INSTITUTO SUPERIOR DE ENGENHARIA DO PORTO

MESTRADO EM ENGENHARIA INFORMÁTICA

isep

Discovering Meta-models of Low-code Applications

FERNANDO GABRIEL FERREIRA MOREIRA

novembro de 2022

POLITÉCNICO
DO PORTO

Instituto Superior de Engenharia do Porto

Discovering Meta-models of Low-code Applications

Fernando Gabriel Ferreira Moreira

A dissertation submitted in partial fulfillment of the requirements for the degree of Master of Informatics Engineering, Specialisation Area of Software Engineering

Supervisor: Alexandre Manuel Tavares Bragança
Co-Supervisors: Isabel De Fatima Silva Azevedo
Nuno Miguel Gomes Bettencourt

Evaluation Committee:
President:

Members:

Porto, 2nd November 2022

Abstract

Low-code platforms (LCAP) enable users to create apps of various types quickly and with little or no coding in general purpose programming languages. Despite their popularity, these platforms are often closed source and do not adhere to standards. Users of these platforms face two major issues: the first is the difficulty in the evolution of applications in terms of platform updates, and the second is the inability to migrate the applications to another platform, constraining the user into using the original platform. Thus, the goal of this work is to investigate the feasibility of discovering these platforms' meta-models using models exported by them as a starting point. This will enable apps to be migrated, for example, to a new version of the platform or to a different platform by describing transformations using the discovered meta-models.

Keywords: Model Driven Engineering, Meta-model Discovery, Low-code Applications

Resumo

As plataformas low-code (LCAP) são plataformas que permitem a utilizadores construir aplicações de todo o tipo rapidamente, com recurso a pouco ou nenhum código escrito numa linguagem de programação de uso genérico. Apesar do sucesso que estas plataformas têm usufruido, geralmente são plataformas de natureza codigo fechado e não seguem padrões. Dois grandes problemas que os utilizadores destas plataformas geralmente tem são evoluir as applicações à medida que a plataforma é atualizada e a impossíbilidade de migrar as suas applicações para outra plataforma, sem ter que as refazer. Assim, o objetivo deste trabalho é explorar a possibilidade de descoberta dos meta-modelos destas plataformas usando como base aplicações exportadas pelas mesmas. Isto irá permitir, por exemplo, que as applicações possam ser migradas para uma nova versão da plataforma ou para uma plataforma diferente, usando os meta-modelos encontrados para definir uma transformação. Posto isto, a solução desenvolvida é capaz de descobrir a estrutura dos modelos de entrada e ainda é capaz de deduzir quais campos são referencias a conceitos no meta-modelo.

Contents

List of Figures ix

List of Tables xi

1 Introduction 1
- 1.1 Context . . . 1
- 1.2 Problem . . . 1
- 1.3 Objective . . . 2
- 1.4 Structure . . . 2

2 State of the Art 5
- 2.1 Model Driven Engineering . . . 5
  - 2.1.1 Model . . . 5
  - 2.1.2 Meta-model . . . 6
  - 2.1.3 Model Transformation . . . 7
- 2.2 Low-Code Application Platforms . . . 8
  - 2.2.1 Comparison of LCAPs . . . 8
- 2.3 Technologies . . . 8
  - 2.3.1 Eclipse Modelling Framework . . . 8
- 2.4 Related work . . . 8
  - 2.4.1 Discovering Implicit Schemas in JSON Data . . . 9
  - 2.4.2 Towards Supporting SPL Engineering in Low-Code Platforms using a DSL Approach . . . 10

3 Value Analysis 13
- 3.1 New Concept Definition . . . 13
  - 3.1.1 Opportunity Identification . . . 14
  - 3.1.2 Opportunity Analysis . . . 15
  - 3.1.3 Idea Generation and Enrichment . . . 16
  - 3.1.4 Idea Selection . . . 16
- 3.2 Value Proposition . . . 17
  - 3.2.1 Costumer Profile . . . 17
  - 3.2.2 Value Map . . . 18
- 3.3 Business Model . . . 18

4 Analysis and Design 21
- 4.1 Requirements . . . 21
  - 4.1.1 Functional Requirements . . . 21
  - 4.1.2 Non Functional Requirements . . . 21
- 4.2 Design . . . 22
  - 4.2.1 Flow . . . 23

4.2.2 Logical 24
4.3 Algorithm 25
4.3.1 Meta-models 25
4.3.2 Definition 27

5 Implementation **33**
5.1 Structure 33
5.2 Archive 33
5.3 Metamodel 34
5.4 Ecore 35

6 Experimentation and Evaluation **37**
6.1 Hypothesis 37
6.2 Evaluation Criteria 37
6.3 Testing Tools 38
6.3.1 Input Model Validator 38
6.3.2 Generated Model Validator 38
6.3.3 Metamodel Validator 38
6.3.4 Model Generators 40
6.4 Results 40

7 Conclusion **43**
7.1 Goals Achieved 43
7.2 Limitations and Future Improvements 43
7.3 Contributions 43
7.4 Final Considerations 44

References **45**

A Value Analysis **47**
A.1 TOPSIS Calculations 47

List of Figures

2.1 Models, meta-models, and meta-metamodels. (Brambilla et al. 2017) . . . 6
2.2 Role and definition of transformations between models. (Brambilla et al. 2017) . . . 7
2.3 Process of discovering schema information from JSON documents, as illustrated by Cánovas Izquierdo et al. . . . 9
2.4 JSON Grammar defined in Xtext and its corresponding meta-model . . . 10
2.5 Process of discovering the meta-model from a LCAP, as shown in Bragança et al. 2021 . . . 11

3.1 Innovation Process (Koen et al. 2002). . . . 13
3.2 New Concept Development (Koen et al. 2002). . . . 14

4.1 Flow of the solution. . . . 23
4.2 Logic design. . . . 24
4.3 Class diagram of the LC-Metamodel. . . . 26
4.4 Class diagram dipicting the target meta-model. . . . 27
4.5 Evolution of a meta-model as more models are inspected, represented using a Class Diagram. . . . 27
4.6 New Concept. . . . 29
4.7 Concepts before merge, represented using an Object Diagram. . . . 30
4.8 Merged Concept, represented using an Object Diagram. . . . 30

5.1 Package structure of the project. . . . 33
5.2 Class diagram representing the implementation of LC-Metamodel. . . . 34
5.3 Class diagram representing the implementation of LC-Metametamodel. . . 35

6.1 Omnia Meta-metamodel. . . . 39

List of Tables

2.1 Comparison of LCAPs 8

3.1 SWOT Analysis. 15
3.2 Weight of each criteria. 17
3.3 Values for each alternative and criteria. 17
3.4 TOPSIS results. 17
3.5 Business Model Canvas 19

6.1 Meta-model validation results 41

A.1 Weight of each criteria. 47
A.2 Values for each alternative and criteria. 47
A.3 Normalized values. 47
A.4 Normalized weighted values. 48
A.5 Ideal and negative ideal solutions 48
A.6 Separation from S* and S’. 48
A.7 Relative closeness. 49

Chapter 1

Introduction

The context of the dissertation is the emphasis of this chapter. The dissertation's problem is then discussed, followed by the objectives that are defined to address that problem. Finally the overall structure of this document is described.

