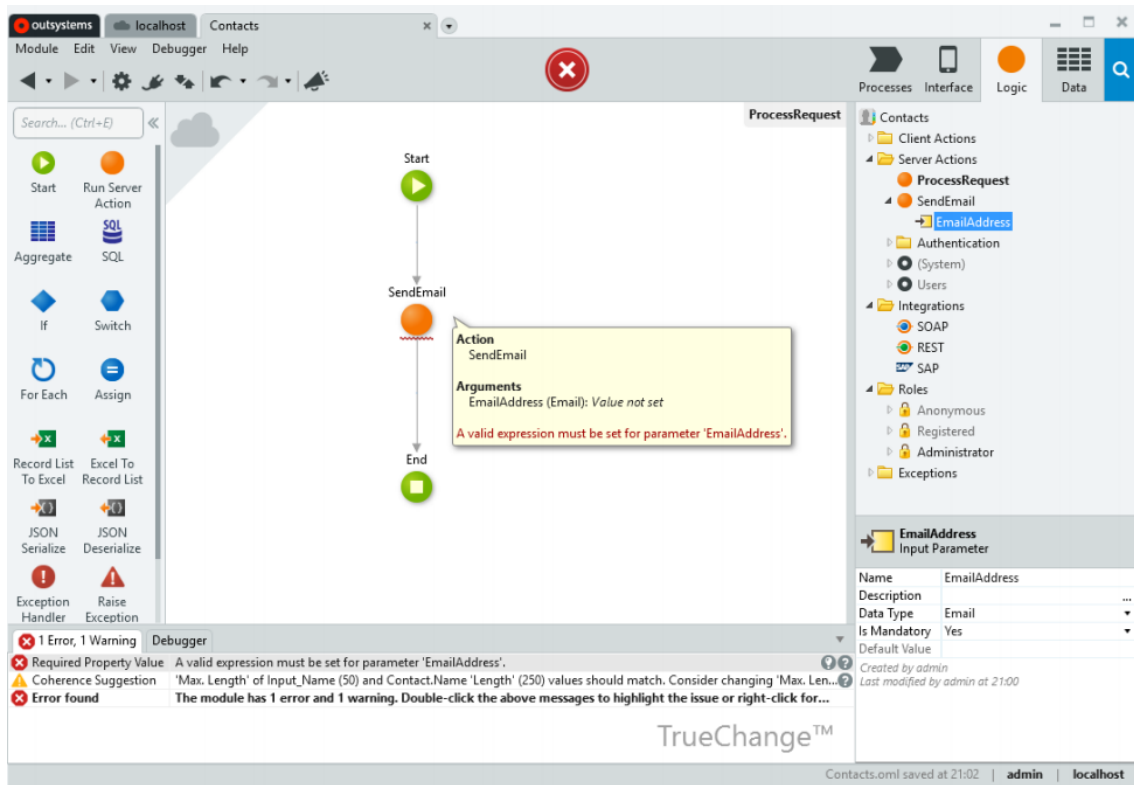
4.1 OutSystems Language

With the aid of **Service Studio**, developers can swiftly create and publish full-fledged web and mobile applications. Its promptness and ease of use are the by-products of the OutSystems visual language, whose internal representation is defined by a set of models persisted and transported as binary XML files. For the sake of simplification, we only consider a subset of the OutSystems Language consisting of Actions, which are used to design business logic, assuming a very similar structure to methods in textual languages such as C# or Java. The OutSystems language supports client actions, executed in the client's device, and server actions, which are run on the server-side. Figure 4.1 illustrates the **Service Studio**'s user interface.

Actions are defined as direct graphs composed of different types of nodes, depicted on the left toolbox. The following example will consider the action *ProcessRequest*, comprised of three nodes: *Start*, *SendEmail*, and *End*. *Start* and *End* have the expected role, indicating the beginning and end of the action. *SendEmail* corresponds to a *Run Server Action* node, utilized to propagate a call to another action. As depicted on the right side of figure 4.1, this action requires a mandatory input parameter, named *EmailAddress*, of type *Email* [7, 9].

4.2 OutSystems Meta-model

The language meta-model is persisted in an XML format supported using an in-house developed schema. In the listing 1, an instance of the meta-model is depicted regarding

Figure 4.1: Service Studio Interface - Action *ProcessRequest*

the example's subset of the OutSystems Language. Several classes are declared, such as the *Espace*, corresponding to an application module, and *Action*. The latter, as it can be seen, features a set of *InputParameter* and a set of *ActionNode* (lines 12 and 13, respectively). Furthermore, the class *Execute*, representing the *Run Server Action* node, discussed above, is comprised of a *Property* and a *Child*. The *Property* corresponds to an *Action*, used to store the action which is being executed, while the *Child* is a collection of *Argument*, i.e. particular parameters and their respective values.

It is relevant to note, that the classes declared in the meta-model can be used as the type for properties and collections in other classes [7, 9].

Figure 4.3, depicts the above mentioned meta-model in UML format.

4.3 OutSystems Model

The meta-model XML is used to generate C# classes used in *Service Studio* and by the compiler. Since these generated model classes conform with the meta-model description, one can define an entirely distinct language by its altering the corresponding meta-model.

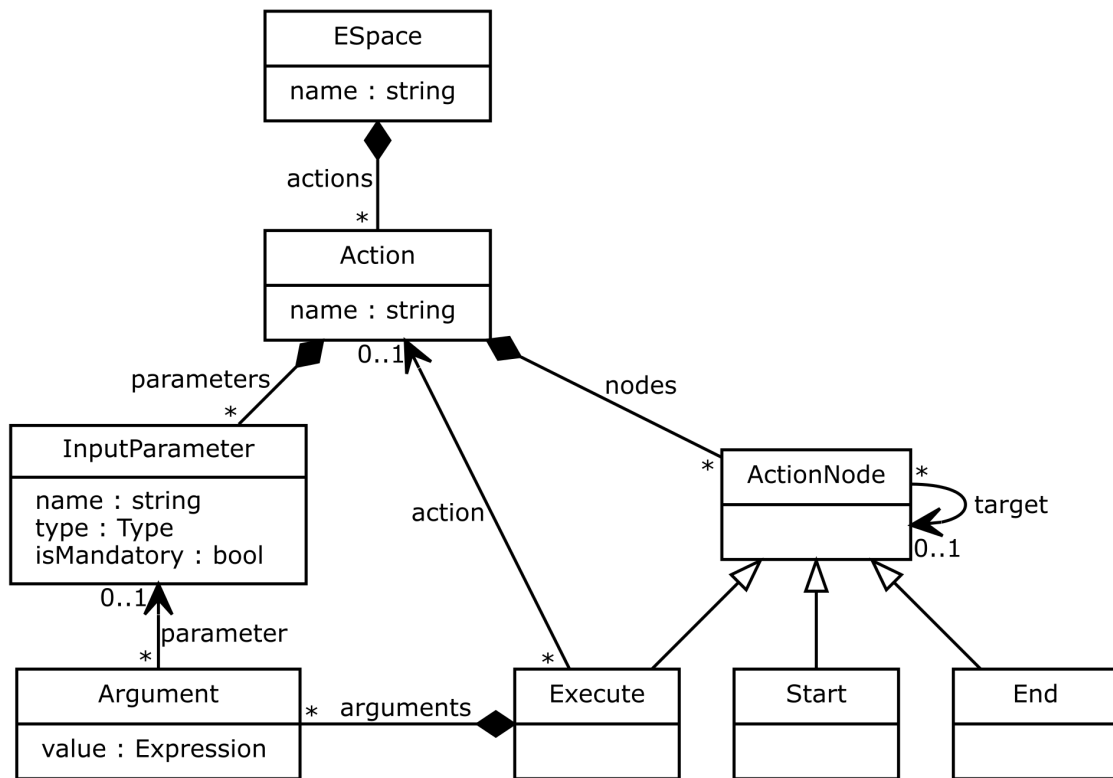
Furthermore, as previously mentioned, the OutSystems model is represented in memory as an object graph persisted as a binary XML file, hence, it should not be surprising that this file corresponds to the serialization of the discussed model classes. Having said that, the model regarding the meta-model illustrated in listing 1 is illustrated in listing 2

```

1 <MetaModel
2   xmlns:xsi="http://www.w3.org/2001/ XMLSchema-instance"
3   xmlns:xsd="http://www.w3.org/2001/XMLSchema"
4   xsi:schemaLocation="http://www.outsystems.com MetaModel.xsd">
5   <Class name="ESpace">
6     <Property name="Name" type="Text"/>
7     <Child name="Actions" type="Action"/>
8   </Class>
9   <Class name="Action">
10    <Property name="Name" type="Text"/>
11    <Child name="InputParameters" type="InputParameter"/>
12    <Child name="Nodes" type="ActionNode"/>
13  </Class>
14  <Class name="InputParameter">
15    <Property name="Name" type="Text"/>
16    <Property name="Type" type="Type"/>
17    <Property name="IsMandatory" type="Bool"/>
18  </Class>
19  <Class name="ActionNode">
20    <Property name="Target" type="ActionNode"/>
21  </Class>
22  <Class name="Start" base="ActionNode"/>
23  <Class name="End" base="ActionNode"/>
24  <Class name="Execute" base="ActionNode">
25    <Property name="Action" type="Action"/>
26    <Child name="Arguments" type="Argument"/>
27  </Class>
28  <Class name="Argument" verifyDependencies="Parameter.IsMandatory,
29  Parameter.Type">
30    <Property name="Parameter" type="InputParameter"/>
31    <Property name="Value" type="Expression" isOptional="true"/>
32  </Class>
33 </MetaModel>

```

Listing 1: OutSystems *Action* Meta-model

Figure 4.2: OutSystems *Action* Meta-model (UML)

and in UML format, in figure 4.3. The representation for object references consists of the target object’s type and its id, as it may be seen in line 15, where the value for the action attribute is interpreted as the action *SendEmail* in line 5, considering the referenced id [7, 9].

4.4 Builders

The **builders** allow users to generate complete software solutions, with a small number of interactions, therefore reducing complexities correlated with the assemble of multiple-application layers. As OutSystems tools, these pieces of software are supported on a strongly visual development environment, built upon the use of models.

Once again, regarding the interest of simplification, we will concentrate on the **Experience Builder**, more specifically on the core-elements manipulated by this tool - Screens, and Flows.

Figure 4.4 illustrates the **Experience Builder** interface. As can be noted, the figure depicts two connected flows *AnimatedOnboardingOption1* and *LoginAndPasscode*. Developers can incorporate new flows by interacting with the *Add Flow* button portrayed on the left side of the figure. Each flow is associated with a pre-built template and is composed of one or more screens. In this example, *AnimatedOnboardingOption1* and *LoginAndPasscode* are part of the *Authentication* template, and it may be seen, the latter flow is composed of

```

1 <Model>
2   <ESpace id="1" name="Contacts">
3     <Actions>
4       <Action id="4" name="SendEmail">
5         <InputParameters>
6           <InputParameter id="5" isMandatory="true"
7             name="EmailAddress"
8             type="Type:3" />
9         </InputParameters>
10      </Action>
11     <Action id="6" name="ProcessRequest">
12       <Nodes>
13         <Start id="7" target="Execute:8" />
14         <Execute action="Action:4" id="8" target="End:9">
15           <Arguments>
16             <Argument id="10"
17               parameter="InputParameter:5" />
18           </Arguments>
19         </Execute>
20         <End id="9" />
21       </Nodes>
22     </Action>
23   </Actions>
24 </ESpace>
25 </Model>

```

Listing 2: OutSystems Model Instance

four distinct screens: *LoginSetPasscode*, *SetPasscode*, *SetFaceID*, *LoginPasscodeOK*. Furthermore, connections can be created among flows by linking screen-specific widgets such as buttons or lists for instance, to a separate flow, therefore defining what is referred to as *ExitPoints*. Inside a particular flow, screen links are predetermined and cannot be changed. In the example below, a connection is defined between *AnimatedOnboardingOption1* and *LoginAndPasscode*, hence establishing the latter flow as the *ExitPoint* of the former.

4.5 Builders Meta-model

Contrary to the OutSystems model, the builder's meta-model is not explicitly defined nor it is uniform across the various *builders*. Even though the tools manipulate elements present in the OutSystems model, in order to better accommodate the domain-specific

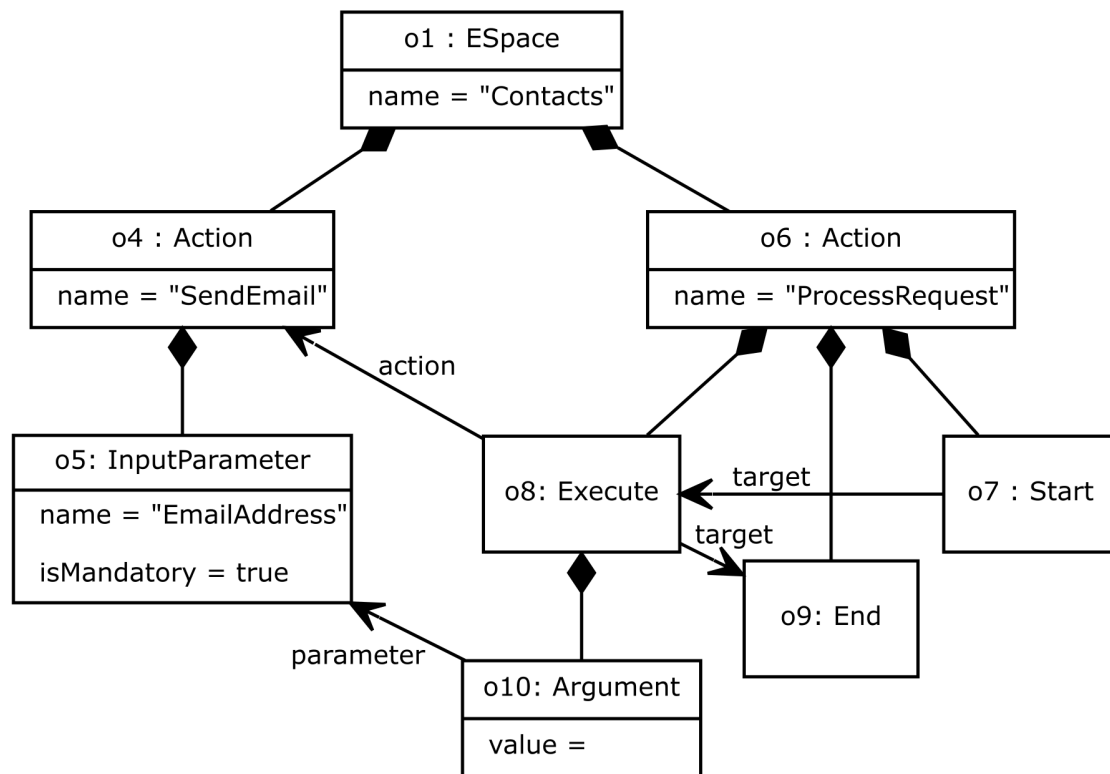


Figure 4.3: OutSystems Action Model (UML)

development features, these elements must assume a distinct representation. Hence, despite the OutSystems meta-model defining a language that overlaps with the one utilized by the [builders](#), there are discrepancies among them.

The following figure proposes a meta-model for the [Experience Builder](#), specifically addressing *Connections*, *Flows* and *Screens*.

4.6 Builders Model

While interacting with the [Experience Builder](#) web-application, the developers are indirectly manipulating the builder's model, which is persisted in a relational database representation. Once the Publish action is triggered in the builder, the application model shall be converted into a JSON file, which will be included in the application generation request sent to the OutSystems infrastructures.

Listing 3 illustrates a snippet of the mentioned JSON representation, focusing on a subset of the model's elements, namely on the previously mentioned *screens* and *connections*, while conforming with the meta-model presented in figure 4.5.

Although listing 3 illustrates an abbreviated version of a real model instance, it demonstrates the correlation within screens, flows, and connections. All the application screens are defined in the first array of the listing. In our example, the two depicted screens correspond to the first two screens composing the right flow in figure 4.4. As we can see,

```

{
  "screens": [
    {
      "name": "LoginSetPasscode",
      "description": "Log in to the app to set a passcode and
      Face ID authentication.",
      "flowType": 1,
      "screenKey": "1+vPGhJzgEu7mM016UD5sQ",
      "comment": "",
      "flowName": "LoginAndPasscode",
      "flowKey": "75ca3de1-e1c0-4ce9-8889-e069493d7a57",
      "sourceModuleKey": "63W0ewv1EEuH5jvx0qSjcQ",
      "hasMenu": true,
      "orderInFlow": 1,
      "imageUrl": ""
    },
    {
      "name": "SetPasscode",
      "description": "Set up a 4-digit passcode.",
      "flowType": 1,
      "screenKey": "emZ1wM_srECkHP3PCeVwRQ",
      "comment": "",
      "flowName": "LoginAndPasscode",
      "flowKey": "75ca3de1-e1c0-4ce9-8889-e069493d7a57",
      "sourceModuleKey": "63W0ewv1EEuH5jvx0qSjcQ",
      "hasMenu": true,
      "orderInFlow": 2,
      "imageUrl": ""
    }
  ],
  "connections": [
    {
      "linkKey": "e48kJtqTBEWSsIEq9odPNw",
      "linkType": 1,
      "originScreenKey": "1+vPGhJzgEu7mM016UD5sQ",
      "destinationScreenKey": "V0w1e3W3nkiGktcU1SxTXA",
      "Label": "Sign Up",
      "Order": 0
    }
  ]
}