1.1 Context

This dissertation is part of the BAMoL project, whose main goal is designing and implementing a Domain Specific Language (DSL) for the Low-Code Application Platform (LCAP) named Omnia (*OMNIA - Low-Code Business Application Development Platform* 2022). This is a joint project between the team behind the LCAP, Instituto Superior de Engenharia do Porto (ISEP) and Faculdade de Engenharia da Universidade do Porto (FEUP).

At the moment, two approaches to building the DSL are being considered. The traditional method of manually constructing the DSL is one option. The other is a technique that seeks to automate as much of the DSL generation as possible, using exported models of LCAP-based applications as a starting point.

The focus of this dissertation will be on the automatic DSL generation. From a set of exported models deduce a meta-model that will serve as the basis for generating a DSL.

1.2 Problem

LCAPs are typically closed-source solutions that do not adhere to standards, with each platform possibly having their own approach on how it is designed and implemented. Furthermore, due to their closed source nature, the meta-model is usually not explicitly provided nor is it properly versioned. This results in the inability to migrate applications modeled with one LCAP to another, causing the user to become dependent on that platform. Another issue that ocasionally affects users of these platforms is related to updating the application with a new version of the LCAP, i.e. the user initially created an application with an older version of the platform and is then unable to make use of the features present in more recent versions, since no migration mechanisms might be implemented. These issues difficult the proper maintenance and evolution of applications. In a lot of cases, the only viable solution is to completely redo the application.

1.3 Objective

Since developing in an LCAP is, at its core, just creating and manipulating a model that represents the final application, a meta-model that sets the bounds of what is possible to develop through the platform needs to exist. The solution proposed by this dissertation aims to find this meta-model from a set of models (i.e. exported applications created in the target LCAP), which are represented in a serializable format, such as JSON or XML. For the sake of simplicity, the approach depicted in this document only deals with models in a JSON format. Applying this approach, the source and target meta-models can be constructed and then used to define transformation with the goal of, for example, migrate an application from a source LCAP to a target LCAP.

The main goal of this article is to test the hypothesis that automatically deducing a meta-model from exported models of applications built through LCAPs is possible. The deduced meta-model, in particular, should be accurate enough to ensure that models that adhere to it are accepted by the LCAP. In other words, models instantiated by the deduced meta-model should be valid when compared to the original meta-model. Furthermore, it should be as complete as possible in order to cover the wide range of features that a LCAP can have. This means that every feature that exists in the initial models should also be present in the deduced meta-model.

1.4 Structure

This document consists of 6 main chapters: **State of the Art**, **Value Analysis**, **Analysis and Design**, **Implementation**, **Experimentation and Evaluation** and finally **Conclusion**. This section gives an overview of each of these chapters.

State of the Art gives an overview of the knowledge relevant to understanding this dissertation, including an introduction to Model Driven Engineering (MDE), LCAPs and to the technologies used for this dissertation. Finally, a compilation of works related to meta-model deduction is presented.

Value Analysis includes a representation of the project's value analysis. The New Concept Definition (NCD) and its processes are explained at the outset. The work of this dissertation is then subjected to the Technique for Order of Preference by Similarity to Ideal Solution (TOPSIS) method in order to determine the best path of action. Finally, the chosen path will be depicted through the Value Proposition, Customer Value, and Canvas Business Model so that the reader may better understand the value that will be provided to the clients.

Analysis and Design starts with a enumeration of the functional and non functional requirements of the solution. The non-functional requirements are classified using FURPS+. The solution's design is then presented, with a flow and a logic view. The flow view is presented using a flowchart and provides an overview of the steps and data flow between steps, whereas the logical view is presented using a component diagram to present the solution's architecture. Finally, the meta-model deducing algorithm is described, beginning with an enumeration of its general steps and then progressing to a more detailed description of each step.

Implementation is where the implementation details of the solution are described.

Experimentation and Evaluation starts by presenting the definition of the testing methodology used to confirm the achievement of the goals that were defined in Objective. This includes the criteria that ensure that the goals are met and the testing infrastructure to validate those criteria. The testing infrastructure includes model generators, a mechanism to import the generated models into the LCAP and a validator that automatically validates the meta-model. Then, the solution is tested in a practical use case, using models exported through the Omnia platform to deduce a meta-model. Then, using the aforementioned testing infrastructure, the meta-model is validated to ensure that the criteria is met.

Conclusion concludes the document by discussing the goals that were met during this dissertation, followed by a discussion of the limitations and potential improvements to this work.

Chapter 2

State of the Art

This chapter contextualizes the work by providing an overview of the technical knowledge required to understand the work described in this dissertation. This includes an introduction to Model Driven Engineering and its basic concepts, as well as an overview of LCAPs, Language Workbenches and the technologies that are used for the implementation of the solution. Finally, the related works that serve as basis for the design of the solution are presented.

2.1 Model Driven Engineering

In the field of Software Engineering, models are used as a way to communicate and share knowledge about complex software. Model Driven Engineering (MDE) is an approach that makes models the core of software development, as opposed to being a supporting tool. This approach brings the benefits of modelling to software engineering tasks. Models and their transformations, both of which are defined using a modelling language, are the core ideas of MDE. This language's definition is a model in and of itself. This is referred to as meta-modelling in MDE, which entails modelling all of the different models that can be represented by a modelling language. This is a recursive procedure, which means that a meta-model can also be used to describe all the meta-models. As a result, everything in the MDE context is a model. (Brambilla et al. 2017)

2.1.1 Model

Because everything in the MDE perspective is a model, it is critical to understand the definition of a model. A model is a human-created representation of a concept or a real-world object, known as the subject. They facilitate a better understanding of the subject by displaying only the features that are relevant to a specific problem. Thus, models have at least two roles (Brambilla et al. 2017):

- **Mapping feature**: models are based on an original subject that is abstracted and generalized from a prototype of a category of subjects;
- **Reduction feature**: models only represent a subset of the original subject's characteristics.
- **Pragmatic feature**: models can be used in substitution of the original for a given purpose.

Models can also be used for a different purposes: **descriptive models** are used to characterize existing systems or contexts, whereas **prescriptive models** are used to

specify the scope and details of a problem study, or to indicate how a system should be built (Brambilla et al. 2017).

2.1.2 Meta-model

A meta-model represents a set of models. More specifically, a meta-model is model that captures all the features of that set of models, i.e., a model can be seen as an instance of a meta-model, as illustrated by Figure 2.1. A meta-metamodel representing all the possible meta-models also exists. Since, by definition, this meta-metamodel is also a meta-model, it recursively describes itself. This results in various levels of meta-modelling. Figure 2.1 depicts an example of a real-world application of meta-modelling. Real-world items (a movie, in this example) are displayed at level M0, while their modelled representations are displayed at level M1, where the model specifies the idea of Video and its properties (title in the example). This model's meta-model, shown at level M2, specifies the concepts used to define the model at level M1, particularly Class, Attribute, and Instance. Finally, level M3 includes the meta-metamodel, which defines the concepts utilized at M2. In the example, this set collapses into the single Class concept. There's no need for further meta-modelling levels beyond M3, as the meta-metamodel recursively describes itself.