```

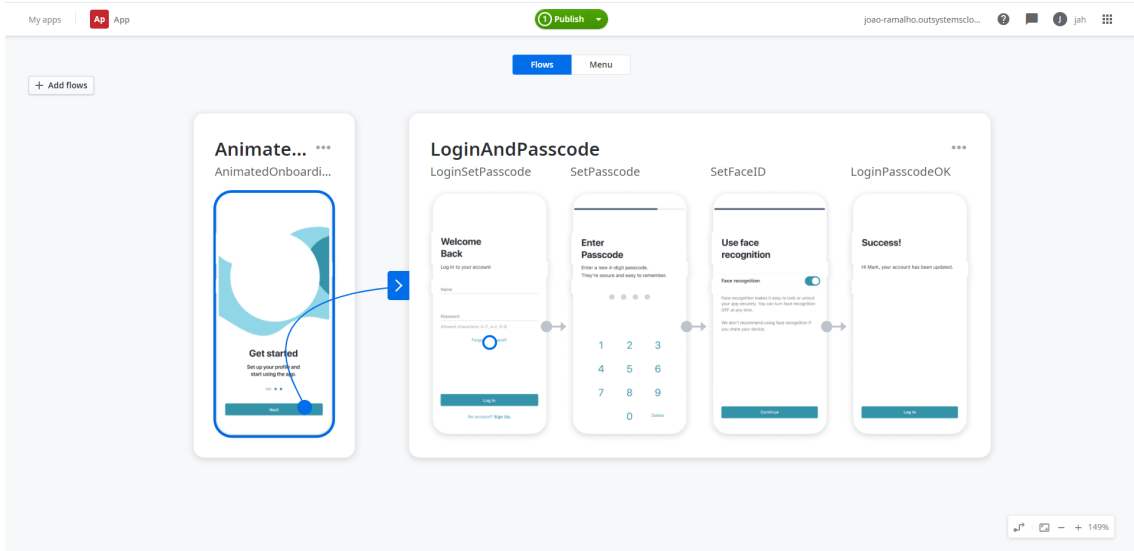


Figure 4.4: Experience Builder Interface - Screens and Flows

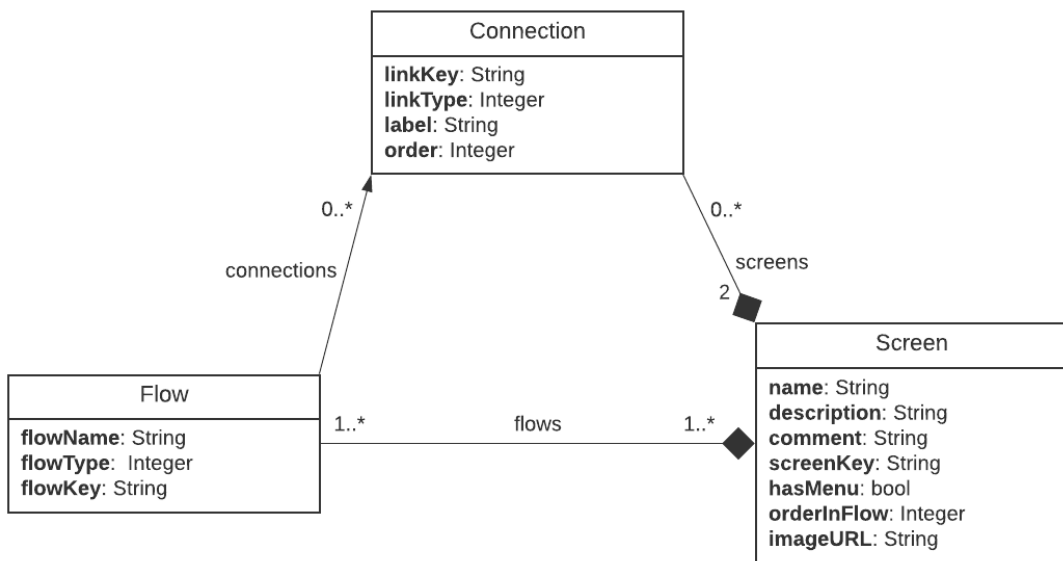


Figure 4.5: Experience Builder - Meta-model

each one has personal data such as a *name*, *description*, a *screenKey*, as well as information regarding the flow to which they belong with the *flowName* or *orderInFlow* keys.

In addition, the model must also describe every connection made between screens. The second array of the listing represents the link between the first two screens of the flow on the right, depicted in figure 4.4, as the *originScreenKey* references the *screenKey* of the *LoginSetPasscode*.

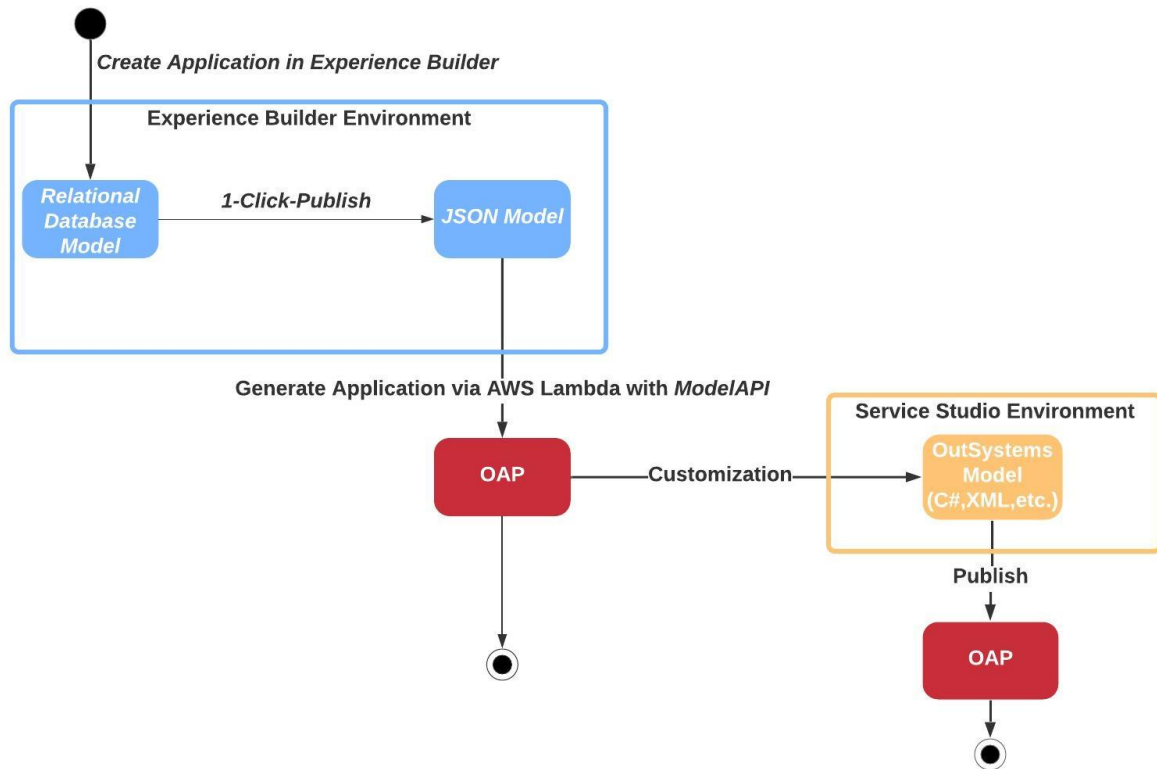


Figure 4.6: Model Flow

4.7 Model Flow

As previously stated, the publish event will invoke a new generation request sent to the OutSystems infrastructure. This request will initiate a building process that requires the upload of all the artifacts needed to produce the web or mobile application. Hence, the initial request must include the application definition, i.e. the respective application model, in this case, described in JSON format.

OutSystems leverages cloud services provided by [Amazon Web Services \(AWS\)](#) to handle these requests. The provided services render a scalable and isolated environment of paramount importance to the generation process. In this infrastructure, the application model is then utilized in [AWS](#) Lambda instances running the ModelAPI, which will be discussed later in this chapter.

The generation process culminates in the creation of a fully functioning [OAP](#), which can be further customized in the [Service Studio](#). Figure 4.6 illustrates the a state diagram regarding the model journey from a builder, to [Service Studio](#). In the [IDE](#), the application elements designed in the builder are now complying with the OutSystems meta-model, hence they can be edited on a more granular level.

```

using var modelServices =
    ↪ OutSystems.ModelAPILoader.Loader.CreateModelServices();
using var app =
    ↪ modelServices.CreateApplication(SegmentationKind.CrossDevice,
    ↪ "MyApp");

var eSpace = app.CreateESpaceFromTemplate("Template_Reactive.oml",
    ↪ "MyModule");
eSpace.Description = "My First Module";

var entity = eSpace.CreateServerEntity("Person");
var idAttr = entity.CreateAttribute("Id");
idAttr.DataType = eSpace.IntegerType;
entity.IdentifierAttribute = idAttr;
var nameAttr = entity.CreateAttribute("Name");

app.Save("MyApp.oap");

```

Listing 4: ModelAPI

4.8 ModelAPI

The ModelAPI consists of an [Application Programming Interface \(API\)](#) used to manipulate the OutSystems model, currently corresponding to a set of .Net Framework DLL that allows for changes and creations of new elements in OutSystems applications. Most of this manipulation power is provided by the API's multiple model interfaces and a service interface. The former allows for altering language-specific elements such as web screens or entity attributes, while the latter provides the entry point methods to the ModelAPI, allowing e.g. to load or create an *ESpace*. In addition, this API provides the capability of creating and managing keys, related to the application and its model elements, and is endly coupled with the behavior of the OutSystems' [builders](#). Listing 4 demonstrates the ModelAPI way of operation. As we can see, the mentioned model services are loaded in order to create the "MyApp" application. Consequently, an *ESpace* is initialized using a pre-defined [OutSystems Markup Language \(OML\)](#) module to which an *Entity* is added with an id as well as a name attribute.

4.9 Builders' Key Management

In the OutSystems environment, a builder can be divided into two categories: Single-Shot or Multi-Shot. This distinction concerns their usage of the ModelAPI's generation and

control mechanisms associated with the above-mentioned keys. As a **Globally Unique Identifier (GUID)**, a key unambiguously identifies either an application or the application's elements. This comes across as a device of paramount importance since it bounds the **Platform Server's** behavior in terms of the management of the resources associated with the application itself. The **Server** has to analyze each incoming generation request and establish, based upon their **app GUID**, if it effectively corresponds to the generation of a new application or if it concerns an update to an existing application. This discernment is made possible with the analysis of the referred **GUID**. By default, the **GUIDs** are defined randomly for both application of its elements, hence without explicitly specifying an identifier, an arbitrary one shall be assigned. Therefore, in the circumstances regarding an application update, the **app's** elements must go through a more granular evaluation. Taking the example of an *Entity*, for instance, if by mistake, its identifier is changed, a new table will be created, consequently causing the loss of all data related to the former *Entity*.

4.9.1 Single-Shot Builders

On the one hand, the single-shot mode of action is defined by its ability to generate each application at most once, meaning the definition of an application can only be made once, leading to future re-generations producing different apps. This means that when an application is published a subsequent time, using a builder of this type, the more recent version of the **app** will always replace any older version, hence ignoring any eventual changes made to the past version, creating from scratch what is ultimately a new application, with the same **GUID**. Internally, this occurs due to every new publication causing the assignment of freshly generated identifiers to every application and their elements.

4.9.2 MultiShot Builders

On the other hand, MultiShot Builders can redefine an existing application multiple times, by explicitly declaring and associating formerly used keys to new versions of previously defined applications and their elements. This means that multiple publications will lead to new iterations of the same **app**. The genesis of this feature lies upon the key re-usability mechanisms provided by the ModelAPI, predicated on a deterministic key generation process and on the use of known keys.

Nowadays, these mechanisms offer two modes of operation:

- **Full App Regeneration:** starting from “scratch” and producing what is technically a new application, but in reality, it consists in a new version of a previous **app**. This strategy requires the guarantee that keys are kept the same for the application

```
1 <Module key="68976C33-0F6F-4D13-8B8F-DCFC69000E0A">
2   <Entities>
3     <Entity key="D88C448E-JR91-42D8-B516-B275553073FB">
4       <Attribute name="Name" type="Text" key="..." />
5       <Attribute name="Type" type="Text" key="..." />
6       <Attribute name="Price" type="Currency" key="..." />
7       <Attribute name="Review" type="Integer" key="..." />
8     </Entity>
9   </Entities>
10 </Module>
```

Listing 5: Model with Key Management Mechanisms - MultiShot Operation Mode

modules and any element that exists in consecutive versions. This approach is highlighted by its efficiency, maintainability, and reduced code size, however, it requires strict control of the application elements' keys and proper element identification.

- **Incremental Model Generation:** allows for an approach more oriented towards a continuous development approach, where changes in subsequent iterations of the same application must be evaluated and compared with the previously generated one, in order to calculate differences and produce a version that can correctly reflect all the made updates. It should be clear that this alternative is more demanding, due to its necessary version control.

Listing 5 considers an XML representation and approximation of an OutSystems Application model. It depicts a *Module* composed of an *Entity* with multiple attributes. For demonstration purposes, the represented model assumes an abbreviated form, although it provides a sufficient illustration of the discussed issue. The used **GUID** associated with the *Module* and the *Entity* are values inserted explicitly by the developer using the Mode-API with the main objective of having the **Platform Server** to consider previous versions of those same application elements and execute an update. As mentioned, when these keys are not passed explicitly in the Single-Shot mode of operation, a new **GUID** will be automatically generated which results in a new application, or application element as far as the **Platform Server** is concerned.

RELATED WORK

This chapter describes the literature research that was made to contextualize and justify the defined approach towards this problem.

5.1 Delta-based model transformations

A considerable portion of this thesis comes from the challenge of developing bidirectional transformations able to address change propagation between a model and what can be perceived as its view since it deals with information originated from the model but represented in higher abstraction. Having mentioned that, a bidirectional transformation applied to this model-view coupling will have to take into account that multiple states of a model might correspond to the same view in a many-to-one type of relationship.

Traditional approaches generally correspond to state-based synchronizations i.e. a bidirectional transformation that considers exclusively the model state while propagating changes from the source model to the target model. These procedures use a specific toolset responsible for mapping the elements of the source model to the corresponding elements in the target model. Subsequently, it detects changed elements in the source model and updates the target. However, this strategy has been connected with several frailties requiring significant technical effort, added complexity, and costs, largely as a result of the mapping phase, known as model alignment or model differencing.

Diskin *et al.* portray in [4] a pertinent strategy closely tied with this thesis' approach. In their paper, the authors purpose a "delta-based" algebraic framework that tackles the shortcomings of state-based approaches. The authors define the term delta as "*a specification of commonalities and differences*" between two distinct states of the same model.

This approach describes the process of update propagation across models as a two-stage operation: in a first instance, is prompted an assessment of the differences between the models in consideration, producing the above-mentioned deltas. This step is referred to as model alignment or differencing and noted as *dif*. Secondly, the computed deltas are propagated and applied to the models in question, in a step noted as *dput*.

As to exemplify the described transformation process let's consider model state A

and view state B . User interaction with B induced a new state to the view, B' , and the modifications can be represented by $\Delta_{BB'}$. In order to propagate the user changes, the transformation should only consider the alignment of the changed view B' and the model A , hence the update propagation operation *dput* takes place producing $\Delta_{AA'}$ which reference the differences between the initial model A and its updated version A' . By applying these differences to the initial model A , we obtain A' . This can be summarized by the following schema:

$$A' = \Delta_{AA'}(A), \text{ where } \Delta_{AA'} = \mathbf{dput}(\Delta_{BB'}, A) \text{ and } \Delta_{BB'} = \mathbf{dif}_Y(B, B')$$

Figure 5.1 illustrates the process of transformation using the procedures mentioned above. On one side of the transformation, we have a concrete model P representing a *Person* identified by his or her first name, last name, and birth date. On the other side, subsists the corresponding view model Q , representing the same *Person* elements omitting their birth dates. The *get* function considers the *state* A of the concrete model and produces the corresponding view *state* B . User interaction with the view produces a new state where the *q1* element was deleted. This, in the light of the author's approach, leads to the generation of the delta b that captures the applied modifications through the model alignment process ($\mathbf{dif}_Y(B, B')$). In order to propagate the changes made to the view, the *dput* is applied considering *state* A , that is, the concrete model state which originated the view, and b , the delta associated with the modification itself. This process culminates in the creation of $\Delta_{AA'}$, i.e. a , which will consequently be applied to *state* A , therefore restoring consistency to the concrete model.

5.1.1 Benefits from using a two-stage operation

This division into two separate operations empowers the approach with increased modularity, allowing users to tweak the model transformation procedure in greater detail, according to their needs. Interesting examples of these added customization capabilities relate to the granularity of the delta discovery, as users can define which model differences should be considered (in the previously illustrated schema, this functionality refers to the Y parameter in the \mathbf{dif} function). The users can also edit the result of the model alignment phase, adding or removing differences between models, therefore customizing the subsequent delta propagation.

Another interesting aspect addressed in this paper, which should be explored in the thesis relates to the computation of deltas. The authors suggest that the model alignment phase doesn't necessarily rely on an actual comparison between two stages of a model. In fact, Diskin *et al.*, indicate that "*if the synchronizer can be tightly coupled with the application, deltas can be obtained by recording the user operations within the application*"[4]. This means the model alignment phase can be skipped if the performed changes in the model are registered in some variety of logs.

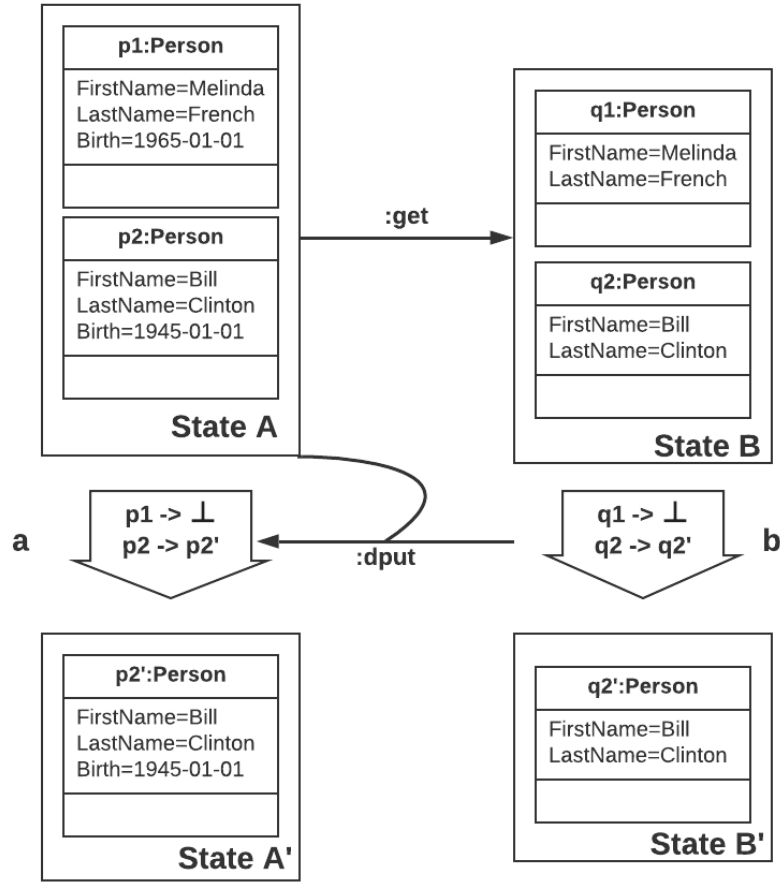


Figure 5.1: Diskin Delta-based Model Transformation (Adapted from [4])

5.2 Concurrent update propagation

The interoperability across the discussed development tools implies a scenario where multiple interconnected models are changed simultaneously. This parallel process prompts the necessity of implementing consistency-maintenance mechanisms capable of supporting concurrent updates. Takeichi *et al.*, in [29] describe a synchronization algorithm that can be applied to bidirectional transformation pipelines that satisfy certain requirements, in order to "ensure a reasonable synchronization behavior".

5.2.1 Requirements for synchronizing concurrent updates

The authors acknowledge the synchronizer as a partial function of the following type:

$$\text{sync} : R \times (M \times N) \Longrightarrow M \times N$$

where M and N as meta-models and $R \subseteq M \times N$ is the consistency relation needed to be established. As an input, the function considers four models: the two original models satisfying consistency relation R , and their respective updated models, while the output corresponds to the two new models for which the updates are synchronized. As a function,

the synchronizer implies a deterministic behavior, i.e. for two equal inputs, the function should produce an equal output. Due to its partial nature, the function also indicates that detection of conflicts in updates should occur, meaning that if the updates to the two models lead to a conflict, the function should be undefined for the input. Apart from these inferred requirements, Takeichi *et al.* argue three additional properties to ensure the synchronizer behaves expectedly:

- **Consistency:** requiring the synchronization process to generate output models preserving consistency, i.e. relation R is established on the resulting models.

$$\text{sync}(m, n, m, n) \text{ is defined} \implies R(\text{sync}(m, n, m, n))$$

- **Stability:** If neither of the two models, m or n , has been updated, the synchronizer should not modify any model.

$$R(m, n) \implies \text{sync}(m, n, m, n) = (m, n)$$

- **Preservation:** In particular scenarios, user intervention is necessary to select which updates should be selected to generate a consistent model. This selection process is generally application-specific and the authors state there should be "*an update preservation relation $P_M \in M \times M \times M$ for any model M , where $P_M(m_o, m_a, m_c)$ implies that the update from m_o to m_a is preserved in m_c . Thereby, users can define particular preservation requirements by defining different preservation relations.*" Formally, let $P_M \in M \times M \times M$ be a preservation relation over M , and $P_N \in N \times N \times N$ be a preservation relation over N .

$$\text{sync}(m, n, m, n) = (m, n) \implies P_M(m, m, m)$$

$$\text{sync}(m, n, m, n) = (m, n) \implies P_N(n, n, n)$$

5.2.2 Algorithm

Before describing the behavior of the planned algorithm, it is relevant to mention that the authors construct a *three-way merger* following the Diskin *et al.* differencing and update propagation operations, previously described. The *three-way merger*, given an original model m_o and two independently modified copies, m_a , and m_b , is a partial function defined as the following:

$$\text{merge}(m_o, m_a, m_b) = (\text{diff}(m_o, m_a) + \text{diff}(m_o, m_b)).\text{post}$$

where *diff* corresponds to the assessment of the differences between two models, therefore, resembling the *model alignment* operation in the Diskin *et al.* approach. The $+$ is a union operation used to merge distinct updates to be applied to the same model, while *post* refers to the application of the resulting differences, which will lead to the propagation of the updates.

The algorithm's input requires two pre-updated models, corresponding to m_o and n_o , and the respective pos-updated versions m_a and n_b . Firstly, is invoked backward transformation \overleftarrow{R} to propagate the updates made to n_b to m_o , therefore generating m_b . Secondly, the three-way merger is constructed regarding model m_a containing update a and model m_b that contains update b , thereby producing a synchronized model, m_{ab} , on the M side. If the updates to the two models conflict, the merger operation will detect the conflict and report an error. The following step is to use forward transformation \overrightarrow{R} to produce a synchronized model, n_{ab} , on the N side. Finally, a preservation testing procedure is executed, in order to check whether the update from m_o to m_b is preserved in n_{ab} .

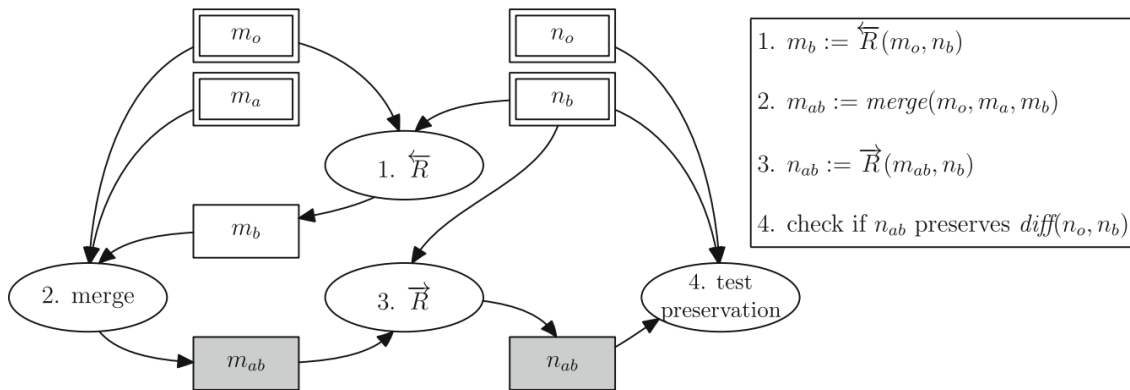


Figure 5.2: Takeshi *et al.* Algorithm [29]

5.3 In summary

The delta-driven approach discussed by Diskin *et al.* directly addresses this thesis' issue, providing valuable fundament for the planned strategy since the envisioned transformations regarding the **builder-Service Studio** direction require segregating and consequently isolating the change operations made in the builder from their actual application to the OutSystems model.

Takeshi *et al.* work is extremely pertinent when discussing the edition of different models in a concurrent and parallel environment of multiple working developers. The envisioned solution will consider the requirements stated by the authors as paramount to a fully working synchronization of concurrent updates. Despite these being valuable matters to be addressed, the overall performance of the development and implementation phases of the prototype will ultimately dictate the concrete course of action.

DESIGN AND IMPLEMENTATION

The design and implementation process intended to address the several solution aspects, starting from less technical issues and gradually advancing to more challenging areas of the problem. This chapter describes the reasoning behind the design decisions and depicts the implementation process in thorough detail.

6.1 Design

The Experience Builder was the tool chosen to develop and implement our approach. However, it was crucial to come up with a solution that not only contemplated the other existing builders but could also serve as a proof-of-concept to future cooperations between development tools and the Service Studio. An important part of the successful cooperation between those tools correlates with an updated view of the current state of an application. Thereby, the solution should guarantee that when a new builder session is started by a developer, the tool should have access to all existing applications in the developer's environment, and not only the applications that were originally created using the tool. Secondly, a pivotal part of collaborative work across tools is continuous and iterative development, therefore the solution had to ensure that changes made to an application using a builder didn't disregard changes made using Service Studio. Our approach had to circumvent the current *Single-Shot* mode of operation currently employed by the builders.

These limitations should be addressed by defining fully bidirectional builder model transformations in OutSystems: builder to *IDE* and *IDE* to builder. This culminated in a synchronization algorithm that maintains consistency thus leading to the point where the builder becomes an editor.

6.1.1 Strategy Overview

The envisioned strategy was comprised of two distinct parts, which aimed to address the issues mentioned in the objective section, employing model transformations involving

both the **OutSystems model**, denoted as M , and the builders model also referred to as the **view model** and denoted as V .

Firstly, we should consider the necessary procedures to enable application development in the Service Studio-builder direction. These will comprise the *forward transformation*. Secondly, we will contemplate the process that took place regarding the opposite direction, i.e. builder-Service Studio, which consequently shall be described as the *backward transformation*.

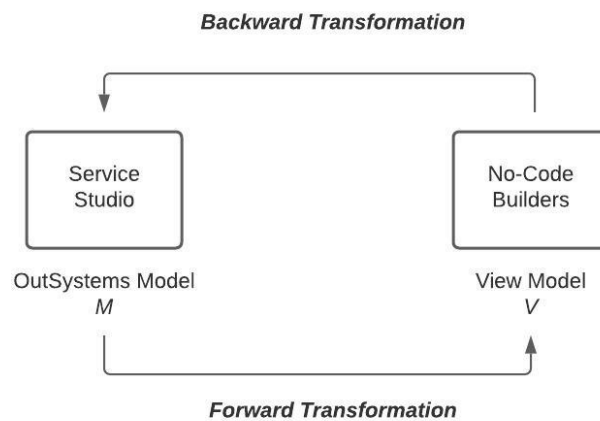


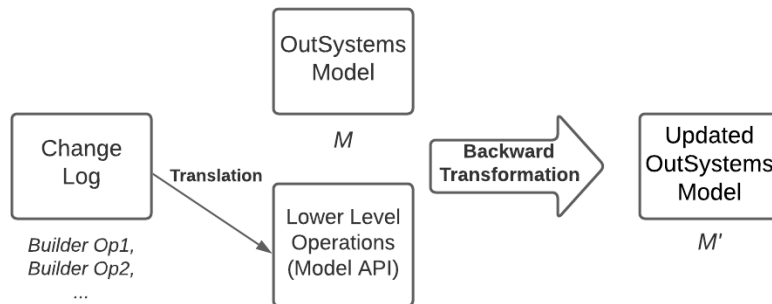
Figure 6.1: Used Notation

Regarding the Service Studio-builder direction, i.e. the *forward transformation* process, the solution had to ensure that editions made to an application using a builder did not disregard changes made using **Service Studio**, hence, the view model V had to be obtained by dynamically calculating which elements of the model M had to be considered in the builder. To attain this goal, the proposed approach considered a **Presenter-like** component, capable of applying a transformation to the OutSystems model as a way of obtaining meaningful elements to the context of the builder. This process, in other words, corresponds to building a view model adjusted to the builder's concrete displayed view. Therefore, the transformation requires the definition of a **projection** aware of the target view, and hence, capable of locating and extracting the relevant concepts existing in the OutSystems model. Furthermore, since the **app** elements in the **IDE** conform to a lower-level representation, the transformation had to implement a conversion to the higher abstraction specific to the Experience Builder.

In terms of the *backward transformation*, the design phase suggested employing a **log** component assigned with registering changes made to the view model V . This component was responsible for gathering the update operations carried out by the developer during his interaction with the builder. Similarly to the reverse transformation, the abstraction differences required a "translation", in this case, a conversion process that would produce a sequence of more elementary and lower-level **ModelAPI** operations, which in turn could

be applied to the OutSystems model M , resulting in the propagation of changes.

The necessity of this conversion process can be easily demonstrated if one considers the example of a *Menu* in the *Experience Builder* and in *Service Studio*. While the builders' meta-model addresses the concept of *Menu* directly, in *Service Studio* the same element must be constructed using several other components since the *IDE* meta-model is oblivious to the concept of *Menu*.



Regarding the *log* component involved in backward transformation, there were two choices whose viability would be explored:

- **Builder-Log Coupling:** The first alternative suggested that while the developer edits an application in the builder, these update operations should be registered in the mentioned *log* component. Hence, a tight coupling had to be implemented between the builder and the referred record.

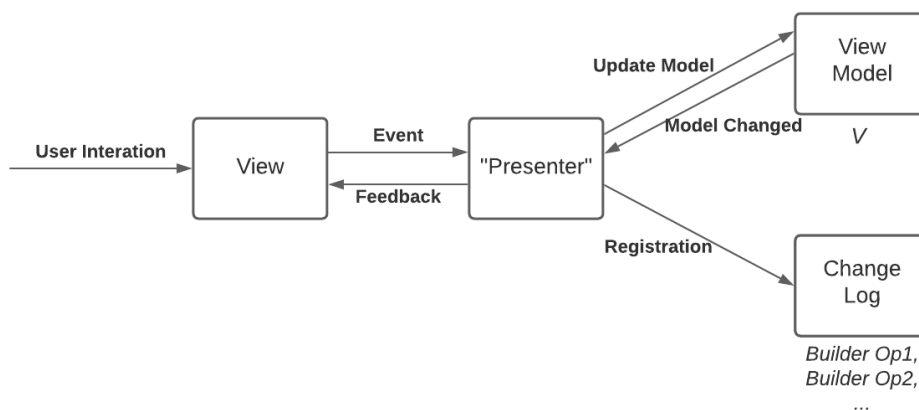


Figure 6.3: Backward Transformation - Builder-Log Coupling

- **Difference Calculation:** The second alternative considered obtaining the modification *log* through an automated process capable of assessing the differences and commonalities between two model states: the original view state, V , and the changed

view state V' , in what can be noted as $dif(V', V)$. This approach would benefit from the implementation of a *Single-Unified-Model* amidst the builders, i.e. a meta meta-model common to every builder, responsible for defining a set of general rules and constraints aiming to standardize model representation and instantiation. Through the employment of a shared meta meta-model, the model differencing procedure could be made more efficient, since it wouldn't have to deal with the current distinct representations amongst the builders. Nevertheless, the calculated differences could subsequently be converted to the necessary [ModelAPI](#) operations.

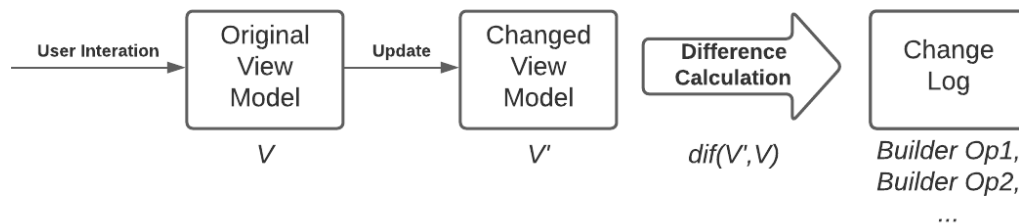


Figure 6.4: Backward Transformation - Difference Calculation

It is important to underscore that the envisioned solution operated on the issue of model transformations by considering model changes as the central characters in the update propagation process. With the aid of the above-mentioned changelog, the execution mechanism of the backward transformation would focus on the source model elements that suffered modifications, thus achieving added modularity and isolation of operations.

6.2 Implementation

6.2.1 Prototype Plan

The developed prototype was envisioned to be an augmented version of the Experience Builder capable of extending the offered interoperability with Service Studio. However, from the start, the main objective was to produce a prototype that could replicate the offered functionalities of the builder, allowing users to customize the application's screens, [flows](#), [connections](#), and menu items. The development of the prototype would naturally use as a referential the code associated with the Experience Builder since it aimed to be an extension of it. Nevertheless, unlike the OutSystems tool, the developed prototype wouldn't focus on the generation of a new application from a JSON model. Instead, the prototype would utilize an existing application with a known JSON model representation, and allow the user to change it by creating, updating, or deleting elements such as flows, connections, etc. This enabled an increased focus when tackling the problem of enabling collaborative development through bidirectional transformations.

6.2.2 Console App Functionality

Having a GUI would be beyond the scope of our work therefore, the prototype would be instantiated as a console exposing the commands associated with the features to be studied, those being:

- **load** <file name> - Loads the JSON model with the given file name.
- **save** <file name> - Saves the loaded JSON model to the given location.
- **diff** <file name> - Calculates differences between the loaded model and the model with the given filename.
- **avai_flows** - Prints the flows that could be added to the JSON model.
- **add_flow** <flow key> - Adds the JSON model a new flow with the given key.
- **rem_flow** <flow key> - Removes from the JSON model, the flow with the given key.
- **chg_flow_name** <flow key> - Changes the name of the flow of the JSON model with the given key.
- **avai_exitpoints** - Prints the link keys that could be used to add connections to the JSON model.
- **add_connection** <link key> <destination screen key> - Adds a new connection to the JSON model between the screen with the given link key and the screen with the given key.
- **rem_connection** <link key> - Removes the connection of the JSON model with the given link key.
- **chg_connection** <link key> <new destination screen key> - Changes the connection of the JSON model with the given link key setting it to the screen with the given key.
- **chg_screen_name** <screen key> <new screen name> - Changes the name of the JSON model's screen with the given key to the new given name.
- **avai_icons** - Prints all the menu item icons, and captions that could be added to the JSON model.
- **add_menu_item** <item name> - Adds to the JSON model menu a new menu item with the given name.
- **rem_menu_item** <item id> - Adds to the JSON model menu a new menu item with the given item id.

- **chg_menu_item_cap** *<item id>* *<new caption>* - Changes the caption of the JSON model's menu item with the given id
- **chg_menu_item_ico** *<item id>* *<new icon name>* - Changes the icon of the JSON model's menu item with the given id
- **chg_menu_item_scr** *<item id>* *<new screen key>* - Changes the screen of the JSON model's menu item with the given id
- **chg_menu_item_ord** *<item id>* *<new order>* - Changes the order of the JSON model's menu item with the given id.

In addition to the currently available functionality provided by the Experience Builder, the developed prototype, in order to address the goal of this dissertation, had to implement **two additional features** regarding the **backward and forward transformation**:

- **back_transf** *<original JSON model>* *<changed JSON model path>* *<OAP path>* - Executes the backward transformation considering the differences between the original and the changed JSON models, and applies those changes to the OAP file in the given path.
- **forw_transf** *<OAP path>* *<projection JSON model path>* - Executes the forward transformation, considering the OAP file in the given path, producing a JSON model to the given path.

```

C:\Users\jah\Desktop\AppBuildersToEditors\BackwardTransformationPrototype\bin\Debug\netcoreapp3.1\BackwardTransformationPrototype.exe
Name: ItemDetails; Description: Details of a product. User can add the product to the shopping cart.
Name: ItemList; Description: List of products. User can use voice search to find specific products. The list can be filtered and sorted.
Name: FavoriteItems; Description: List of favorite items saved by user.
Name: FeaturedItems; Description: List of featured items.

Flow Name: ShopBagAndCheckout;
Name: ShoppingBagCheckout; Description: Shopping bag with added products and checkout process to buy them.

Flow Name: Settings;
Name: ProfileSettings; Description: User profile, including personal information, and preferences.
Name: ChangePasscodeOK; Description: Success message after changing an authentication passcode.
Name: ManageNotifications; Description: Set up notification settings preferences.
Name: NotificationDetails; Description: Notification settings detail. Can edit settings per notification.

Flow Name: ShopBranchLocations;
Name: ShopBranchLocationDetail; Description: Check a branch's detailed information, and find out how to get there.
Name: ShopBranchLocator; Description: Find branches, searching by location.

>> load TinyApp.json
Model ready to be used.

>> chg_flow_name wrongkey NewName
Flow does not exist.

>> chg_flow_name 8cb7b67a-083f-461c-8f2c-c57b9333f698 NewName
Flow was changed with success.

>>

```

Figure 6.5: Prototype - Console Application


```

{
  "applicationInfo": {
    "appKey": "a4245d4e-be5b-4317-8827-8721094267f7",
    "name": "TinyApp",
    "description": "",
    "icon": {},
    "splashscreen": {}
  },
  "visualProperties": {
    "primaryColor": "#C3272B",
    "nativeThemeName": "ExB_UI",
    "extensibilityProperties": "{\\"resource\\" : \\"res.zip\\",\r\n
    ↵ \\"splashscreens\\" "..."},
    "iconTextColor": "#FFFFFF"
  }
}

```

Listing 6: JSON Payload - Basic Info

6.2.3 TinyApp

6.2.3.1 JSON Payload

The application used was called *TinyApp*, and as the name suggests it was the most simplistic [app](#) the Experience Builder could generate. The Experience Builder team provided the JSON of the application holding all the information defined by the user when interacting with the builder's [User Interface \(UI\)](#), which would be sent in an application generation request to the company's [AWS](#) infrastructure. Since all the data is included in a single request, its payload is extensive. The request holds basic information pertaining to the application's name, description as well as the application key, a value that uniquely identifies the [app](#) in the OutSystems platform. The request also gathers information regarding the visual properties of the [app](#), concerning its primary color or its native theme for instance.