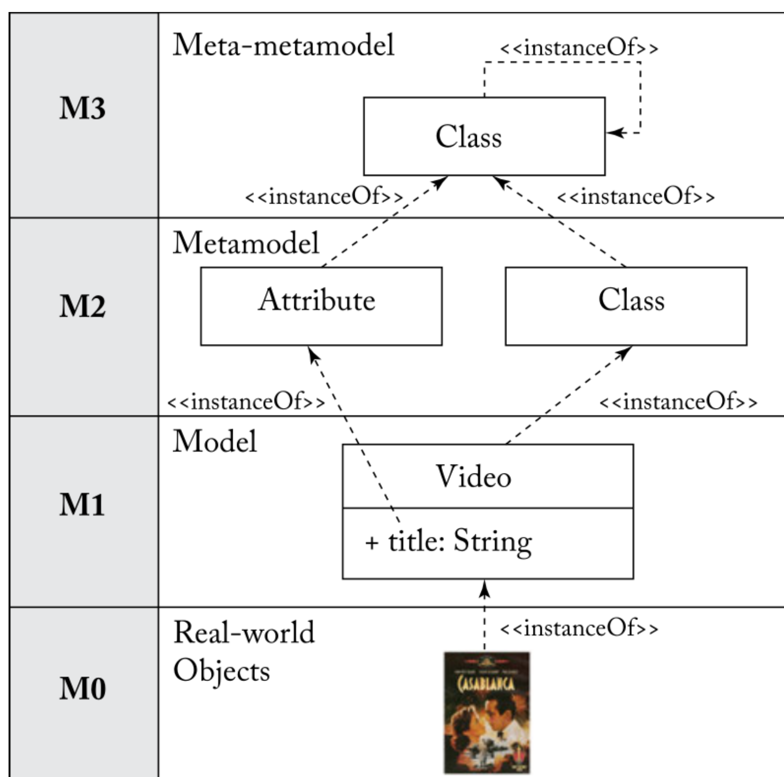


FIGURE 2.1: Models, meta-models, and meta-metamodels. (Brambilla et al. 2017)

Metamodels can be useful in a variety of applications (Brambilla et al. 2017):

- definition of new modeling or programming languages;
- definition of new modeling languages for information exchange and storage;

- definition of new properties or features to be linked with existing information (metadata).

2.1.3 Model Transformation

Another core concept of MDE is model transformations. They allow the mapping of one model to another. Transformations are performed between a source and a target model (level M1), but they are actually defined using their respective meta-models (level M2) (Brambilla et al. 2017). Model transformations can classified as **Model-to-Model**(M2M) or **Model-to-Text**(M2T). M2M transformations can be further classified into:

- *Exogenous* - The source meta-model differs from the target meta-model. This type of transformation creates an entirely new model based on the input model;
- *Endogenous* - The source and target meta-models are the same. Such transformations usually create, delete or update elements in the input model and are useful for performing refactorings to a model.

Transformations, including their meta-modeling, can be thought of as models and maintained as such. Two models (Ma and Mb) are shown in the lower section of Figure 2.2, along with a transformation Mt that converts Ma to Mb. The related meta-models (MMa, MMb, and MMt) are defined at the level above, to which the three models (Ma, Mb, and Mt) adhere. They all follow the same meta-metamodel, as a result. (Brambilla et al. 2017)

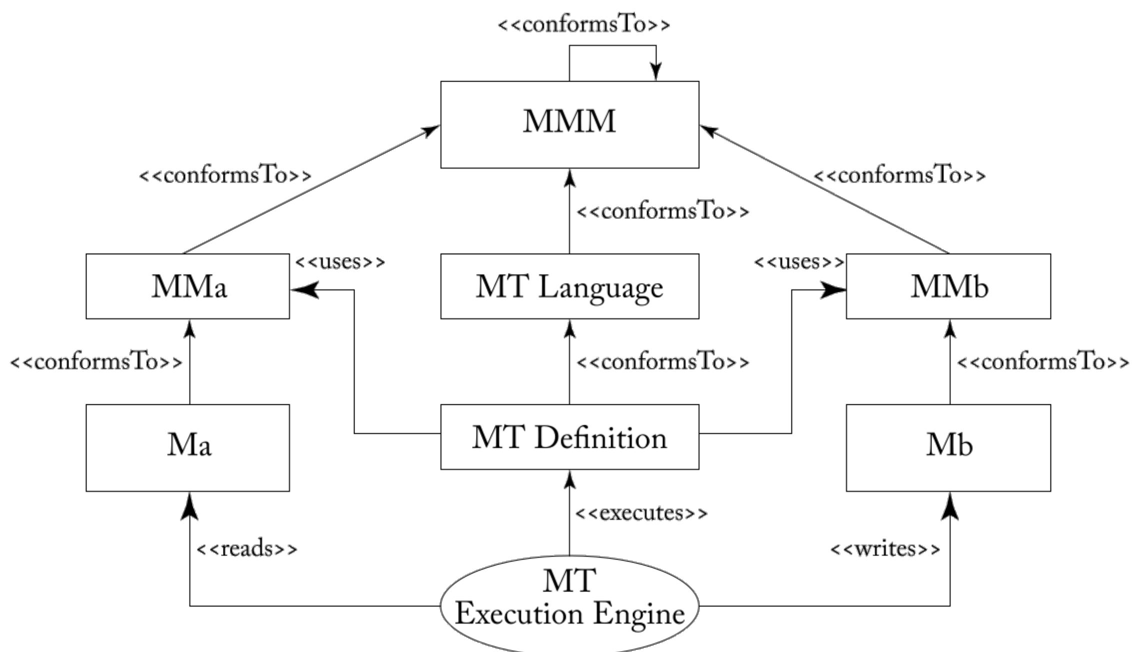


FIGURE 2.2: Role and definition of transformations between models. (Brambilla et al. 2017)

2.2 Low-Code Application Platforms

LCAPs are platforms designed using MDSE principles that allow users to build and deploy software applications that require little to no code written in a traditional programming language. Examples of such platforms include OutSystems, Microsoft PowerApps, Omnia and Mendix. These kinds of platforms provide numerous advantages when compared to traditional software development, such as requiring little to no knowledge of programming languages, faster development time, provide a centralized environment for configuration, delivery and maintenance of apps. (Richardson et al. 2014)

2.2.1 Comparison of LCAPs

A comparison of different LCAPs is now going to be presented. The focus of this comparison is going to fall particularly on their model import and export features: the available data formats (e.g. XML or JSON) and the way they can triggered (e.g. UI action, REST API, ...). Table 2.1 shows this comparison.

TABLE 2.1: Comparison of LCAPs

LCAP	Data Type	Manual Export/Import	Export/Import API	Data Format
Omnia	App Model	Yes	Yes (REST API)	JSON
PowerApps	App Model	Yes	Yes (SDK)	JSON
Mendix	App Model	Yes	Yes (SDK)	JSON

2.3 Technologies

The relevant technologies that were used in this dissertation are compiled in this section.

2.3.1 Eclipse Modelling Framework

The Eclipse Modelling Framework (EMF) is a modelling framework and code generator for creating tools and other applications based on a structured data model. EMF generates a set of Java classes for the model from a model specification provided in XMI, as well as a set of adapter classes that enable viewing and command-based editing of the model, and a basic editor, using tools and runtime support. The Eclipse Modelling Project is built around the EMF project. (Gronback 2022)

2.4 Related work

This section compiles and sums related works in the area of meta-model deduction.

- Discovering Implicit Schemas in JSON Data (Cánovas Izquierdo et al. 2013)
- Towards Supporting SPL Engineering in Low-Code Platforms using a DSL Approach (Bragança et al. 2021)

Some other works like Javed et al. 2008 or Zolotas et al. 2019 also exist, but their applicability in the context of this dissertation is not clear.

2.4.1 Discovering Implicit Schemas in JSON Data

In this work, Cánovas Izquierdo et al. 2013 demonstrate an approach in order to find the implicit domain model from a set of JSON-based services. To illustrate this, they use the REST API provided by TAN, which is the public transportation entity of Nantes, France. This API provides a set of services to query the bus/tram transportation system, such as querying by nearest bus stop by geolocation or which buses stop at a given stop.

Their suggested approach is illustrated by Figure 2.3 and consists of three steps:

1. **Pre-discovery**: From multiple textual JSON documents, their respective JSON model conforming to a JSON meta-model are extracted;
2. **Single-Service Discoverer**: Transformation of the previous JSON models into `Ecore` models by inferring their the JSON schema and then merging them, with the aim of obtaining the schema information for a concrete service;
3. **Multi-service Discoverer**: Merging of the models obtained in the previous phase in order to find a more complete application domain.

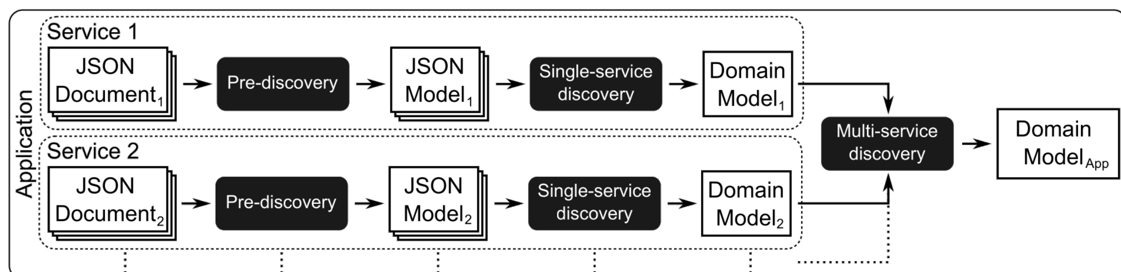


FIGURE 2.3: Process of discovering schema information from JSON documents, as illustrated by Cánovas Izquierdo et al.

In the **Pre-discovery** step, JSON documents are transformed into models conforming to a JSON meta-model. Xtext was used for this step, since it allows, from a grammar-based language definition, the generation of a meta-model and the tools for mapping instances of that language to their respective meta-model instance. Figure 2.4a shows part of the grammar definition and Figure 2.4b the generated meta-model.

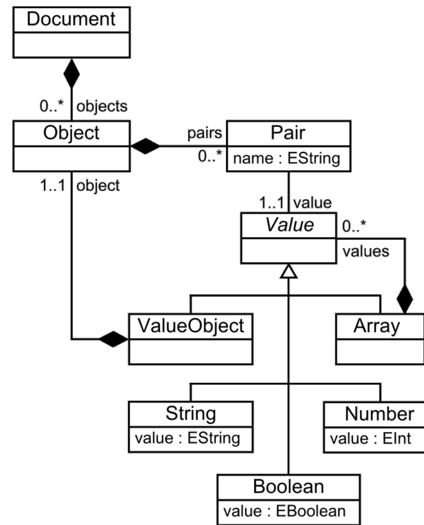
During the **Single-Service Discoverer** step, for each JSON service, the JSON models are transformed into Ecore models by applying defined mapping rules, applied to each `Object` concept in the JSON model to create a matching `Eclass`. The set of rules to be applied varies depending on if the target `Eclass` exists or not. If it does not exist, **creation** rules are applied and if it does **refinement** rules are applied instead. The definition of these rules can be found in the original article. The end result should be, for each service, an `Ecore` model representing its sub domain.

Finally, the **Multi-service Discoverer** step is performed. During this step, all of the `Ecore` models from the previous step are merged according to a set of rules as well. The final result is a model that is as close as possible to the complete domain model of the set of JSON-based services.

The implementation of the tool developed as part of this work can be found in the following link: https://github.com/SOM-Research/jsonDiscoverer

```
Document :
    objects+=Object
  | '[' objects+=Object (',' objects+=Object)* ']'
;
Object :
  '{' pairs+=Pair (',' pairs+=Pair)* '}'
;
Pair :
  name=STRING ':' value=Value
;
Value :
    StringValue | NumberValue | BooleanValue
  | ObjectValue | ArrayValue
;
ArrayValue :
  '[' values+=Value (',' values+=Value)* ']'
;
ObjectValue :
  value=Object
;
...
```

(A) JSON Grammar Definition



(B) JSON Meta-model

FIGURE 2.4: JSON Grammar defined in Xtext and its corresponding meta-model

2.4.2 Towards Supporting SPL Engineering in Low-Code Platforms using a DSL Approach

This paper describes the experience of using domain-specific languages to add support for a software product line engineering approach for a low-code application platform. To that end, a DSL capable of representing all of the LCAP's modeling concepts is developed (LC-DSL). Since LCAPs are usually closed source and don't provide access to the underlying meta-model but do provide ways to export models, a method was devised that allowed a meta-model to be deduced from exported models. The LC-DSL is then used to model groups of LCAP applications that can be managed as an SPL, in other words, the LC-DSL is a model that covers all LCAP applications that can be managed as a SPL. A variability DSL is also required to express variability (Variability DSL). The Variability DSL can be used to express the product line's variability as well as the configurations used for each SPL application. The LCAP or the LC-DSL can be used to model applications in this approach, but variability is expressed only outside the LCAP, through the use of annotations in the LC-DSL that reference elements of the Variability DSL. Specific models of the LCAP can be created by removing elements in the LC-DSL that are annotated with Variability DSL variability elements that are not included in the application's variability configuration model, which is also expressed in the Variability DSL. An import/export process was implemented to obtain the LC-DSL from the models in the LCAP and vice versa in order to integrate the existing LCAP with the new LC-DSL.