When it comes to the scope of this dissertation, the most relevant segment of the application generation request had to do with the actual representation of the core elements of the application in the Experience Builder. The [TinyApp](#) was composed of three flows, *AnimatedOnboardingOption1*, *SimpleLogin*, *BankingDashboard*, each one including only one screen with the same name. When it comes to the [connections](#) among [flows](#), the onboarding screen directed the user to the login, and upon correct credentials, the user was subsequently directed to the dashboard screen.

Regarding the application menu, the *TinyApp* defined a single menu item which upon click would lead the user to the *BankingDashboard* screen.

In addition to the above-mentioned, the JSON request also held information concerning the modules to be generated and the location of the required resource templates. The relevance of these segments of the payload will be addressed later on.

6.2.3.2 OAP

To adequately investigate how to enable the bidirectional interoperability between the builder and Service Studio, not only should be considered the Experience Builder model but also the OutSystems model, instantiated in the form of [OutSystems Application \(Package\)](#) (OAP). That being said, the depicted JSON request, as previously mentioned, would be responsible for triggering a new app generation process taking place in the OutSystems [AWS](#) instances. To produce the new application, the builder has to locate and therefore load the resources specified in the payload. However, currently, part of the assets necessary to fully instantiate any application is stored in [S3 Buckets](#) in the form of OutSystems modules in [OMLs](#) format. Most of these resources correspond to modules gathering sets of [UI flows](#), grouped by similar use-cases. For instance, a module may include *Banking* related flows, while another one can be associated with *Healthcare* applications.

However, due to troubles concerning access permissions to this cloud infrastructure, it was deemed more viable to download the needed resources and locally run a version of the Experience Builder which was not dependent on the AWS S3 buckets. This version of the code made possible the generation of the *TinyApp* now in [OAP](#) format.

As expected, the app was composed of the three flows specified in the JSON payload, with all the provided information, as well as a built-in [UI flow](#) - the *Common Flow* - constant to any application created with the builder, comprising several screens linked with typical scenarios of use such as the *OfflineScreen* or the *ErrorScreen*, along with common building blocks such as the *BottomBar*, the placeholder for the application's menu.

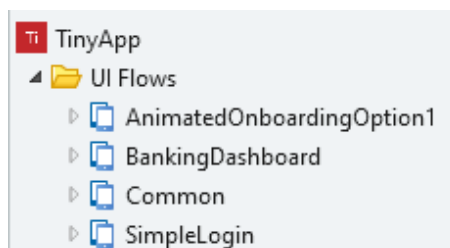


Figure 6.6: Service Studio - Application (Flows)

```

{ "screens": [
  {
    "name": "AnimatedOnboardingOption1",
    "description": "Adding an onboarding is a great way to
↪ communicate",
    "flowType": 3,
    "screenKey": "qUWoGIVL80+8g_T3mCpujQ",
    "flowName": "AnimatedOnboardingOption1",
    "flowKey": "fb9cc0cc-4e4f-4d3b-bc49-b5a40f98a5a8",
    "sourceModuleKey": "63W0ewv1EEuH5jvx0qSjcQ",
    "hasMenu": false,
    "orderInFlow": 1
  },
  {
    "name": "SimpleLogin",
    "description": "Simple login screen.",
    "flowType": 1,
    "screenKey": "sPQ2_NuZpUeemrTDykS2qw",
    "flowName": "SimpleLogin",
    "flowKey": "2321090f-5b16-4b71-8cd9-d327ed7f6ca7",
    "sourceModuleKey": "63W0ewv1EEuH5jvx0qSjcQ",
    "hasMenu": false,
    "orderInFlow": 1
  },
  {
    "name": "BankingDashboard",
    "description": "Transfer money to an account, identified by a
↪ payee.",
    "flowType": 5,
    "screenKey": "ELqk0A1CHk2QIcTQ9aahcA",
    "flowName": "BankingDashboard",
    "flowKey": "8cb7b67a-083f-461c-8f2c-c57b9333f698",
    "sourceModuleKey": "paYD5xm6q0KzdPLVwEXr8Q",
    "hasMenu": true,
    "orderInFlow": 1
  }
]
}

```

Listing 7: JSON Payload - Model (Screens)

```
{ "connections": [  
  {  
    "linkKey": "3GwQ01xED0CxkAR1XLZ3Ag",  
    "linkType": 4,  
    "originScreenKey": "qUWoGIVL80+8g_T3mCpujQ",  
    "destinationScreenKey": "sPQ2_NuZpUeemrTDykS2qw"  
  },  
  {  
    "linkKey": "iX3VD_sozk0tr12j26mT1A",  
    "linkType": 2,  
    "originScreenKey": "sPQ2_NuZpUeemrTDykS2qw",  
    "destinationScreenKey": "ELqk0A1CHk2QIcTQ9aahcA"  
  }  
]  
}
```

Listing 8: JSON Payload - Model (Connections)

```
{ "menu": {  
  "typeId": "1",  
  "items": [  
    {  
      "ssMenuItemId": 1,  
      "caption": "Dashboard",  
      "iconName": "star",  
      "targetScreenName": "",  
      "order": 0,  
      "targetScreenKey": "ELqk0A1CHk2QIcTQ9aahcA"  
    }  
  ]  
}  
}
```

Listing 9: JSON Payload - Model (Menu)



Figure 6.7: Service Studio - Application (*AnimatedOnboardingOption1* Screen)

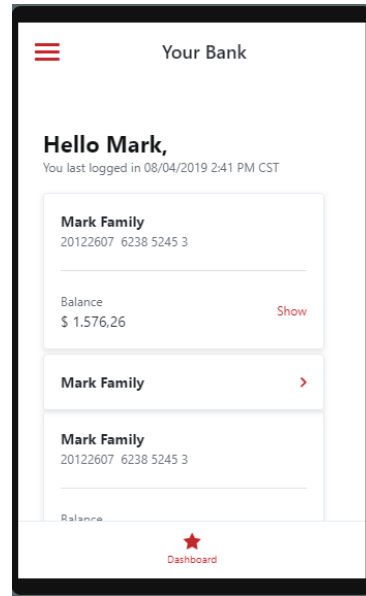


Figure 6.8: Service Studio - Application (*BankingDashboard* Screen)

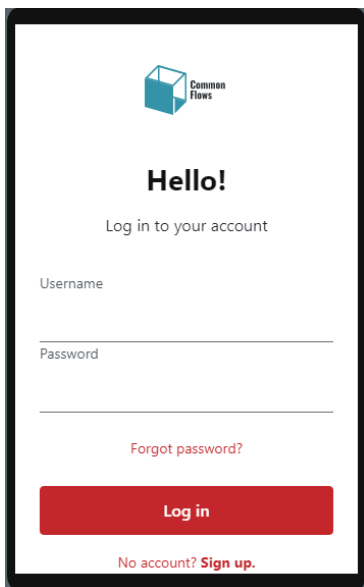


Figure 6.9: Service Studio - Application (*SimpleLogin*) Screen

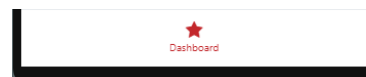


Figure 6.10: Service Studio - Application (Bottom Bar Menu)

Listing 10: Implementation - Experience Builder Model

6.2.3.3 Roadmap

The development of the prototype was divided into several steps that were addressed sequentially. The completion of each phase laid the foundations of the next one. The following list describes the multiple planned stages of the prototype development:

- **Backward Transformation:**
 - **Serialization/Deserialization of the builder (JSON) models:** To compare and identify changes in the builder models, the prototype had to not only initialize the model objects in memory by loading the JSON, but it also had to be able to save the model state into JSON format.
 - **Delta Computation:** In order to track the changes made to an existing builder model, a comparison process had to be implemented.
 - **Operation "Translations":** The backward transformation was only completed when the change operations, made to the builder model, had been propagated to the OutSystems model. Hence, the high-level builder operations had to be converted into sequences of lower-level operations that could be applied to the OutSystems model.

- **Forward Transformation:**
 - **Metadata injection:** In order to restore the builder model, it was mandatory to assess what builder data was missing in the OutSystems model. After identifying such data, it was necessary to add the missing elements in the model.
 - **Projection:** With all the needed data, the projection only required the mapping of OutSystems model elements to Experience Builder model elements.

6.3 Backward Transformation

6.3.1 Serialization/Deserialization of the builder model

Firstly it was necessary to define the shape of the builder model object when the deserialization process occurred. As previously mentioned the prototype would extend the currently offered functionality by focusing on the core elements of the Experience Builder, those being the menus, [flows](#), screens, and [connections](#). Naturally, this meant the model object, when initialized, had to be composed of those same data structures. Figure 10 depicts the mentioned structures of the Experience Builder model class.

The actual deserialization process i.e. the loading of a builder model was relatively simple to implement. The chosen strategy was to use the Newtonsoft library [1] that

```
1 private ExperienceBuilderModel LoadModel(string filePathSource) {
2     using StreamReader file = File.OpenText(filePathSource);
3     JsonSerializer serializer = new JsonSerializer();
4
5     var model = (ExperienceBuilderModel)serializer.
6         Deserialize(file, typeof(ExperienceBuilderModel));
7
8     JObject modelJSON =
9         ↪ JObject.Parse(File.ReadAllText(filePathSource));
10    model.InitializeModelObjects(modelJSON);
11
12    return model;
13 }
```

Listing 11: Implementation - *LoadModel*

provided a **serializer** capable of serializing and deserializing objects into and from the JSON format. As expected, the serializer object needed the type of object to be deserialized. When successfully implemented, the prototype was then able to load into memory and initialize a builder model by consuming its JSON representation. Figure 11 shows a snippet of the *LoadModel* method responsible for the deserialization.

When it comes to the serialization i.e. the saving of the builder model object back to JSON format, the process resembled the deserialization described previously to a great degree. The expected difference had to do with the employment of the `serialize` method presented by the JSON serializer, responsible for saving the in-memory model object to the provided file destination according to the defined formatting options. This can be seen in the figure 12.

6.3.2 Delta Computation

The backward transformation process relied on the ability to propagate all the changes made to an application using the Experience Builder without overlooking or damaging any modification made to the app in Service Studio. Hence, the first step to be taken had to do with the identification of the changes made by the builder user. The solution had to be able to analyze and compare two distinct builder models: the original one, corresponding to the current state of the application, obtained through the projection phase; and the changed model, resulting from modifications applied to the original model, employing the provided Experience Builder functionalities.

In order to pinpoint any change made to a builder model, it was mandatory to firstly define what set of the modifications made possible in the Experience Builder would be

```
1 public object SaveModel(string filePathDestination, object model) {
2     using StreamWriter file = File.CreateText(filePathDestination);
3     var serializerSettings = new JsonSerializerSettings
4     {
5         ContractResolver = SerializerContract.Instance,
6         Formatting = Formatting.Indented
7     };
8     JsonSerializer serializer =
9     ↪ JsonSerializer.Create(serializerSettings);
10    serializer.Serialize(file, model);
11    return model;
12 }
```

Listing 12: Implementation - *SaveModel*

captured by the prototype in the **delta computation** process. The established strategy was to focus on the major functionalities provided by the builder, those being the creation, update, and deletion of **flows**, **connections**, and menu items. Hence, minor operations were disregarded, such as the instantiation of blank screens or the possibility of changing the application menu type (from a bottom bar to a side menu, or vice-versa). Thereby, the following list describes the types of operations that were captured, as well as the attributed notation, which will be of greater interest further on:

- Element Creation:
 - *NewFlow*: new added flow;
 - *NewConn*: new added connection;
 - *NewMenuItem*: new added menu item;
- Element Update:
 - *UpdFlow*: updated flow name;
 - *UpdScr*: updated screen name;
 - *UpdMenuItemCaption*: updated menu item caption;
 - *UpdMenuItemIcon*: updated menu item icon;
 - *UpdMenuItemOrder*: updated menu item order;
 - *UpdMenuItemTargetFlow*: updated menu item target flow;
- Element Deletion:
 - *DelFlow*: deleted flow (respective screens and connections);

Algorithm 1 Delta Computation Phase

```

input
  originalEBModel           ▶ Experience Builder Model
  changedEBModel           ▶ Updated Experience Builder Model
output
  osModel                   ▶ Updated OutSystems Model

1: function BackwardTransformation(originalEBModel,
2: changedEBModel,osModel)
3:   curModelFlows←GetFlows(originalEBModel)
4:   newModelFlows←GetFlows(changedEBModel)
5:   changeOperations←CalculateDifferences(curModelFlows,
6:   newModelFlows)
7:   for all operation in changeOperations do
8:     RunOp(operation,osModel)
9:   return osModel

output
  changeOperations         ▶ Operations necessary to reinstate consistency

10: function CalculateDifferences(curModelFlows,newModelFlows)
11:   addedFlows← Exclusion(newModelFlows,curModelFlows,
12:   flowsSameKeyNameScrNames)
13:   for all flow in addedFlows do
14:     changeOperations←NewFlowOp(flow)
15:   remFlows← Exclusion(curModelFlows,newModelFlows,
16:   flowsSameKeyNameScrNames)
17:   for all flow in remFlows do
18:     changeOperations←DelFlowOp(flow)
19:   updFlows← Intersection(addedFlows,remFlows, flowsSameKey)
20:   for all flow in updFlows do
21:     flowsUpdatedName← Intersection(flow,curModelFlows,
22:     flowsSameKeyDiffName)
23:     changeOperations←UpdFlowOp(flowsUpdatedName)
24:     for all screen in GetFlowScreens(flow) do
25:       screenUpdName← Intersection(screen,screensCurrModel,
26:       screensSameKeyDiffName)
27:       changeOperations←UpdScrOp(screenUpdName)
28:   return changeOperations

```

- *DelConn*: deleted connection;
- *DelMenuItem*: deleted menu item;

In this subsection, we portray in pseudo-code the developed algorithm thus illustrating in greater detail how the model transformation operations are obtained and subsequently applied. For the sake of brevity, the displayed pseudo-code focuses on the Experience Builder flows, despite the algorithm, in reality, addressing the connections and menu items elements as well.

Both `INTERSECTION` and `EXCLUSION` are auxiliary functions responsible for executing set operations upon the model object collections. As the names suggest, one function will return the common elements between the collections, while the other shall obtain

the elements present in the first argument collection and not in the second one. Both selection processes will depend on the result of the comparator predicate.

`CALCULATEDIFFERENCES` function starts by obtaining the flows added to the new model iteration. This is easily obtained using the `EXCLUSION` function, as it leaves us with the flows with the same key, existing in the new model version and not in the current model version. The used comparator `flowsSameKeyNameScrNames` tests the equality regarding other fields, which is unnecessary at this moment but shall be crucial further on. The resulting flows are subsequently used to generate `NEWFLOWOP` operations.

A similar process occurs regarding the removed flows, with the expected difference of applying `EXCLUSION` function in a “reverse” manner as in this case, we are interested in getting the flows existing in the current model version and not in the new model version.

The process to obtain the updated flows is slightly more intricate. An update operation considers a screen or flow name change hence, an unchanged flow must present naturally the same flow key, equal name, equal screen names. The variable `updFlows` holds every performed update however, it is necessary to distinguish between changes in flow and, or screen names.

The `INTERSECTION` function call in line 20, will search for flows with updated names. While the `INTERSECTION` function call in line 23 will determine if the screen(s) of the current flow, obtained through the `GETFLOWSCREENS` function, have been updated.

6.3.3 Operation “Translations”

The **delta computation** phase is responsible for creating **operation objects** that refer to changes made to the Experience Builder model. This process produced a **changelog**, containing all the change operations that must be applied to the OutSystems model to restore application consistency among models and their respective tool environments. In the developed solution, this log (in the pseudo-code defined as `changeOperations`) corresponded to a data structure collecting the multiple change operation objects. For debugging purposes, this log could be printed out, in JSON format. Figure 13 illustrates an example of a **changelog** comprised of two *Update* operations: `UpdFlow` and `UpdScr`. As we can see, each operation object is formed by its type and information relevant to correctly map the element to be changed in the OutSystems model. All of this will be analyzed at greater lengths further on.

Each change captured in the delta computation generated a new operation object containing a `Run` method, comprising a particular sequence of `ModelAPI` operations that when applied to the OutSystems model would successfully propagate the modifications made with the builder.

Having mentioned that, when it comes to the implementation of the operations, the strategy was to create an abstract `Operation` class with a virtual method (`Run`), implemented differently according to the operation at issue, as seen in listing 14.

```
[
  {
    "operationObject": {
      "flowKey": "fb9cc0cc-4e4f-4d3b-bc49-b5a40f98a5a8",
      "name": "NewFlowName"
    },
    "operationType": "UPDATE"
  },
  {
    "operationObject": {
      "flowKey": "2321090f-5b16-4b71-8cd9-d327ed7f6ca7",
      "screenKey": "sPQ2_NuZpUeemrTDykS2qw",
      "name": "NewScreenName"
    },
    "operationType": "UPDATE"
  }
]
```

Listing 13: Change Log

```
1 public abstract class Operation
2 {
3
4     public string OperationType {get; set;}
5     public virtual IApplication Run(IApplication application,
6     ↪ IModelServices modelServices) { return null; }
7 }
```

Listing 14: Abstract Operation Class

```
1 public override IApplication Run(IApplication application,
2   IModelServices modelServices)
3   {
4       ESpace = GetFrontModule(application);
5       ModelServices = modelServices;
6       UpdateFlow();
7       return application;
8   }
```

Listing 15: *Run* method - UpdateFlow

It should be noted that most of the logic of the Run operations regarding any element creation (flows, connections, or menu items) was supported on the existing Experience Builder code, as the builder, currently addresses creation operations only. As pointed out before, this is due to the current “start from scratch” [approach](#), where every use-case addressed by the builder corresponds to the creation of new elements: if the user performs any **deletion** or **update** to an existing Experience Builder application, the tool will consider this new state as a completely different app and will **regenerate everything** once again. Therefore, the current Experience Builder code doesn’t accommodate any delete or update operations thereby, the new functionalities had to be built from the ground up.