Since most if not all LCAPs are closed source solutions, the proposed methodology for finding the meta-model leverages those platforms' export/import functions, which allow users to export the application they were working into a model (usually in JSON or XML format). From this model, an approach similar to the one described in Section 2.4.1 is used to infer the meta-model. This is the **LCModel2Metamodel** step. Additionally, a semi-automatic step is also included in this process, known as **MetamodelCustomization**. The goal of this step is to allow an expert of the LCAP to validate and correct the resulting meta-model. This can include correcting non-identified class hierarchies,

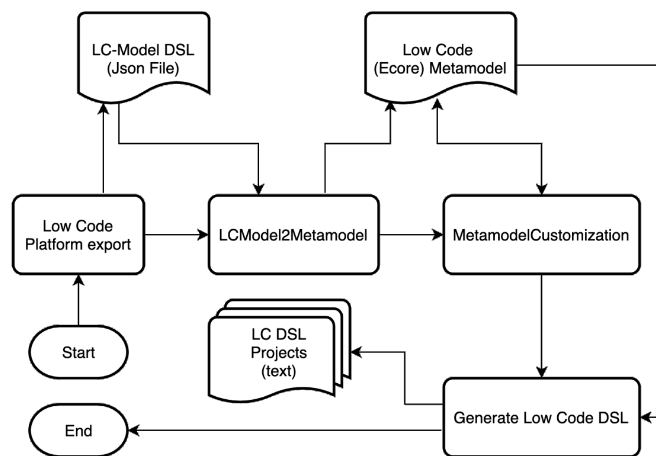


FIGURE 2.5: Process of discovering the meta-model from a LCAP, as shown in Bragança et al. 2021

adding missing classes like abstract classes and correcting wrong cardinalities and data types. Figure 2.5 provides a graphic overview of this process.

Chapter 3

Value Analysis

In this chapter, the value of this dissertation is evaluated. To that end, the New Concept Development (NCD) is first applied. Then, the value proposition is presented followed by the business model using the Business Model Canvas.

3.1 New Concept Definition

NCD is a model proposed by Koen et al. It aims to provide insight and a common terminology for the Fuzzy Front End (FFE) of the innovation process. As illustrated by Figure 3.1, the innovation process is composed by the FFE, the New Product Development (NPD) and Commercialization areas.

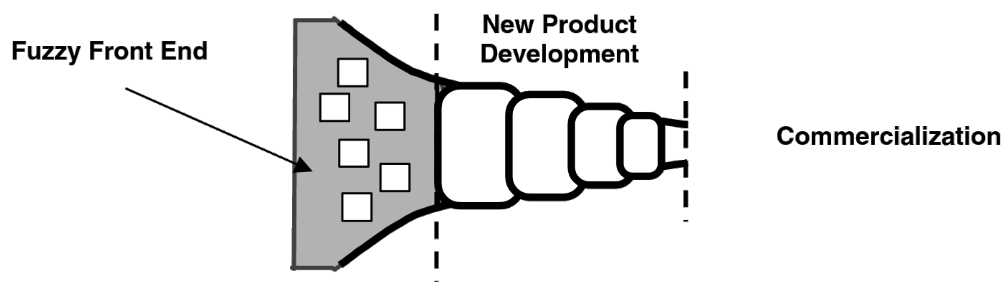


FIGURE 3.1: Innovation Process (Koen et al. 2002).

Figure 3.2 represents the NCD model. It has three key parts (Koen et al. 2002):

- The organization's leadership, culture, and business strategy drive the five key elements that are controllable by the corporation, which is known as the **engine or bullseye portion**;
- The FFE's five controllable activity elements are defined by the **inner spoke area**;
- Organizational capabilities, the external world (distribution channels, law, government policy, customers, competitors, and the political and economic climate), and the enabling sciences (internal and external) that may be involved are among the **influencing factors**.

This section will provide an application of the five controllable activities in the inner spoke area. They are the following (Koen et al. 2002):

- **Opportunity Identification** - During this activity, potential opportunities are identified;

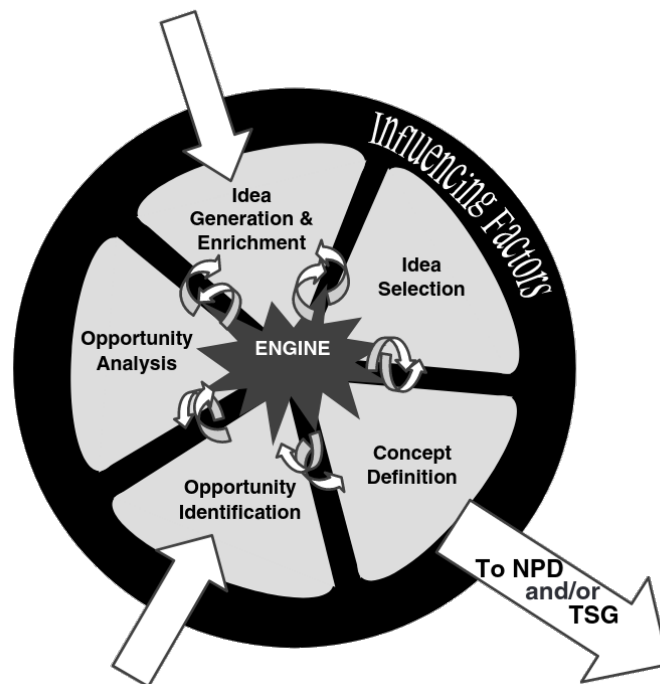


FIGURE 3.2: New Concept Development (Koen et al. 2002).

- **Opportunity Analysis** - An opportunity is assessed with the goal of confirming if it's worth pursuing. More information is needed to convert opportunity identification into specific business and technological opportunities. Early and often speculations about the technology and market is part of this process. A significant amount of effort may be required for focus groups, market surveys, and/or scientific trials;
- **Idea Generation and Enrichment** - This activity concerns the birth, development, and maturation of a concrete idea. The process of generating ideas is cyclical. The creative process encompasses building, tearing down, combining, reshaping, modifying, and upgrading ideas;
- **Idea Selection** - It's during this activity that ideas are chosen. Making the right decision is crucial to the company's long-term health and success.
- **Concept Definition** -

3.1.1 Opportunity Identification

The identified opportunity is the implementation of a DSL for the Omnia platform. The market for LCAPs is expected to grow from USD 13.2 billion in 2020 to USD 45.5 billion by 2025 (MarketsandMarkets 2022) and it's also a market with a lot of competitors (Gartner 2022). Providing a DSL gives a competitive advantage to the Omnia platform.

3.1.2 Opportunity Analysis

In order to further analyse the opportunity found in 3.1.1, a SWOT analysis is performed. Table 3.1 shows the SWOT analysis performed.

TABLE 3.1: SWOT Analysis.

STRENGTHS	WEAKNESSES
• Help users do repetitive modelling tasks more efficiently • Intuitive (graphical DSL) • Flexibility of using the DSL or the LCAP	• Less intuitive for inexperienced users (textual DSL)
OPPORTUNITIES	**THREATS**
• Growing market for LCAPs	• Competitive market for LCAPs

Strengths

- Textual DSLs can help users do repetitive tasks by simply allowing copy and paste of text or by using more advanced features, such as auto-complete or generation of DSL code. This alleviates much of the repetitive work that needs to be performed by a user;
- Graphical DSLs have the advantage of being more intuitive, since humans tend to understand pictures at a first glance than only text;
- Users have the choice of either using the DSL or the built in editor provided by the Omnia platform. This means they can either use the DSL for a more powerful alternative aimed at more experienced users, or the graphical editor, aimed at less experienced users, providing a more intuitive experience.

Weaknesses

- Less intuitive for inexperienced users. This applies mostly to textual DSLs, as trying to work with a textual language requires reading documentation in order to understand the features the language, as opposed to using a graphical DSL or the platform's editor.

Opportunities

- The LCAP market is growing fast. The market for LCAPs is expected to grow from USD 13.2 billion in 2020 to USD 45.5 billion by 2025 (MarketsandMarkets 2022).

Threats

- The LCAP market is fairly competitive, as there are many products offered in the low-code category (Gartner 2022).

3.1.3 Idea Generation and Enrichment

In order to capitalize on the opportunity, sfter considering and analysing it, two main ideas were thought of:

A **DSL Generation by deducing the meta-model** - The DSL is generated from a meta-model, which is automatically deduced from the model(s) of an exported application;

B **Manually building the DSL** - The DSL is manually built using the standard approach.

3.1.4 Idea Selection

While both approaches have the same value for the company behind Omnia, since this dissertation has an academic purpose, the approach with better applicability, while also remaining sufficiently accurate is the one that brings more value.

Using a method called Technique for Order of Preference by Similarity to Ideal Solution (TOPSIS), it is possible to compare the alternatives using a specified set of criteria, each having a weight, expressed as a percentage. Criteria can be categorized as benefits (more is better) or as negative (less is better). Each criteria was assigned with a value from the Satty scale for each alternative. The following criteria were considered:

- **Accuracy** - The accuracy of the resulting DSL regarding the coverage and correctness of all the features of the LCAP;
- **Applicability** - Measures how the approach has more applications beyond the main goal;
- **Time** - The time taken to develop the approach;
- **Difficulty** - The technical difficulty required to implement the approach.

Some factors are required in order to identify the initial parameters to be employed in the TOPSIS decision method. Starting with the weights of each criteria, the accuracy criteria is definitely the highest criteria, since if the resulting DSL is not accurate enought, i.e., it doesn't correctly cover enough of the LCAP's features, it has not practical use. The second most important criteria is applicability. Since this dissertation has an academic purpose, it's important to consider an approach that has other uses beyond being integrated into a closed source solution. The remaining criteria, difficulty and time, aren't as important, since, for the purpose of this work, they do not constitute a huge barrier.

Regarding the accuracy criteria, there is no doubt that having humans develop and validate the DSL will ensure its correct and complete coverage of the features of the LCAP. On the other hand, while the deduction approach surely won't produce a 100% complete DSL, the approach can be enhanced with a manual step for verifying and completing the feature coverage of the DSL.

Alternative A clearly has an advantage in terms of the applicability criteria, as it is designed to function with other LCAPs. Furthermore, the meta-model deduction approach can be used for purposes other than developing a DSL, such as allowing applications to migrate from one LCAP to another.

Alternative A will take less effort in terms of the time criteria because it is a nearly automatic operation. Modelling an application in the LCAP is the only manual step necessary, and it should take less time than manually constructing the DSL, as detailed in alternative B.

In terms of difficulty, alternative A is the more difficult approach because it requires the same technical knowledge as alternative B, while also requiring extra knowledge to properly deduce the meta-model.

With these considerations in mind, the input parameters for the TOPIS method were invented. Table 3.2 shows the weight for each criteria and Table 3.3 shows the score for each criteria and alternative.

TABLE 3.2: Weight of each criteria.

Criteria	Accuracy	Applicability	Difficulty	Time
Weight	50%	30%	10%	10%

TABLE 3.3: Values for each alternative and criteria.

	Accuracy	Applicability	Difficulty	Time
Alternative A	5	5	7	5
Alternative B	9	1	4	7

The intermediate steps of the application of the TOPSIS method can be found in appendix A.1. Table 3.4 shows the final result of the method. The alternative with the C* value closer to 1, which in this case is **Alternative A** is the chosen alternative.

TABLE 3.4: TOPSIS results.

	C*
Alternative A	0.545
Alternative B	0.455

3.2 Value Proposition

This section the value proposition is presented. It is separated into two dimensions: the **costumer profile** and **value map**. The costumer profile, as the name suggest, describes the costumer and the value map describes the value that is delivered to the customer.

3.2.1 Costumer Profile

The customer profile is divided into the following:

- Jobs - Describes the activities that the customer engages in;
- Pains - Describes the obstacles, risks or negative emotions that the customer faces when doing his job(s);
- Gains - Describes the benefits that the costumer values and/or expects.

The costumer that benefits the most from the integration of a DSL with a LCAP is an **advanced user**.

Jobs

An advanced user, being a user with enough technical knowledge, intends to use a LCAP to cut on development time, thus achieving a faster time to market.

Pains

During development, an advanced user typically needs to do similar / repetitive tasks, taking more time than should be necessary.

Gains

Advances users like using the best tool for the job. In other words, having flexibility to approach a problem in the way that achieves the most efficient solution is of great value to them.

3.2.2 Value Map

The value map can be divided into the following:

- Products and Services - The value delivered to the costumer;
- Pains Relievers - Describes how the value delivered helps the costumer with their pains;
- Gain Creators - Describes how the value delivered can be meaningful to the customer.

Products and Services

- DSLs allows users to do repetitive modelling tasks more efficiently
- Flexibility of using the DSLs or the Low Code Platform
- Applicable to other platforms
- Meta-model deduction approach has a possible application in other contexts

Pain Relievers

Using a textual DSL for example, the user can cut down the time spent doing tasks that are repetitive in nature.

Gain creators

The approach allows the development of applications to either be done through the LCAP's editor or using the DSL.

3.3 Business Model

The Business Model Canvas can be used to describe how a business operates. Table 3.5 shows the Business Model Canvas for the work developed in this dissertation.

TABLE 3.5: Business Model Canvas

Key Partners	Key Activities	Value Propositions	Customer Relationships	Customer Segments
• Omnia • ISEP • FEUP	• Research • Development	• DSLs allows users to do repetitive modelling tasks more efficiently • Flexibility of using the DSLs or the Low Code Platform • Applicable to other platforms • Meta-model deduction approach has a possible application in other contexts	N/A	**Advanced users**: Users that benefit from the advanced features of having a DSL integrated with the Omnia platform.
	Key Resources		**Channels**	
	• Developers		N/A	

Cost Structure	Revenue Streams
Research and development costs.	Financing from the Portugal 2020 programme.

Chapter 4

Analysis and Design

This chapter presents the analysis of the requirements and the possible designs of the solution.

4.1 Requirements

The section focuses on presenting the requirements necessary to achieve the intended solution.

4.1.1 Functional Requirements

FR1 The user should be able to generate a meta-model from an exported model;

FR2 The user should be able import into the LCAP a model created using the generated meta-model.

4.1.2 Non Functional Requirements

The identified non functional requirements are present here. They are classified according to the FURPS+ model.