To provide a practical example of the implementation regarding the Run method, let’s consider a changelog comprised of two change operations: a UpdFlow operation concerning the name change of a flow, in addition to a NewConn operation, as the name suggests the creation of a connection between two flows.

6.3.3.1 UpdateFlow Operation

The Run method (listing 15) starts by locating the FrontOfficeModule, the main eSpace of the application, responsible for providing functionalities and [app](#) resources with direct relation to the users (line 4). After loading the ModelAPI services (line 5), we start the concrete flow update.

In listing 16 the operation had to consider an object coupled with the key of the flow to be updated (line 3) (in the figure depicted as FlowWithUpdatedName). Subsequently, it was only necessary to iterate over the several MobileFlows of the eSpace, searching for a flow object with the correct key. Upon finding it, the update was complete with the flow’s name changed.

6.3.3.2 NewConn Operation

Let’s analyze the *NewConn* operation responsible for linking two flows together.

```

1 private void UpdateFlow()
2 {
3     IKey flowKey = ModelServices.ParseKey(FlowWithUpdatedName.FlowKey);
4     IMobileFlow flow = ESpace.MobileFlows.Single(f =>
5         ↪ f.ObjectKey.Equals(flowKey));
6     flow.Name = FlowWithUpdatedName.Name;
7 }

```

Listing 16: *UpdateFlow* method

```

1 public override IApplication Run(IApplication application,
2     ↪ IModelServices modelServices)
3 {
4     ESpace = GetFrontModule(application);
5     ModelServices = modelServices;
6     ConnectScreens();
7     return application;
8 }

```

Listing 17: Create connection - *Run* method

The *Run* method, once again starts by locating the *FrontOfficeModule*. After loading the *ModelAPI* services, everything was set to start instantiating the new connection.

The main method *ConnectScreens* 18 commences by considering the *Connection* object, the new connection to be instantiated. Two variables are initialized with the corresponding keys of the origin and destination screens composing this connection in lines 3 and 4. The *Connection* object stores these values in string format however, the keys of the *OutSystems Model* objects subsist as implementations of the *IKey* interface thereby, parsing had to be made. Fortunately, the *ModelServices* provides *ParseKey*, an appropriate method to do so. Afterward, it is necessary to locate the correct mobile screens in the **ESpace**, using their respective keys (lines 6 and 7). It is important to remember that the *OutSystems Model* assumes a tree structure hence, certain elements contain collections of “**descendants**”. This is the case for **ESpaces**, which among others, assemble a set of mobile screens.

Subsequently, it was necessary to instantiate the actual connection between the concerning screens. The *NewConn* method using as arguments the concrete *NewConn* objects is responsible for gathering all the exit points of the origin screen and organizing them in links, buttons, or redirection nodes. This was achieved through the methods depicted in lines 3, 4, and 5 of listing 19. Once again, is important to mention, that

```

1  public void ConnectScreens()
2  {
3      IKey originScreenKey =
4          ↪ ModelServices.ParseKey(Connection.OriginScreenKey);
5      IKey destinationScreenKey =
6          ↪ ModelServices.ParseKey(Connection.DestinationScreenKey);
7
8      IMobileScreen originScreen =
9          ↪ ESpace.GetAllDescendantsOfType<IMobileScreen>().Single(s =>
10         ↪ s.ObjectKey.Equals(originScreenKey));
11     IMobileScreen destinationScreen =
12         ↪ ESpace.GetAllDescendantsOfType<IMobileScreen>().Single(s =>
13         ↪ s.ObjectKey.Equals(destinationScreenKey));
14
15     Connect(originScreen, destinationScreen, ModelServices, ESpace);
16 }

```

Listing 18: Create connection - *ConnectScreens* method

the *IButton*, *ILink*, and *IDestinationNode* stand as descendants of screen objects, similarly as described before with the relation between an *ESpace* and its screens. Therefore, the above-mentioned methods, retrieve all the exit points of a screen by iterating over the screen object descendants. The *linkKey* attribute present in the *Connection* object uniquely identifies the exit point associated with the connection to be established therefore, as seen in line 7, the challenge came from the location of the link key in the gathered exit points of the origin screen.

```

1  public void SetDestination(IMobileScreen destinationScreen,
2     ↪ IObjectSignature item, ScreenParameterEditor parameterEditor)
3  {
4     if (item is ILink link)
5     {
6         if (link.OnClick.Destination is IExternalSite)
7             ScreenParameterEditor.
8             ↪ ClearAllArguments(link.OnClick.Arguments);
9
10         link.OnClick.Destination = destinationScreen;
11
12         parameterEditor.
13         ↪ FillMandatoryInputArguments(link.OnClick.Arguments);

```

```

1  private void Connect(IMobileScreen originScreen, IMobileScreen
    ↪ destinationScreen, IModelServices modelServices, IESpace eSpace)
2  {
3      var links = OperationUtils.IdentifyExitPointLinks(originScreen);
4      var buttons =
    ↪ OperationUtils.IdentifyExitPointButtons(originScreen);
5      var redirectionNodes =
    ↪ OperationUtils.IdentifyExitPointNodes(originScreen);
6
7      IObjectSignature objSig = OperationUtils.
    ↪ FindLink(modelServices.ParseKey(Connection.LinkKey), links,
    ↪ buttons, redirectionNodes);
8
9      ScreenParameterEditor parameterEditor = new
    ↪ ScreenParameterEditor(new ExperienceBuilderLib.
    ↪ Domain.Services.OSModel.OSDataTypes(eSpace));
10
11     SetDestination(destinationScreen, objSig, parameterEditor);
12 }

```

Listing 19: Create connection - *Connect* method

```

11
12     ProjectionUtils.CreateMetadata(link,
    ↪ Consts.Operations.LinkType, Connection.LinkType.ToString());
13     ProjectionUtils.CreateMetadata(link,
    ↪ Consts.Operations.LinkOrder, Connection.Order.ToString());
14     ProjectionUtils.CreateMetadata(link,
    ↪ Consts.Operations.LinkLabel, Connection.Label);
15 }
16 else if (item is IButton button)
17 {
18     if (button.OnClick.Destination is IExternalSite)
19         ScreenParameterEditor.
    ↪ ClearAllArguments(button.OnClick.Arguments);
20
21     button.OnClick.Destination = destinationScreen;
22     parameterEditor.
    ↪ FillMandatoryInputArguments(button.OnClick.Arguments);
23

```

```

24     ProjectionUtils.
        ↪ CreateMetadata(button,Consts.Operations.LinkType,
        ↪ Connection.LinkType.ToString());
25     ProjectionUtils.
        ↪ CreateMetadata(button,Consts.Operations.LinkOrder,
        ↪ Connection.Order.ToString());
26     ProjectionUtils.
        ↪ CreateMetadata(button,Consts.Operations.LinkLabel,
        ↪ Connection.Label);
27     }
28     else
29     { var node = (IDestinationNode)item;
30
31         if (node.Destination is IExternalSite)
32             ScreenParameterEditor.ClearAllArguments(node.Arguments);
33
34         node.Destination = destinationScreen;
35         parameterEditor.FillMandatoryInputArguments(node.Arguments);
36
37         ProjectionUtils.CreateMetadata(node,
        ↪ Consts.Operations.LinkType,
        ↪ Connection.LinkType.ToString());
38         ProjectionUtils.CreateMetadata(node,
        ↪ Consts.Operations.LinkOrder, Connection.Order.ToString());
39         ProjectionUtils.CreateMetadata(node,
        ↪ Consts.Operations.LinkLabel, Connection.Label);
40     }
41 }

```

Listing 20: Create connection - *SetDestination* method

The *SetDestination* method (listing 20) uses the found link object (*IObjectSignature*) as well as the destination screen. The method starts by testing if the object is of type *ILink*, *IButton*, or *IDestinationNode*. Upon discovery, the process is very similar regardless of the actual object type. In the first moment, it is important to examine if the default destination of the link, button or node, is an external site, that being a URL to a location outside the application. In that case, it is necessary to clear the default arguments of the *OnClick* events. This was achieved through the provided methods of a class specialized in the editing of the screen parameters. This class was also responsible for filling the mandatory input arguments of the *OnClick* events. Some screen links are associated

with flags specifying a parameter is mandatory, and a boolean, text-based, or numeric-based for instance. Thus, in these cases, those parameters must be filled with default values such as "false", empty strings, or 0 respectively. All this is a responsibility of the `FillMandatoryInputArguments` method of line 10.

When it comes to the concrete connection definition, for links or buttons, it is necessary to initialize the `OnClick` event destination in order to direct the users to the desired destination screen, as seen in lines 8 and 21, respectively. The destination node does not provide a clickable interface thus, the destination setting occurs directly on the node object (line 35).

Finally, the creation operation, as mentioned before, is tied to the injection of metadata, as one can see in the listing with the `CreateMetadata` method of the `ProjectionUtils` class. Nonetheless, this relates to the [Forward Transformation](#) process and therefore will be discussed in the following section.

6.3.3.3 Change Propagation: Experience Builder - Service Studio

All the process depicted in the previous sections is triggered by the `BackwardTransformation` method and will culminate in the desired builder-Service Studio change propagation. This method, as we can see in listing 21, the method takes as parameters the original Experience Builder model (`originalEBModel`), the changed version (`changedEBModel`), and the path of the OutSystems application (`OAPPath`) to which the computed deltas shall be propagated. Firstly, the original builder model is initialized, and the delta computation occurs following what was described in the previous sections. Subsequently, one can observe the initialization of the `ModelServices` variable with an instance of the `IModelServices` interface, the Model API's entry point empowering its users with the capacity of creating and manipulating applications and **ESpaces**. In the following line, the interface is used to load the OutSystems application to memory.

To accomplish the desired transformation, as previously stated, all the `Run` methods of the captured change operations have to be executed. Finally, with the made modifications correctly propagated, the new state of the OutSystems application is saved.

6.4 Forward Transformation

Despite a large part of the Experience Builder model containing meaningful information regarding the interaction of the user with the [UI](#) and to the generation process of an [OAP](#), **this data does not transition** to the OutSystems model dealt in the Service Studio, nor should it be visible to the [IDE](#) users since it addresses internal behavior and is irrelevant to the Service Studio context. Thus, for the sake of enabling the reverse transformation, i.e. Service Studio-Experience Builder is important to note that some additional data had to be stored and consequently retrieved, to allow the reconstruction of the builder's model.

```
1 public void BackwardTransformation(string originaleBModel, string
2 changedEBModel, string OAPPath)
3 {
4     InitializeModel(originalEModel);
5     DeltaComputation(changedEBModel);
6
7     IModelServices modelServices =
8     OutSystems.ModelAPILoader.Loader.ModelServicesInstance;
9     IApplication application =
10         modelServices.LoadApplication(OAPPath);
11
12     foreach (Operation changeOp in ChangeOperations)
13         changeOp.Run(application, modelServices);
14
15     application.Save(OAPPath);
16 }
```

Listing 21: BackwardTransformation method

6.4.1 Metadata Injection

In order to retrieve the prototype-generated elements, the chosen strategy was to **piggy-back** all the necessary builder model data in the created elements upon the moment of their creation. Therefore, each prototype-created flow, connection, or menu item carried in metadata format the data required to fully regenerate the builder model.

In this regard, the following sections describe the implementation of the metadata injection mechanism, along with the reasoning behind the selection of the builder data fields that were injected.

6.4.1.1 AddFlow operation

In terms of the AddFlow operation, it was crucial to add to the created flow object data addressing the SourceModuleKey and the FlowType, as these fields are used and disregarded in the app generation process, but prevail relevant to the builder environment. The SourceModuleKey subsists as string responsible for identifying the OutSystems module template encompassing the flow, while the FlowType is used by the builder to organize the flows in categories addressing typical use-cases such as *Login* or *Onboarding*, for instance. Listing 22 depicts the method in control of adding the flow metadata to each created flow (in the listing referred as the flowsToMerge).

Furthermore, each created flow is composed of one or more screens also requiring added metadata. In this case, every screen object was linked with data referring to the

```

1 private void AddFlowMetadata(IEnumerable<IMobileFlow> flowsToMerge,
  ↪ IModelServices modelServices)
2 {
3     foreach(IMobileFlow flow in flowsToMerge)
4     {
5         ProjectionUtils.CreateMetadata(flow,
  ↪ Consts.Operations.SourceModuleMetadata,
  ↪ Flow.SourceModuleKey);
6         ProjectionUtils.CreateMetadata(flow,
  ↪ Consts.Operations.FlowTypeMetadata,
  ↪ Flow.FlowType.ToString());
7         AddScreenMetadata(flow.Nodes.OfType<IMobileScreen>(),
  ↪ modelServices);
8     }
9 }

```

Listing 22: Metadata Injection - *AddFlowMetadata* method snippet

```

1 private void AddScreenMetadata(IEnumerable<IMobileScreen> screens,
  ↪ IModelServices modelServices)
2 {
3     foreach (IMobileScreen scr in screens)
4     {
5         var screen = Flow.Screens.Single(s =>
  ↪ scr.ObjectKey.Equals(modelServices.ParseKey(s.ScreenKey)));
6
7         ProjectionUtils.CreateMetadata(scr,
  ↪ Consts.Operations.HasMenuMetadata,
  ↪ screen.HasMenu.ToString());
8         ProjectionUtils.CreateMetadata(scr,
  ↪ Consts.Operations.OrderInFlowMetadata,
  ↪ screen.OrderInFlow.ToString());
9
10    }
11 }

```

Listing 23: Metadata Injection - *AddScreenMetadata* method snippet

```

1  public void SetDestination(IMobileScreen destinationScreen,
   ↪  IObjectSignature item) {
2      ...
3
4      ProjectionUtils.CreateMetadata(link, Consts.Operations.LinkType,
   ↪  Connection.LinkType.ToString());
5      ProjectionUtils.CreateMetadata(link, Consts.Operations.LinkOrder,
   ↪  Connection.Order.ToString());
6      ProjectionUtils.CreateMetadata(link, Consts.Operations.LinkLabel,
   ↪  Connection.Label);
7
8      ...
9
10 }

```

Listing 24: Metadata Injection - *CreateConnection* method snippet

HasMenu boolean flag denoting if the screen is associated with a menu item and the OrderInFlow integer depicting the order of the screen in the flow. Similarly to the previous listing, listing 22 depicts the method in control of adding the metadata to each screen associated with the created flow.

6.4.1.2 *AddConnection* operation

Let's consider the implementation regarding the creation of a new connection. As seen previously, in listing 20 after setting the destination, several connection attributes are passed in the form of metadata, with the call of the CreateMetadata method. These attributes are consumed in the application generation process by the Experience Builder infrastructure and do not travel to the OutSystems Model. To overcome this hurdle, the implemented method makes direct use of the functionality provided by the ModelAPI of associating metadata with new or existing model elements. As evidenced by listing 24, depicting a portion of the SetDestination method, it was possible to associate the link object with metadata referring to the LinkType responsible for indicating to the Experience Builder if the exit point of the link is associated with a login, or a back button, for instance, and the LinkLabel referring to a comment associated with the connection. All the injected attributes being distinguished by their pre-defined name, and the "Editor-Prototype" tag (represented by the constant Consts.Operations.MetadataManager), as seen in listing 25, expanding the CreateMetadata method.

6.4.1.3 *AddMenuItem* operation

```
1 public static ITextMetadata CreateMetadata(IObjectSignature obj, string
  ↳ metadataName, string metadataValue)
2 {
3     ...
4
5     if (obj.GetType().Name.Equals(Constants.Operations.TypeOfElementLink))
6         metadata = ((ILink)obj).CreateMetadata<ITextMetadata>(name:
  ↳ metadataName, managedBy:
  ↳ Constants.Operations.MetadataManager);
7
8     ...
9     metadata.Value = metadataValue;
10    metadata.Hidden = false;
11    return metadata;
12 }
```

Listing 25: Metadata Injection - *CreateMetadata* method snippet

```
1 public void AddItemToMenuBottomMenu()
2 {
3     ...
4     ProjectionUtils.CreateMetadata(bottomBarItem,
  ↳ Constants.Operations.SSMenuItemId, MenuItem.SSMenuItemId);
5     ProjectionUtils.CreateMetadata(bottomBarItem,
  ↳ Constants.Operations.MenuItemOrderMetadata,
  ↳ container.Widgets.Count().ToString());
6     ...
7 }
```

Listing 26: Metadata Injection - *CreateMenuItem* method snippet

In regards to the creation of a new menu item, the injected metadata referred to the `SSMenuItemId`, the data field responsible for unequivocally identifying a menu item in the builder environment, and the `Order`, pertaining to the order of the considered item in the menu itself. Listing 26 illustrates a code snippet belonging to the `AddItemToMenuBottomMenu` method involved in the builder data injection.

As seen in line 5 in listing 26, the `Order` field had to be initialized in run time, taking into account the number of existing application menu items (in the `OutSystems` model environment corresponding to `widgets`). This was necessary due to the possibility of the prototype handling an outdated view of the application menu. I.e., the builder model used by the prototype is obtained by a **projection** that exclusively contemplates prototype-generated elements therefore, the application menu might contain items absent in the projection model hence, to adequately define the order of the menu item being added, the program must assess the correct number of `widgets` in run time.

6.4.2 Projection

The concrete Projection occurs with the retrieval of all the necessary information now existing in the `OutSystems` model back to the Experience Builder model. This is initiated in the `GetProjection` method (listing 27) which takes as arguments the main `ESpace` of the application, and an instance of the `ModelAPI` services.

A new builder model object is instantiated with the purpose of storing the extracted elements. Then, the method takes care of the flows (and screens) and the connections. For consistency purposes, let's consider the implementation of the Projection process concerning the connections. As illustrated in line 10, as the program iterates over the *Mobile Flows* of the application, additional processing is done over the connections associated with these flows. It is relevant to mention that, since the developed solution only minds Experience Builder elements, created in the builder itself, and these, as seen in the previous section carry added **metadata**, it was relatively easy to identify the `OutSystems` model elements that needed to be retrieved (line 12). This testing process occurs in the `GetConnectionsFromLinks` method for instance, a method belonging to the chain of calls initiated in the `GetEBConnectionsFromSSMobileScreens` and depicted in listing 28.

Considering that when creating a connection, metadata is injected in the `ILink` object of the destination screen, it is not surprising the program starts by testing if the screen links have metadata. This testing process is encompassed in `HasEditorMetadata` (line 9) and works in a complementary manner with the previously seen `CreateMetadata` method. When the condition is true, the connection was created in the builder therefore, the program can start retrieving the necessary attribute values. This is done through the `GetMetadata` method, which is responsible for returning the metadata value associated with a given tag. In the case of a connection, the `LinkType`, `Label`, `Order` and the `DestinationScreenKey` are restored using the metadata process (lines 11, 12, 13 and 14) while all the other attributes are obtained by accessing available object properties (such

```

1  private ExperienceBuilderModel GetProjection(IESpace eSpace,
   ↪  IModelServices modelServices)
2  {
3      ExperienceBuilderModel model = new ExperienceBuilderModel();
4
5      IEnumerable<IMobileFlow> mobileFlows = eSpace.MobileFlows;
6
7      List<Connection> connections = new List<Connection>();
8
9      foreach (IMobileFlow flow in mobileFlows)
10     {
11         if (ProjectionUtils.HasEditorMetadata(flow,
   ↪     Consts.Operations.MetadataManager))
12
13             model.AddFlowAndScreens(ProjectionUtils.
   ↪     SSFlowToEBFlow(flow));
14
15             connections.AddRange(ProjectionUtils.
16             GetEBConnectionsFromSSMobileScreens(flow.Nodes.OfType
17             <IMobileScreen>()));
18     }
19
20     model.AddConnections(connections);
21
22     IMobileBlock menuBlock = OperationUtils.GetBottomMenuBlock(eSpace,
   ↪     modelServices);
23     IEnumerable<IMobileBlockInstanceWidget> menuItemsInit =
   ↪     menuBlock.GetAllDescendantsOfType<IMobileBlockInstanceWidget>();
24     model.SetMenuType(Consts.Operations.BottomBarTypeId);
25
26     foreach (var menuItem in menuItemsInit.ToList())
27         if (ProjectionUtils.HasEditorMetadata(menuItem,
   ↪     Consts.Operations.MetadataManager))
28             model.AddMenuItem(
29             ProjectionUtils.SSMMenuItemToEBMenuItem(menuItem));
30
31     return model;
32 }

```

Listing 27: Projection - *Projection* method

```
1 private static List<Connection> GetConnectionsFromLinks(IMobileScreen
2     ↪ screen)
3 {
4     var links =
5     ↪ Operations.OperationUtils.IdentifyExitPointLinks(screen);
6
7     var result = new List<Connection>();
8
9     foreach (ILink link in links)
10    {
11        if (HasEditorMetadata(link, Consts.Operations.MetadataManager))
12        {
13            var linkType = int.Parse(GetMetadata(link,
14                ↪ Consts.Operations.LinkType,
15                ↪ Consts.Operations.MetadataManager).Value);
16            var label = GetMetadata(link, Consts.Operations.LinkLabel,
17                ↪ Consts.Operations.MetadataManager).Value;
18            var order = int.Parse(GetMetadata(link,
19                ↪ Consts.Operations.LinkOrder,
20                ↪ Consts.Operations.MetadataManager).Value);
21            string destinationScreenKey =
22            ↪ link.OnClick.Destination.ObjectKey.ToString();
23
24            result.Add(new Connection
25            {
26                LinkKey = link.ObjectKey.ToString(),
27                LinkType = linkType,
28                Label = label,
29                OriginScreenKey = screen.ObjectKey.ToString(),
30                OriginScreen = SSScreenToEBScreen(screen),
31                Order = order,
32                DestinationScreenKey = destinationScreenKey,
33            }); ;
34        }
35    }
36    return result;
37 }
```

Listing 28: Projection - *GetConnectionsFromLinks* method


```

1  public ExperienceBuilderModel ForwardTransformation(string OAPPath,
   ↪ string EBModelPath)
2  {
3      IModelServices modelServices =
   ↪ OutSystems.ModelAPILoader.Loader.ModelServicesInstance;
4
5      IApplication application = modelServices.LoadApplication(OAPPath);
6
7      var eSpace = (IESpace)application.Modules.First(m =>
   ↪ m.Key.Equals(application.FrontOfficeEspaceKey)).Load();
8
9      ExperienceBuilderModel projection = GetProjection(eSpace,
   ↪ modelServices);
10
11     SaveModel(EBModelPath, projection);
12
13     return projection;
14 }

```

Listing 29: Projection - *ForwardTransformation* method

as the `LinkKey` contained in the `Link` object or the `OriginScreenKey`, included in the `IMobileScreen`).

6.4.2.1 Change Propagation: Service Studio - Experience Builder

All the process depicted in the previous sections is triggered by the *ForwardTransformation* method and will culminate in the desired Service Studio to builder change propagation.

This method is analogous to the previously discussed *BackwardTransformation method*, and, as we can see in listing 29, it starts by considering the path of the OutSystems application (`OAPPath`) and the path to Experience Builder model (`EBModelPath`) representing the current stage of the application. Once again, the program starts by initializing the `ModelServices` necessary to load the OutSystems application to memory. Thereafter, only with the main `ESpace` of the app located, the program is capable of proceeding with the described `GetProjection` method, which will return a complete and updated Experience Builder model. Subsequently, the model is serialized to the desired location and the obtained projection is returned.

RESULTS AND EVALUATION

The goal of this dissertation was, first and foremost, to investigate the possibility of a bidirectional flow of interaction between the builders and Service Studio. Our investigation culminated in a prototype based on the Experience Builder with added functionality and modifications targeting pre-existing features.

This chapter details more extensively all the features of the developed prototype and compares them with the current behavior of the Experience Builder. In addition, this chapter also elaborates on the performed evaluation procedures and the prototype's limitations.

7.1 Results

The final prototype resulted in an augmented version of the Experience Builder capable of supporting continuous collaborative development of an application together with Service Studio. Through the analysis of the end result, it is clear **there was an extension** to the currently offered builder's **functionality**, with each prototype operation modifying the OutSystems model in a **precise** and **isolated** manner, as opposed to fully regenerating a new application and disregarding any change made in Service Studio. In addition, the developed algorithm implemented in the prototype proves with a high degree of certainty, the Experience Builder can be effectively modified to be used as an editor.

Nonetheless, when comparing the attained functionality of the prototype with the Experience Builder's, it is relevant to keep in mind every prototype's implemented operation considers a bidirectional application development scenario, while the same can not be said in regards to the builder. The **builder** is associated with the Service Studio in a **unidirectional** manner therefore, an application is created in the Experience Builder, it is published, and if the developer chooses to, he can furtherly customize it using the IDE. If the builder is used again in this scenario, its user will have no update view of the state of the application, and if he publishes the app once again, any Service Studio will be disregarded. However, any application change made employing the builder's functionalities effectively results in an OutSystems model with those exact changes, despite being

detrimental to the previous state of the app. Whereas, the **prototype** despite targeting a specific scenario (where an application has been published using the builder) guarantees **full interoperability with Service Studio**.

On a more detailed level, all the operations enabled by the prototype differ from the corresponding operations in the builder. Due to the **Single-Shot** mode of operation, the Experience Builder does only implement creation operations hence, any element deletion or update is inconceivable. Whereas in the prototype environment, any deletion or update successfully targeted the concerned element(s) hence achieving the **desired change propagation**.

Regarding the creation operations, much of the prototype code was based on the Experience Builder with the added difference of, in this case, providing contingencies for future projections with the injection of metadata, as detailed in the following sections.

7.1.1 Add Flow operation

7.1.1.1 Experience Builder

Using the builder, the developer can easily add a new flow by clicking on the “Add Flows” button highlighted in red in [Figure 7.1](#). This will open a dialogue window displaying all the available flows, organized in typical application development use-cases such as “Authentication” or “Banking” ([Figure 7.24](#)).

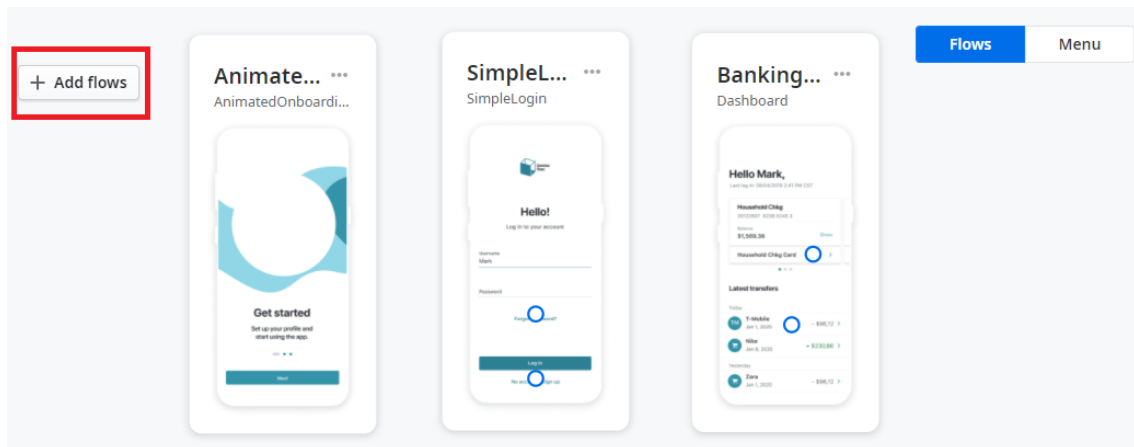


Figure 7.1: Experience Builder Interface - Add Flows button

7.1.1.2 Prototype

The final prototype ran locally and did not provide a **GUI** hence multiple changes had to be made to the common behavior of the Experience Builder, in order to accommodate this operation in the console application interface. One of those modifications was to load to memory all the possible flows the developers could add to their applications.

Therefore, to add a new flow to the application the prototype user had to consult the possible flows through the `avai_flows` command, and enter the flow's name, as depicted in Figure 7.2:

```
>> avai_flows
Flow Name: LoginAndPasscode;
Name: LoginSetPasscode; Description: Log in to the app to set a passcode and Face ID authentication.
Name: SetPasscode; Description: Set up a 4-digit passcode.
Name: SetFaceID; Description: Set up Face ID, to authenticate with face recognition.
Name: LoginPasscodeOK; Description: Success message after setting up a passcode and Face ID.

Flow Name: PasscodeCheck;
Name: PasscodeCheck; Description: Once logged in, verify your identity with a passcode.

Flow Name: AccountDetails;
Name: AccountDetails; Description: Detailed personal, account and security information.

>> add_flow PasscodeCheck
Flow was added with success.
```

Figure 7.2: Prototype Console - Add Flow use-case

In terms of the **metadata injection** regarding this operation, the process was deemed **successful** since each created flow object effectively carried data addressing the “Source-ModuleKey” pertaining to the flow template, along with the “FlowType” used by the builder to organize the flows in categories. Figure 7.3 shows in Service Studio an example of a created flow with the additional metadata fields, visible only for demonstration purposes. Furthermore, each screen object of the added flow was also successfully linked with metadata such as the “HasMenu” boolean flag denoting if the current screen is associated with a menu item, and the “OrderInFlow” integer depicting the order of the screen in the encompassing flow. This metadata is illustrated in Figure 7.4 and once again is only visible to the user for demonstration purposes.

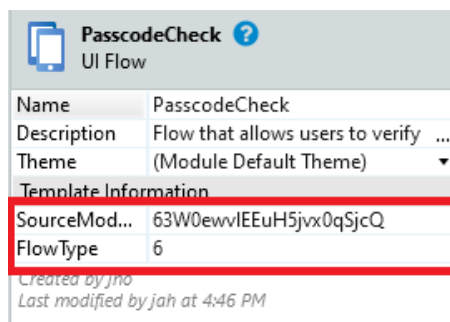


Figure 7.3: Service Studio - Added Flow Metadata

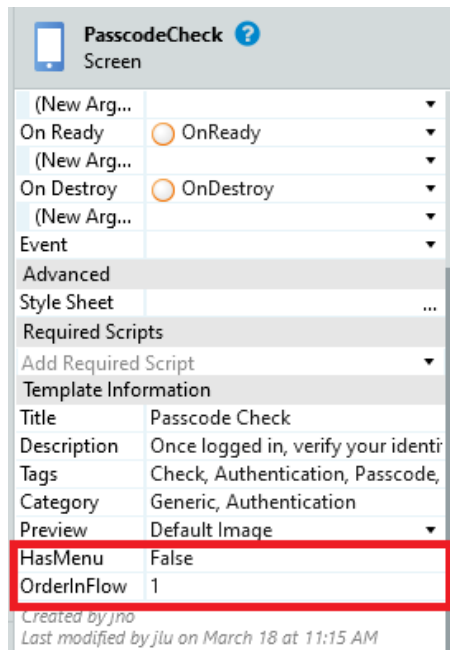


Figure 7.4: Service Studio - Added Screen Metadata

7.1.2 Add Connection operation

7.1.2.1 Experience Builder

Using the builder, the developer can easily create a new [connection](#), linking two screens to each other. This is possible by using a screen's exit point marked in the user interface by a blue circle hovering a button or a link, for instance. The developer can click or drag a connecting line from a screen to another flow, as seen in [Figure 7.5](#).

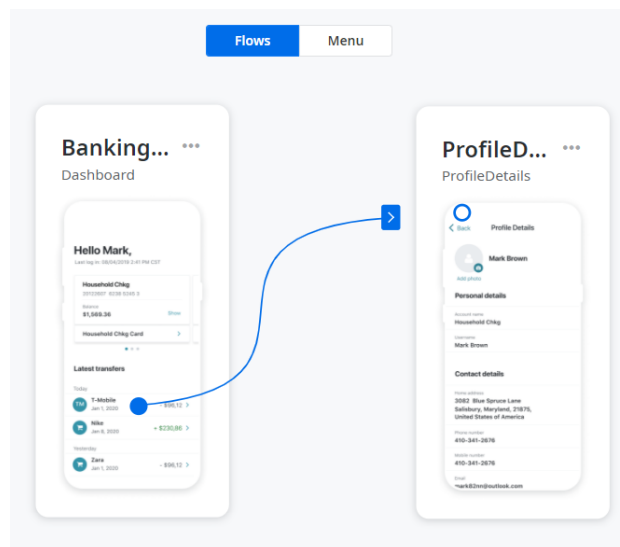


Figure 7.5: Experience Builder Interface - Add Connection

7.1.2.2 Prototype

Since the builder’s user experience cannot be replicated in the console prototype, every screen’s possible exit point had to be previously loaded to memory for the sake of enabling the creation of new connections. Thereby, in the interest of setting a new link between screen X and Y, the prototype user had to consult the available exit points of screen X through the *avai_screen_exit_points* command, and enter the screen Y’s name in the *add_flow* command, as denoted in the following example:

```
>> avai_screen_exit_points SimpleLogin
Screen Name: SimpleLogin;
Label: Login;
Link Key: iX3VD_sozk0tr12j26mTlA;
Label: Sign Up;
Link Key: esIhm7dcDUq0jxo25gqlnw;

>> add_connection iX3VD_sozk0tr12j26mTlA AccountDetails
Connection was added with success.
```

Figure 7.6: Prototype Console - Add Connection

The necessary builder model data **transitioned successfully** to the OutSystems model through the metadata injection process. The added data regarded “LinkType”, an integer which identifies the link as a back button or a login button, for instance, and the “LinkLabel”, a comment associated with the connection. [Figure 7.7](#) shows in Service Studio an example of a created connection, linking an existing screen with the “AccountDetails” screen, with the added metadata, visible only for demonstration purposes.

AccountDetails\AccountDetails ?	
Destination	AccountDetails\AccountDe
(New Argu...	
Transition	(Module Transition)
Template Information	
LinkType	2
Label	Login

Figure 7.7: Service Studio - Added Connection Metadata

7.1.3 Add Menu Item operation

7.1.3.1 Experience Builder

The Experience Builder developer can seamlessly add a new item to the existing menu. This is possible through the “Menu” tab, where the click on the plus button depicted

in the [Figure 7.8](#). Developers can then select from a wide array of icons with associated names as seen in [Figure 7.9](#).

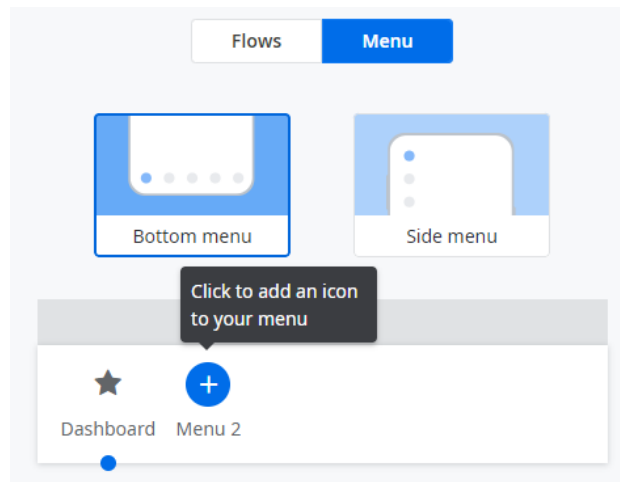


Figure 7.8: Experience Builder Interface - Add Menu Item dialogue window

Add icon

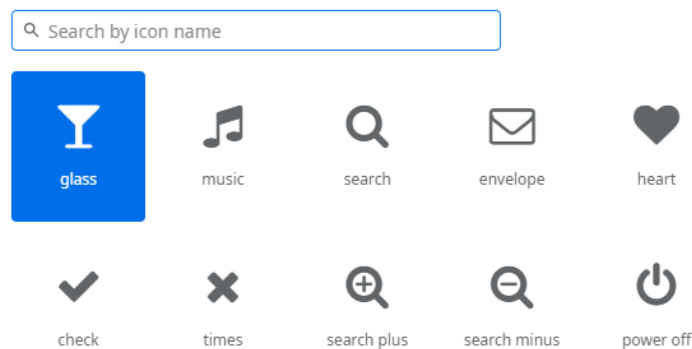


Figure 7.9: Experience Builder Interface - Add Menu Item

7.1.3.2 Prototype

The prototype provided the *avai_icons* command printed the icon names and respective caption of all the available menu items. The user could then add the chosen item by providing its name to the *add_menu_item*, along with the screen name where the user will be directed to if he uses the button. This process is represented in [Figure 7.10](#).


```

>> avai_icons
Caption: Dashboard;
Icon Name: home;

Caption: Transfers;
Icon Name: exchange;

Caption: Bills;
Icon Name: money;

Caption: Deposit;
Icon Name: btc;

Caption: Settings;
Icon Name: cogs;

>> add_menu_item money BankingDashboard
Menu item was added with success.

>>

```

Figure 7.10: Prototype Console - Add Menu Item

The metadata was **successfully injected** into each created menu item object. The added data regarded the menu item identifier “SSMenuItemId”, as well as a unique key based on that same identifier and noted as “MenuItemKeyMetadata”, and “MenuItemOrder” depicting the order of the item in the menu. [Figure 7.11](#) depicts the Service Studio view of an added menu item with the injected metadata visible for demonstration purposes.