Functionality

NFR1 The meta-model generation should be able to deduce the correct data type for a field;

NFR2 The meta-model generation should be able to correctly deduce the class of an object;

NFR3 The meta-model generation should be able to correctly deduce references to other elements;

NFR4 The meta-model generation should provide a manual step for verifying and completing the meta-model;

Usability

None were specified.

Reliability

None were specified.

Performance

NFR5 The meta-model generation should not take more than a 5 seconds.

Supportability

None were specified.

Design Constraints

NFR6 The solution should be as flexible as possible, in order to allow it to be integrated into a LW.

Implementation Constraints

NFR7 The solution must generate a meta-model in EMF (Ecore).

Interface Constraints

NFR8 The solution should integrate with VS Code.

NFR9 If possible, the solution should automatically integrate with the LCAP, i.e., exporting the model from an API and importing it back to the LCAP when changes to the model occur.

Physical Requirements

None were specified.

4.2 Design

The section describes the design of the solution. The relevant design decisions are documented here.

The proof of concept described in Bragança et al. 2021 serves as the main basis for the solution described here, as it is also a part of the BAMoL Project. Therefore, the design of the solution uses this work as the foundation and builds upon it, addressing some of its limitations. For example, it does not support more than one input model and, as a result, the input model must be as complete as possible, such that it covers all the features of the LCAP.

The approach described by Cánovas Izquierdo et al. 2013 deduces the M1 model from a M0 model (see 2.1.2), but the same approach can be applied to the context of this work, which is to deduce an M2 model from an M1 model. This approach accepts more than one input model, but it doesn't provide a step for manually refining the resulting meta-model.

4.2.1 Flow

The solution's design seeks to improve the constraints of the related works on which it is based. Support for a variety of exported formats is desired because it is for the solution to be as generic as possible. Figure 4.1 shows the flow of deducing the meta-model from an exported model. Importing a model back into the LCAP is just the same process but in reverse.

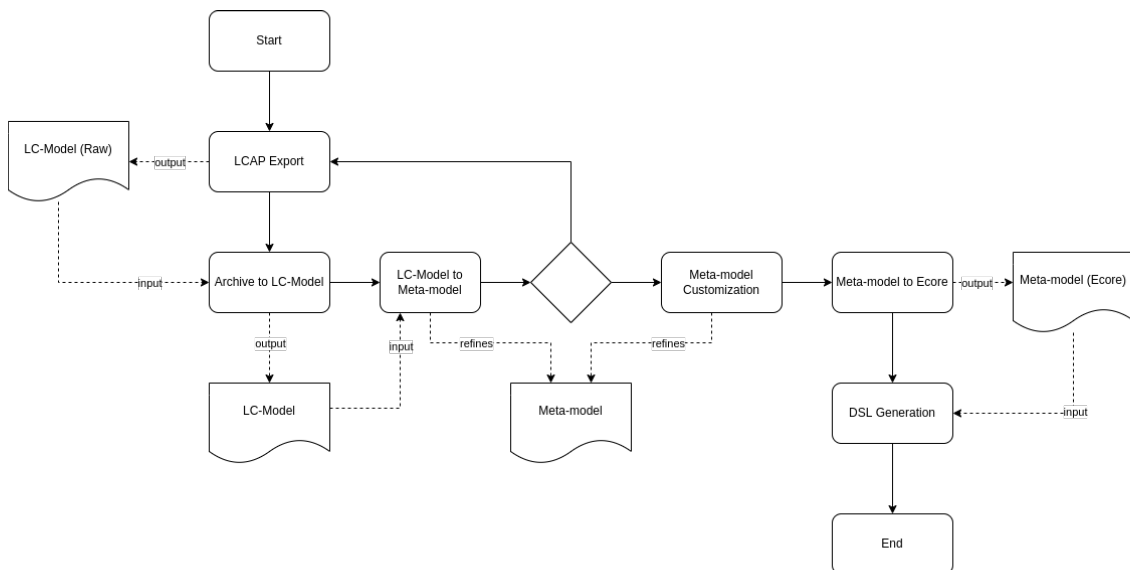


FIGURE 4.1: Flow of the solution.

LCAP Export, as the name implies, exports the model of an application built using a LCAP. This step can be manual, in which case the user uploads the file containing the LCAP app model, or it can be automatic, integrating with the LCAP through its API. As seen in 2.2, the result of the export is usually an archive containing a collection of files, represented as **LC-Model (archive)**.

The **Archive to LC-Model** step takes a **LC-Model (archive)** as an input and converts it to a **LC-Model**. The LC-Model follows a meta-model and captures the features of the exported model, such as the directory structure and files, as well as their contents, regardless of format. This allows the solution to process a variety of models.

LC-Model to Meta-model is where the actual deduction of the LCAP meta-model happens. Since it should be possible to deduce the meta-model from multiple models, this step results in the refinement of the **Meta-model** artifact. After this step, the process can loop back to the **LCAP Export** step or proceed to the **Meta-model Customization** step.

The **Meta-model Customization** step allows the user to manually correct or complete the resulting meta-model by adding, removing or changing existing elements.

The **Meta-model to Ecore** step performs a convertion of the meta-model to Ecore format so it can be used in the **DSL Generation** step.

The **DSL Generation** step abstracts the remaining process of the DSL generation, which is out of scope for this solution.

4.2.2 Logical

Figure 4.2 shows the logical overview of the solution. Note that, given the goal of this dissertation, the solution will not implement all the features described in Subsection 4.2.1, but rather only the features which are relevant for the goal, which is to prove that it is possible to infer all the features of input models. That being said, the solution will only implement the **Archive to LC-Model**, **LC-Model to Meta-model** and **Meta-model to Ecore** steps. The description of the components is as follows:

- **Meta-model Deduction** - represents the solution;
- **Archive** - responsible for features related to the creating of the **LC-Model**, particularly parsing the raw exported model;
- **Metamodel** - responsible for operations which involve the creation of the meta-model, such as **LC-Model to Meta-model**;
- **Ecore** - responsible for functionality related to Ecore, such are **creating meta-models in Ecore format**;