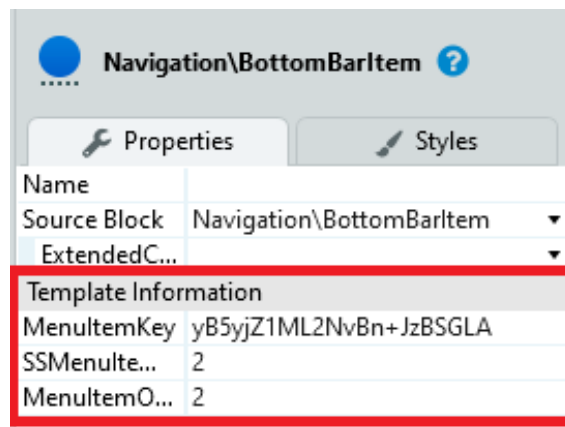


Figure 7.11: Service Studio - Added Menu Item Metadata

7.1.4 Backward and Forward Transformations

Along with the described operations, the final prototype also provided its users with the capability of manually triggering the change propagation both in the Experience Builder-Service Studio direction (with the backward transformation) and in Service Studio-Experience Builder direction (with the forward transformation), as discussed in the following sections.

7.1.5 Backward Transformation operation

7.1.5.1 Experience Builder

If we consider the **backward transformation** operation as the conversion of the Experience Builder model to the OutSystems model, the builder natively supports this functionality through the “Publish” button (Figure 7.12). However, as described, the current model transformation displays shortcomings that damage the continuous collaboration in the [OutSystems Platform](#).



Figure 7.12: Experience Builder Interface - Publish button

7.1.5.2 Prototype

The final prototype enabled users with the possibility of publishing any builder model change in a controlled, sound, and isolated manner. The *back_transf* command triggered the transformation and required users to input several arguments: the path to the original builder model, the path to the altered model, and the path to the OutSystems Application to which the deltas would be propagated. The following figure illustrates a scenario where after saving the changes made to the “TinyApp” builder model, the **backward transformation** is evoked, with the command considering the original builder model (“TinyApp.json”), the new version with the user’s modifications (“NewTinyApp.json”) and the OutSystems Application (“TinyApp.oap”).

```
>> save NewTinyApp.json
Saving model...
Model was saved.

>> load NewTinyApp.json
Model ready to be used.

>> back_transf TinyApp.json NewTinyApp.json TinyApp.oap
Changes were applied!

>>
```

Figure 7.13: Prototype Console - Backward Transformation

The **delta computation** phase was **implemented successfully**, with the program correctly analyzing, comparing two builder models, and finally yielding a **changelog** containing all the change operations that had to be propagated to the OutSystems model to restore application consistency among the builder and the OutSystems models. The following listing denotes an example of a changelog, in this case, comprised of a single flow creation operation.

```
[
  {
    "operationObject": {
      "flowName": "AccountDetails",
      "flowKey": "40649b12-a3aa-4e6d-90ff-b656cd7a6626",
      "flowType": 0,
      "sourceModuleKey": "paYD5xm6q0KzdPLVwEXr8Q",
      "screens": [
        {
          "name": "AccountDetails",
          "description": "Detailed personal, account and security
↔ information.",
          "comment": "",
          "screenKey": "SpIzWrAjcUGp3wmfk16H6Q",
          "orderInFlow": 1,
          "hasMenu": true,
          "imageUrl": ""
        }
      ]
    },
    "operationType": "CREATE"
  }
]
```

Listing 30: Change Log Example

All the chosen app customizations enabled by the Experience Builder were **successfully** mapped to operation objects, whose execution led to the desired change propagation to the OutSystems Model. In addition, it was possible to verify the propagation occurred without disregarding any pre-existing application element present in the OutSystems model.

7.1.6 Forward Transformation operation

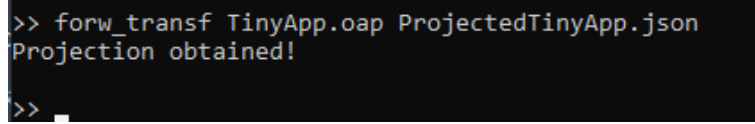
7.1.6.1 Experience Builder

Due to the current mode of operation employed by the builder, it is impossible to continue to develop an application after its customization in Service Studio. This is the result of the unidirectionality of application development regarding the use of the Experience Builder together with the platform's IDE. Hence, the **forward transformation**, which concisely represents the capacity of retrieving the builder model from the OutSystems model is not

available in the Experience Builder.

7.1.6.2 Prototype

To remedy this shortcoming, the final prototype empowered its users with the *forw_transf* command, responsible for providing an updated view of the application in the builder environment. The command required users to input the path to the OutSystems application from which the builder model will be extracted and the path to the file involved in storing the extracted model. [Figure 7.14](#) illustrates a scenario where the forward transformation is considering “TinyApp.oap” application and storing the resulting builder model into the “ProjectedTinyApp.json” file.



```
>> forw_transf TinyApp.oap ProjectedTinyApp.json
Projection obtained!
>>
```

Figure 7.14: Prototype Console - Forward Transformation

This achievement was made possible through the metadata injection process, which successively added all the necessary builder model data in the OutSystems model. The execution of the projection process naturally also weighed greatly in the success of the forward transformation.

The **desired outcome** was **reached**, with the projection yielding the current state of the application, restoring a complete builder’s model composed of all the necessary data regarding flows, screens, connections, and menu items.

7.2 Evaluation

7.2.1 Validation

The working prototype was not yet integrated into the Experience Builder development pipeline, since the main purpose of the development and implementation stages was to assess the viability of a bidirectional and continuous development flow. Therefore, it was not feasible to use the prototype for user testing. However, it was already possible to predict with a high degree of certainty that the achieved augmented interoperability **will not impact user experience**. Contrary to current behavior, the builder’s homepage will display all applications present in the user’s environment, with their state reflecting eventual changes made with Service Studio.

The success of our approach relies on machine injected information that is added to the OutSystems model in order to allow future restorations of the Experience Builder model hence, even though this dissertation does not provide a formal proof of soundness, a **correct implementation of the change operations** does, in principle, guarantee a

closed model evolution. This can be assured in practice by testing a number of representative scenarios in the platform.

Regarding usability tests, the prototype development was anchored in a well-defined and complete set of simulated use-cases that not only comprehend the entire universe of conceivable interactions between the tools but also lay the foundation for the real scenario test cases to take place in future stages. The simulated scenarios test the interoperability between Experience Builder and Service Studio during the development of an application. These scenarios were divided into three categories validating the solution's behavior in development workflows where an application is created in the builder, subsequently altered in the IDE, and it is later changed in the builder again.

7.2.1.1 Category 1 - Changes that do not impact Experience Builder-related elements

The developed solution assures that any creation, modification, or deletion of an element not represented in the Experience Builder model (which is e.g. the case of *Entities*) is preserved throughout the application development.

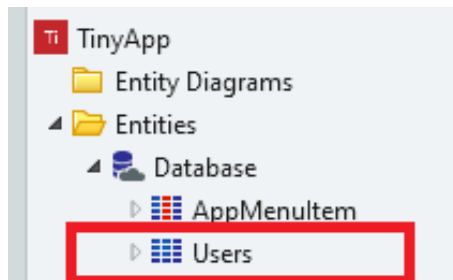
This is a direct consequence of the **partiality property** associated with the **projection** function, as well as due to the isolation of the consistency restoring operations. The projection is only sensitive to objects created in Experience Builder hence, it will only retrieve the data necessary to reconstruct flows, screens, connections and menu item data present in the builder's model while ignoring elements such as *Entities*. When it comes to the operations, these were deliberately designed to **exclusively** capture changes made to the Experience Builder model elements and their execution only impacts those same elements in the Service Studio side.

As an example of the behavior of the developed prototype in the use-cases encompassed in this category, one can consider the following:

Ann loads the **TinyApp** model in the prototype and publishes it, hence producing a fully-functional OutSystems application. In the matter of the builder's core model elements, the produced app, as seen before is composed of three flows, two connections and its menu contains one single item (see *TinyApp* subsection). Since all these elements are created using the prototype, adequate metadata is injected to enable a restoration of the updated application model on the builder side.

In the Service Studio, Bob opens the produced application and starts customizing the data layer. More specifically, he adds a new entity (*Users*) to the existing database, addressing the future users of the application (figure 7.15).

Ann unhappy with the name given to the onboarding flow, uses the prototype to change it, nevertheless, Bob has changed the application in the IDE which requires a **forward transformation** producing a model noted as *ProjectedTinyApp*. However, Bob's changes did not affect a flow, a screen, a connection, or a menu item hereby the projected builder model will be the same as the original *TinyApp* model used by Ann, who won't

Figure 7.15: Added *Users* entity

```

chg_menu_item_ord <item id> <new order> - Changes menu item order
chg_menu_type <item id> <new type> - Changes menu item order
avai_icons - Get all possible icon options
quit - Quit application

>> forw_tranf TinyApp.oap ProjectedTinyApp.json
Projection obtained!

>> load ProjectedTinyApp.json
Model ready to be used.

>> chg_flow_name AnimatedOnboardingOption1 Onboarding
Flow was changed with success.

>> add_flow AccountDetails
Flow was added with success.

>> back_transf ProjectedTinyApp.json TinyApp.json TinyApp.oap
Changes were applied!

>>

```

Figure 7.16: Category 1 - Ann’s prototype commands

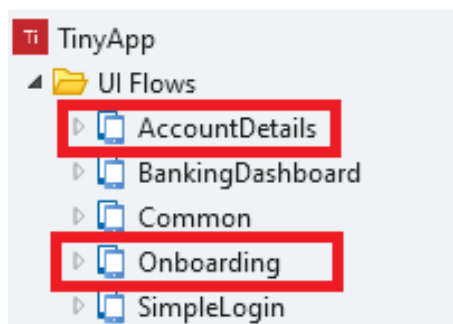


Figure 7.17: Service Studio - Backward Transformation Result

and shouldn’t detect any change made to the application. She will proceed with the desired customization of the **app**, by **changing the name** of the onboarding flow from “AnimatedOnboardingOption1” to “Onboarding”. This is achieved using the *chg_flow_name* command. In addition, Ann **adds** a new “AccountDetails” **flow** through the *add_flow* command (as seen in figure 7.16).

The **backward transformation** occurs: the new builder model is compared with the

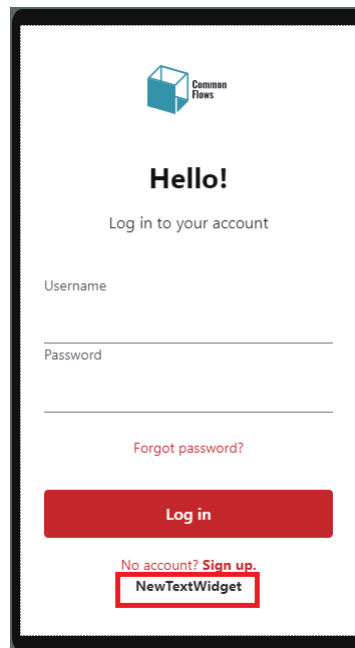


Figure 7.18: Service Studio - Bob's added text widget

previous one in the **delta computation** process leading to the production of the concerning changelog; The deltas are translated and applied to the OutSystems model, propagating Ann's changes. Since the change propagation exclusively impacts the elements that were actually changed when Bob opens the application in the **IDE**, his new entity will still be preserved.

7.2.1.2 Category 2: Changes that impact Experience Builder-related elements, despite these not being displayed by the builder

This category comprises all the modifications to flows, screens, connections and menu items, at a level of detail not available in Experience Builder. In the Experience Builder context, the screens that compose a flow are predetermined in terms of appearance, order, number. Thus, internal alterations to these elements, such as new widgets in a screen, are not visible in the builder's **UI**. However, this does not come as a problem to the developed solution since it adopts an "out of sight, out of mind" strategy: the projection retrieves only the builder's related objects with no extra data other than the one addressed in the builder's meta-model, and the change operations only affect precise components of the application, such as the name of a screen, hence preserving alterations made to any other part of the elements.

As an example consider that Ann once again publishes the TinyApp without any added change. Bob, using the Service Studio, opens the "LoginAndPasscode" flow and starts editing its screen. He adds a new text widget as seen in figure 7.21 highlighted in red.

```

    chg_menu_type <item id> <new type> - Changes menu item order
    avail_icons - Get all possible icon options
    quit - Quit application

>> forw_tranf TinyApp.oap ProjectedTinyApp.json
Projection obtained!

>> load ProjectedTinyApp.json
Model ready to be used.

>> add_flow AccountDetails
Flow was added with success.

>> add_connection iX3VD_sozkOtr12j26mTlA AccountDetails
Connection was changed with success.

>> back_tranf TinyApp.json ProjectedTinyApp.json TinyApp.oap
Changes were applied!

>>

```

Figure 7.19: Category 2 - Ann’s prototype commands

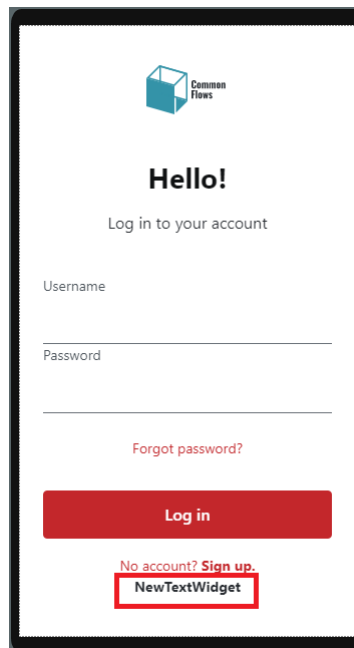


Figure 7.20: Service Studio - Bob’s added text widget (after Ann’s changes)

Ann using the builder wants to add a new flow and connect it to the “LoginAndPasscode” screen therefore, the forward transformation occurs, and as expected the retrieved builder model is no different from the original `TinyApp` since Bob’s modification is not captured in the developed prototype, due to the described projection behavior.

Nevertheless, Ann can subsequently customize the application by adding the desired flow with the `add_flow` command and connecting the “LoginAndPasscode” screen to it (via the existing login button’s link key “`iX3VD_sozkOtr12j26mTlA`”) with the `add_connection`. As mentioned, these operations execute in an isolated manner, affecting only the involved elements thus, the previously **created widget is preserved**.


```

    chg_menu_type <item id> <new type> - Changes menu item order
    avai_icons - Get all possible icon options
    quit - Quit application

>> forw_tranf TinyApp.oap ProjectedTinyApp.json
Projection obtained!

>> load ProjectedTinyApp.json
Model ready to be used.

>> rem_flow BankingDashboard
Flow was removed with success.

>> add_flow ManageCards
Flow was added with success.

>> add_connection iX3VD_sozk0tr12j26mTlA CardsDashboard
Connection was changed with success.

>> back_transf TinyApp.json ProjectedTinyApp.json TinyApp.oap
Changes were applied!

```

Figure 7.21: Category 3 - Ann's prototype commands

7.2.1.3 Category 3: Changes that impact Experience Builder-related elements that are displayed by the builder

This category comprises all the modifications possible in both the builder and in Service Studio, and corresponds to the more relevant category since it is directly correlated with the envisioned continuous collaboration between the Experience Builder and the Service Studio. As described, this subset of use-cases was directly tackled by using a sound projection function and controlled propagation of changes. As a representative example of the solution's good behavior in this category of operations lets consider the following scenario:

Ann publishes the [TinyApp](#) without further changing it. Bob, using Service Studio, opens the produced application and removes the connection linking the "LoginAndPasscode" flow to the "BankingDashboard".

Ann equipped with the Experience Builder wants to add a new flow to the application hence, a forward transformation occurs, and as expected the retrieved builder model depicts a missing connection between the login screen and the previous dashboard. With these new changes in mind, Ann deletes the "BankingDashboard", adds the "ManageCards" flow, and connects the "CardsDashboard" screen of the added flow to the existing login screen.

The changes are applied to the OutSystems model through the backward transformation, guaranteeing Bob will **have an updated view** of the working application in future editions with the Service Studio, as seen in figure 7.22 depicting the updated flows of the application.

In summary, and using [Figure 7.23](#) as a guide, we can validate the developed algorithm by considering $OutSys_B$, an OutSystems model resulting from a change $c1$ in Service Studio, model $ExpB_B = proj(OutSys_B)$, the result of projecting that model, and a change $c2$ in Experience Builder that results $ExpB_C$. This change is propagated to the OutSystems

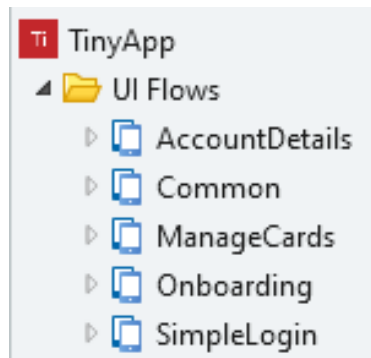


Figure 7.22: Service Studio - Application flows after Ann’s changes

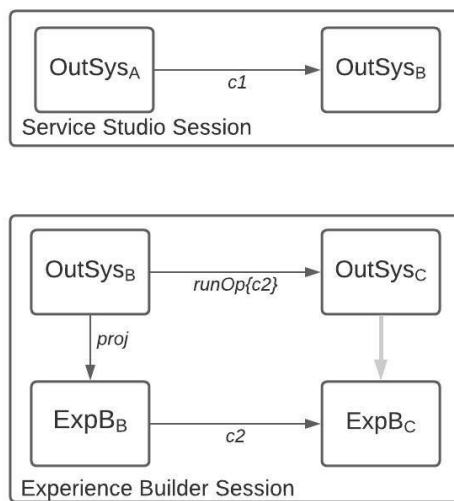


Figure 7.23: Solution’s behavior

model by applying $c2$ to the previous model, i.e.

$$OutSys_C = runOp(OutSys_B, dif(ExpB_C, ExpB_B))$$

This ensures that the effects of $c1$ are preserved. Note also that $ExpB_C = proj(OutSys_C)$, which corresponds to the grey arrow in Figure 7.23.

7.3 Unattained scenarios

It is important to note that two of the Experience Builder functionalities were not addressed by the prototype. The first one has to do with the “Empty Screen” flow. The developer can choose to add an ‘Empty Screen’ flow (Figure 7.24), to which he can later add a comment for reference and an image serving as a preview for the screen.

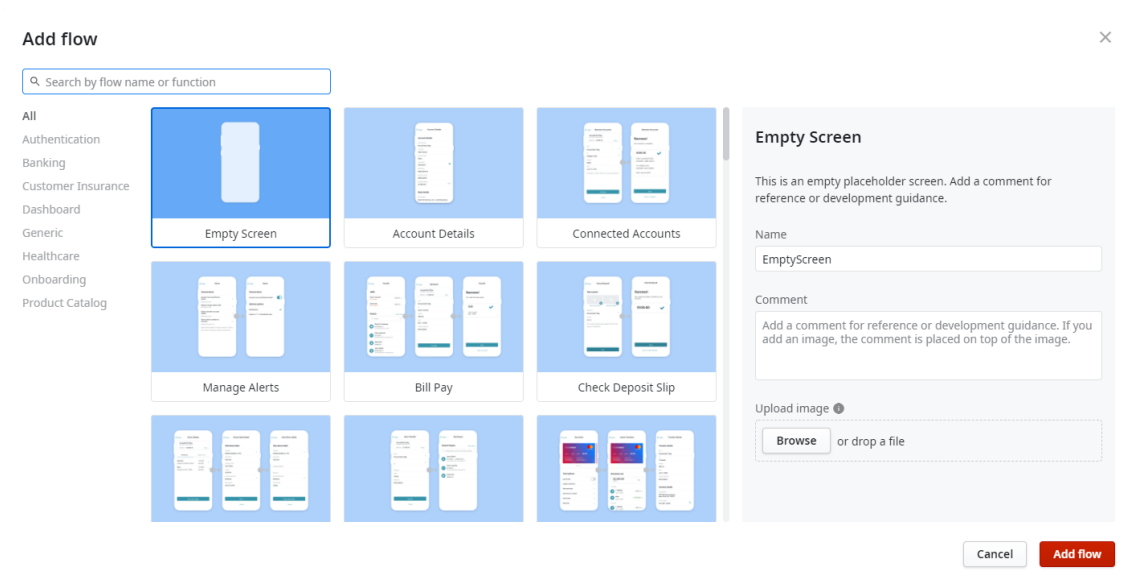


Figure 7.24: Experience Builder Interface - Add Flows dialogue window with Empty Screen selected

This functionality addresses scenarios where users want to integrate pre-developed screens with the application they are working in the builder. Due to its very particular use in addition to not pertaining to the main scope of the dissertation subject, the prototype did not implement this functionality.

Moreover, the builder supports applications with a side menu (Figure 7.25) or with a bottom menu. However, considering this dissertation's main objective fell into a proof-of-concept regarding the bidirectionality of model transformations among the builder and the Service Studio, it was deemed sufficient to extend the current functionality addressing only the bottom menus.

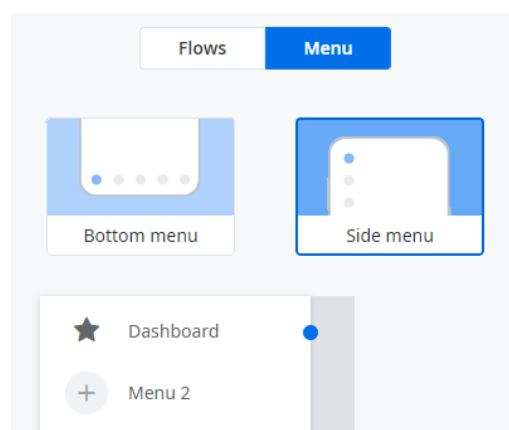


Figure 7.25: Experience Builder Interface - Side Menu

7.3.1 Limitations

Currently, our solution exists in the shape of a working prototype from which we identified a few limitations. The interoperability across the discussed development tools demands scenarios where multiple interconnected models are changed simultaneously. However, this has yet to be addressed in our solution. For instance, while the projection function is being calculated, a Service Studio developer might change a flow name, causing the projection to generate an outdated Experience Builder model. This parallel process prompts the necessity of implementing consistency-maintenance mechanisms capable of supporting concurrent updates, which require further user testing for better assessment. However, it is worth noting that Service Studio has native merge mechanisms that can be used in the resolution of this type of problem.

Apart from concurrency issues, the projection function is also unable to recognize Experience Builder-related elements that were created in Service Studio. As described, our solution addresses continuous development scenarios, all starting from an app generation via the Experience Builder. Here, as new elements are created, metadata is added to facilitate future projections. When it comes to flows, screens, or connections generated in the IDE, the necessary metadata is not available, thereby a viable alternative is currently being studied.

Lastly, it should be mentioned the lack of versatility regarding both the computation of deltas, as well as the change operations, as these, currently stand intrinsically associated with the Experience Builder model. This comes off as an issue since the end objective of this work is to enable bidirectional model transformations between each OutSystems' builder and the Service Studio. A solution to this problem could include the adoption of a standard meta meta-model across all builders in what is called a *Single-Unified-Model*. Through the employment of a shared meta meta-model, the model differencing procedure and consequent generation of the change operations could be automated.

7.4 Running example

Referring back to the example of [subsection 3.2.4](#), the developed prototype enables a new flow of interaction between the developers Ann and Bob.

As previously, Ann, the builder user, starts from a baseline $OutSys_{empty}$ consisting of an empty OutSystems model. The corresponding Experience Builder model obtained by the projection function is therefore also an empty model, $ExpB_{empty}$. Ann starts her work and produces a new model $ExpB_A$ with two new flows as well as a connection relating them both. The delta computation process dif , yields a changelog with the set of all the operations that lead to $ExpB_A$:

$$dif(ExpB_A, ExpB_{empty}) = \{NewFlow[Flow1], NewFlow[Flow2], NewConn[Conn1]\}$$

By applying these operations to $OutSys_{empty}$ the program produces an updated version of the OutSystems model, $OutSys_B$:

$$OutSys_B = runOp(OutSys_{empty}, \{NewFlow[Flow1], NewFlow[Flow2], NewConn[Conn1]\})$$

This model is changed by Bob using Service Studio, resulting in $OutSys_C$. Ann now wants to proceed to make additional changes. The projection function is used to get an updated builder model, $ExpB_C$. Thus, instead of starting from scratch Ann can now see that Bob changed the *SetFaceID* screen's name to *ActivateFaceRecognition*.

$$ExpB_C = proj(OutSys_C)$$

Ann adds the *GoogleLogin* screen, connecting it to *LoginAndPasscodeFlow* and changing the *AnimatedBoardingOption1* flow name, from which we get a new model, $ExpB_D$. The program computes the delta between the two models:

$$dif(ExpB_D, ExpB_C) = \{NewFlow[Flow3], NewFlow[Flow4], UpdFlow[Flow1]\}$$

This delta is then applied to $OutSys_C$ in order to get the final OutSystems model $OutSys_D$:

$$OutSys_D = runOp(OutSys_C, \{NewFlow[Flow3], NewFlow[Flow4], UpdFlow[Flow1]\})$$

Ann and Bob were thus able to collaboratively work on the same app using their own preferred tools.

7.5 Final Remarks

The developed solution proved the designed strategy was indeed **viable and effective** in the goal of upgrading the builders (particularly, the Experience Builder) to application editors and consequently enhancing the interoperability across the **OutSystems Platform** tools. Furthermore, the achieved prototype **circumvented** the typical challenges associated with view-model bidirectional transformations scenarios where a **UI/UX** element can usually be converted with low effort to its respective code, whereas the reverse process holds significantly higher complexity and technical effort.

In this case, despite the Experience Builder (as well as the other builder tools) being tailored to the experience level of their users by abstracting unnecessary complexity, the tool model elements are derived from the OutSystems model, employing similar structures, attributes, and behavior. This promoted a consistent treatment of elements across models, which was beneficial to our goal of implementing bidirectionality over the different tools.

CONCLUSION AND FUTURE WORK

OutSystems provides a model-driven low-code development environment and delivery platform that allows developers to create enterprise-grade web and mobile applications. OutSystems developers interact with Service Studio, the platform's [IDE](#), using domain-specific visual languages to shape the fine-grained application model.

In recent years, OutSystems introduced the “Application Builders”, a set of tools that lower the learning curve even more and provide an inclusive entrance to non-IT developers. The builders focus on specific application development scopes such as User Experience or the definition of Business Processes. Each application domain requires a specific and appropriate meta-model abstracting selected parts of the underlying OutSystems model. Ultimately, they represent OutSystems concepts in a higher degree of abstraction, just like *views* of the base model.

OutSystems builders define a unidirectional model transformation strategy. This limitation hinders the interoperability between the different development tools in the OutSystems ecosystem. An application developed with one particular builder is unreadable in other builders and can only be further customized in Service Studio. However, changes made in the [IDE](#) are irreversible and unreadable by the original builder. This comes off as a direct consequence of the current problem of OutSystems builders defining unidirectional model transformations.

In this dissertation, it is presented a bidirectional model transformation algorithm aiming to improve the interoperability between the Outsystems Builders and Service Studio. The instantiated and evaluated implementation focuses on one particular builder of OutSystems, the “Experience Builder”.

The implemented strategy consists of a synchronization algorithm divided into two major parts. On the one hand, a projection function is responsible for locating and extracting the Experience Builder elements existing in the OutSystems model to reconstruct the builder model. On the other hand, a delta computation process gathers the alterations carried out by the developer during his interaction with the builder and produces the corresponding change operations necessary to propagate those changes to the OutSystems model.

The approach was validated by classifying into a set of comprehensive categories all the possible evolution scenarios of app development occurring from the combined use of the Experience Builder with Service Studio. Thus, it was possible to conclude the algorithm performs in a sound and predictable manner extending the currently offered interoperability.

As future work, we recognize the need to address the limitations referred to in [subsection 7.3.1](#). One of the major points to be improved shortly has to do with the broadening of the projection's scope, in order to have it acknowledge Experience Builder elements that were created in the [IDE](#).

On the subject of enforcing a possible *Single-Unified-Model*, it is safe to assume this would come as a significant challenge, that despite improving the algorithm's performance would not only prompt profound refactorings to the builder's code, as well as substantial architectural changes.

Regarding the modification of a model in a concurrent environment, Takeshi *et al.* [29] propose a synchronization algorithm capable of supporting concurrent updates which will be considered in future development stages. However, the overall performance of the prototype in the User Testing phase will ultimately dictate the concrete course of action.

Even though the work was conducted in a specific scenario, namely the OutSystems set of tools, it is believed our approach is general enough to be applicable in other scenarios.

Finally, it should be important to mention that the developed solution does not prevent a future evolution of Experience Builder to become a real-time editor of OutSystems models, instead of the current approach of applying changes in batch. This could be achieved by directly mapping interactions in Experience Builder to the suggested change operations and running those change operations immediately. As a consequence, this would make the computation of deltas unnecessary since it exists as a way of gathering all the change operations.

BIBLIOGRAPHY

- [1] *"Json.NET"*. URL: <https://www.newtonsoft.com/json> (cit. on p. 50).
- [2] P. Boström et al. "Formal Transformation of Platform Independent Models into Platform Specific Models in MDA". In: vol. 4355. Dec. 2006, pp. 186–200. ISBN: 978-3-540-68760-3. DOI: [10.1007/11955757_16](https://doi.org/10.1007/11955757_16) (cit. on pp. 9, 10).
- [3] K. Czarnecki et al. "Bidirectional Transformations: A Cross-Discipline Perspective". In: vol. 5563. June 2009, pp. 260–283. ISBN: 978-3-642-02407-8. DOI: [10.1007/978-3-642-02408-5_19](https://doi.org/10.1007/978-3-642-02408-5_19) (cit. on p. 10).
- [4] Z. Diskin et al. "From State- to Delta-Based Bidirectional Model Transformations: The Symmetric Case". In: vol. 6981. Oct. 2011, pp. 304–318. ISBN: 978-3-642-24484-1. DOI: [10.1007/978-3-642-24485-8_22](https://doi.org/10.1007/978-3-642-24485-8_22) (cit. on pp. 33–35).
- [5] N. Foster et al. "Combinators for bidirectional tree transformations: A linguistic approach to the view-update problem". In: *ACM Transactions on Programming Languages and Systems* 29 (May 2007). DOI: [10.1145/1232420.1232424](https://doi.org/10.1145/1232420.1232424) (cit. on p. 11).
- [6] ISO. *ISO/IEC 19501:2005 Information technology - Open Distributed Processing - Unified Modeling Language (UML) Version 1.4.2*. 2004. URL: <https://www.iso.org/standard/32620.html> (cit. on p. 5).
- [7] 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. DOI: [10.1109/MODELS-C.2019.00056](https://doi.org/10.1109/MODELS-C.2019.00056) (cit. on pp. 21, 22, 24).
- [8] H. Lourenço, J. Seco, and L. Carvalho. *"The View Update Problem in the OutSystems Aggregate Language"*. 2016. URL: https://nova-lincs.di.fct.unl.pt/system/publication_files/files/000/000/647/original/INForum_2016_paper_52-2.pdf?1469187572 (cit. on p. 10).

- [9] H. Lourenço et al. “LUV is not the answer: continuous delivery of a model driven development platform”. In: *MODELS '20: ACM/IEEE 23rd International Conference on Model Driven Engineering Languages and Systems, Virtual Event, Canada, 18-23 October, 2020, Companion Proceedings*. Ed. by E. Guerra and L. Iovino. ACM, 2020, 52:1–52:10. DOI: [10.1145/3417990.3419502](https://doi.org/10.1145/3417990.3419502). URL: <https://doi.org/10.1145/3417990.3419502> (cit. on pp. 21, 22, 24).
- [10] J. M. Lourenço. *The NOVAtesis L^AT_EX Template User's Manual*. NOVA University Lisbon. 2021. URL: <https://github.com/joaomlourenco/novathesis/raw/master/template.pdf> (cit. on p. ix).
- [11] T. Mens and P. Van Gorp. “A Taxonomy of Model Transformation”. In: vol. 152. Mar. 2006. DOI: [10.1016/j.entcs.2005.10.021](https://doi.org/10.1016/j.entcs.2005.10.021) (cit. on p. 9).
- [12] OMG. *INTRODUCTION TO OMG'S UNIFIED MODELING LANGUAGE™(UML®)*. 2020. URL: <https://www.uml.org/what-is-uml.htm> (cit. on p. 6).
- [13] OMG. *MDA®- THE ARCHITECTURE OF CHOICE FOR A CHANGING WORLD*. 2020. URL: <https://www.omg.org/mda/> (cit. on p. 5).
- [14] OMG. *The MetaObject Facility Specification™*. 2020. URL: <https://www.omg.org/mof/> (cit. on p. 7).
- [15] OutSystems. *Experience Builder*. 2020. URL: https://success.outsystems.com/Documentation/Experience_Builder (cit. on p. 16).
- [16] OutSystems. *Integration Builder*. 2020. URL: https://success.outsystems.com/Documentation/Integration_Builder_EAP/Introduction_to_Integration_Builder (cit. on p. 18).
- [17] OutSystems. *Integration Studio*. 2020. URL: https://success.outsystems.com/Documentation/11/Reference/Integration_Studio (cit. on p. 14).
- [18] OutSystems. *No Limits for Citizen Developers*. 2020. URL: <https://www.outsystems.com/citizen-developers/> (cit. on p. 15).
- [19] OutSystems. *Platforms' Services*. 2020. URL: <https://www.outsystems.com/evaluation-guide/platform-services/#1/> (cit. on pp. 13, 14).
- [20] OutSystems. *Service Studio*. 2020. URL: https://success.outsystems.com/Documentation/11/Getting_started/Service_Studio_Overview (cit. on p. 13).
- [21] OutSystems. *Workflow Builder*. 2020. URL: https://success.outsystems.com/Documentation/Workflow_Builder (cit. on p. 17).
- [22] H. Pacheco and A. Cunha. “Generic Point-free Lenses”. In: *MPC*. 2010 (cit. on p. 11).
- [23] R. Sanchis et al. “Low-Code as Enabler of Digital Transformation in Manufacturing Industry”. In: *Applied Sciences* 10 (2019), p. 12 (cit. on pp. 1, 2).

- [24] D. Schmidt. “Guest Editor’s Introduction: Model-Driven Engineering”. In: *Computer* 39 (Mar. 2006), pp. 25–31. DOI: [10.1109/MC.2006.58](https://doi.org/10.1109/MC.2006.58) (cit. on p. 6).
- [25] B. Selic. “Selic B.: The pragmatics of model-driven development. *IEEE Softw.* 20(5), 19-25”. In: *Software, IEEE* 20 (Oct. 2003), pp. 19–25. DOI: [10.1109/MS.2003.1231146](https://doi.org/10.1109/MS.2003.1231146) (cit. on p. 6).
- [26] S. Sendall and W. Kozaczynski. “Model Transformation: The Heart and Soul of Model-Driven Software Development”. In: *Software, IEEE* 20 (Oct. 2003), pp. 42–45. DOI: [10.1109/MS.2003.1231150](https://doi.org/10.1109/MS.2003.1231150) (cit. on p. 1).
- [27] M. Sjölund, P. Fritzson, and A. Pop. “Bootstrapping a Compiler for an Equation-Based Object-Oriented Language”. In: *Modeling, Identification and Control (MIC)* 35 (Mar. 2014), pp. 1–19. DOI: [10.4173/mic.2014.1.1](https://doi.org/10.4173/mic.2014.1.1) (cit. on p. 7).
- [28] P. Stevens. “Bidirectional Transformations in the Large”. In: *2017 ACM/IEEE 20th International Conference on Model Driven Engineering Languages and Systems (MODELS)*. 2017, pp. 1–11. DOI: [10.1109/MODELS.2017.8](https://doi.org/10.1109/MODELS.2017.8) (cit. on pp. 2, 10).
- [29] Y. Xiong et al. “Synchronizing concurrent model updates based on bidirectional transformation”. In: *Software and System Modeling - SOSYM 12* (Feb. 2011), pp. 1–16. DOI: [10.1007/s10270-010-0187-3](https://doi.org/10.1007/s10270-010-0187-3) (cit. on pp. 35, 37, 92).