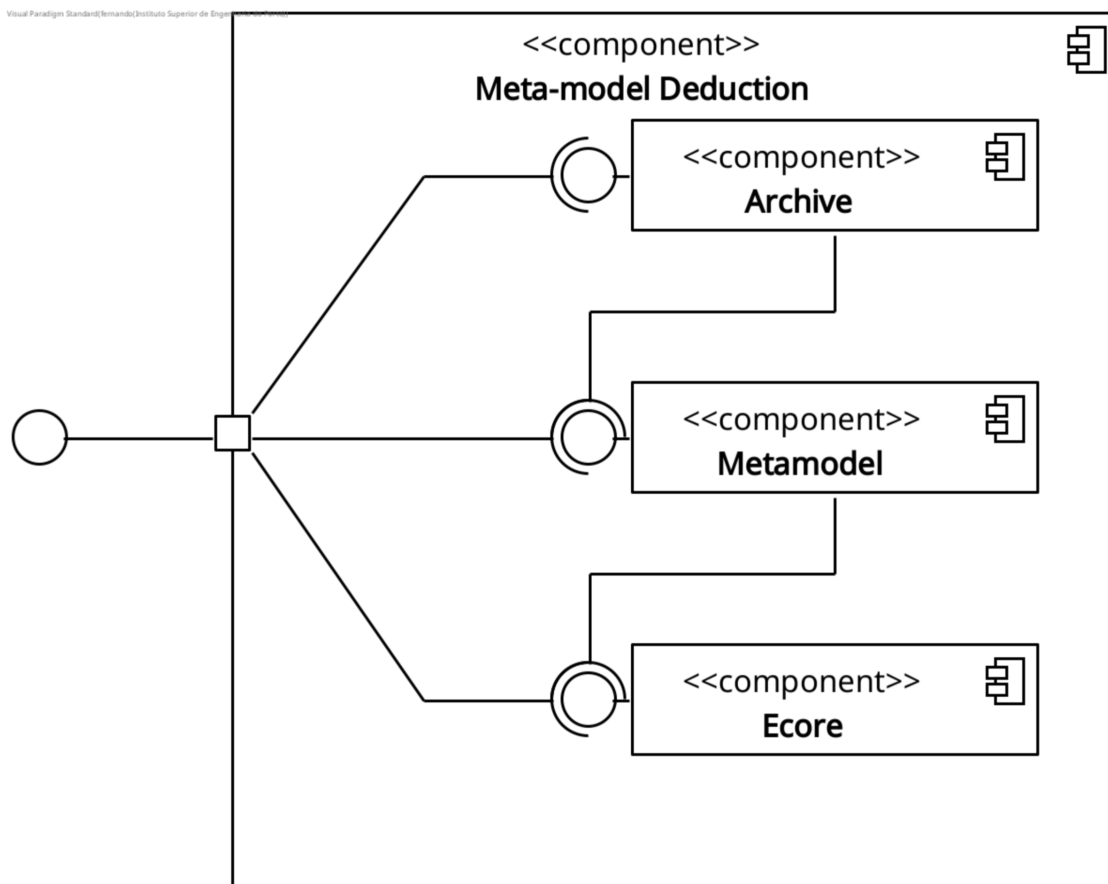


FIGURE 4.2: Logic design.

4.3 Algorithm

This Section presents the algorithm responsible for achieving the **LC-Model to Meta-model** step described in Subsection 4.2.1. Since this step is a model to model transformation in the MDE context, Subsection Meta-models exists to give an overview of the source and target meta-models, i.e., the **meta-model for LC-Model** and the **meta-model for the meta-model**, followed by a description of the algorithm that performs this step, starting with an enumeration of its general steps and then a more detailed description of each step.

4.3.1 Meta-models

Since this step is essentially **a model to model transformation** in the MDE context, it is necessary to define what are the source and target meta-models.

The source meta-model is the **meta-model of LC-Model**, which shall be referred to as **LC-Metamodel**. Its representation can be found in Figure 4.3 and the description of its elements is as follows:

- **Archive** - Is a collection of **ArchiveFile**;
- **ArchiveFile** - Represents a file and if it contains JSON is **JsonFile** or a **DefaultFile** if otherwise. Has a **name** and a **parent path**, which is relative to the root of the archive;
- **DefaultFile** - Extends **ArchiveFile** and represents a generic file. Adds the file's contents as the **content** property;
- **JsonFile** - Extends **ArchiveFile** and represents a JSON file. Adds a reference to a **JsonNode**;
- **JsonNode** - Represents a JSON value and can be an **ArrayNode**, an **ObjectNode**, a **TextNode**, a **BooleanNode**, a **NumberNode** or a **NullNode**;
- **ArrayNode** - Represents a JSON array and has the **elements** property, which is a collection of **JsonNodes**;
- **ObjectNode** - Represents a JSON object and has one or more **properties**, which are **KeyValuePairs**;
- **Property** - Represents a key/value pair that belongs to a JSON object, has a **key** property to identify the property and a **value**, which is a reference to a **JsonNode**;
- **TextNode**, **NumberNode**, **BooleanNode** and **NullNode** - Represent a JSON primitive value, whose type depends on the instance.

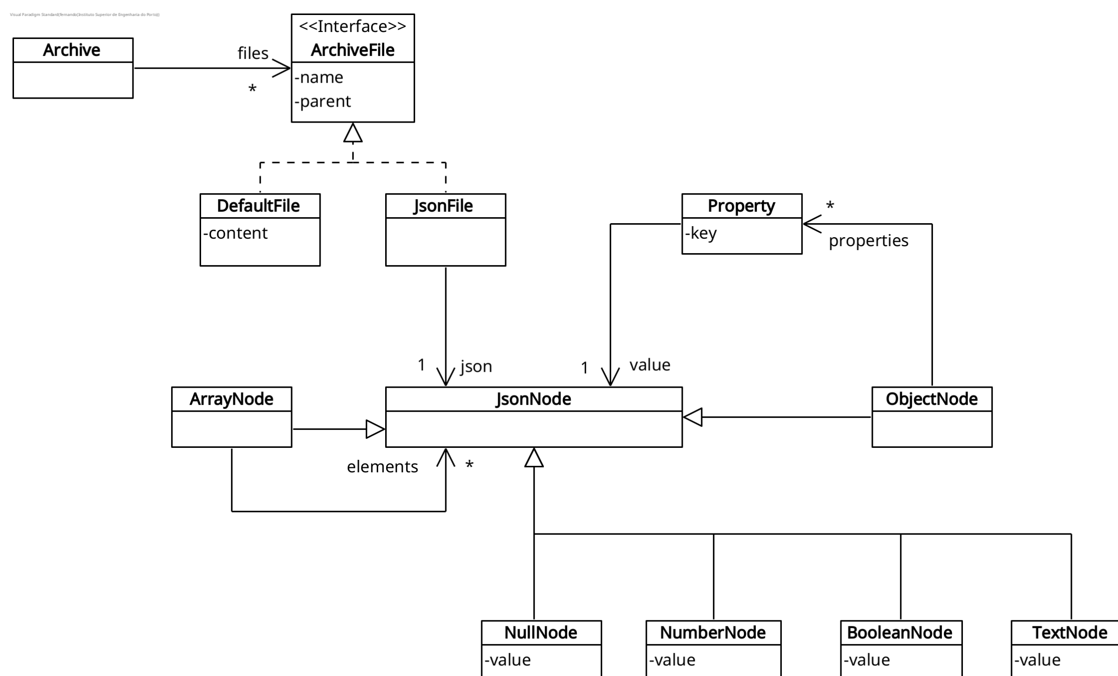


FIGURE 4.3: Class diagram of the LC-Metamodel.

The target meta-model is the **meta-model of Meta-model** (the artifact that is refined by the **LC-Model to Meta-model** step). Since it is a meta-model of a meta-model, by definition, the target meta-model is a **meta-metamodel for all LCAPs**, referred from now on as **LC-Metametamodel**. Figure 4.4 presents a class diagram representing LC-Metametamodel. Its elements are described as follows:

- **Metamodel** - Is a collection of **Concepts**;
- **Concept** - Is essentially a collection of **Attributes**. It has a name and two boolean properties, **isRoot** and **isAbstract**;
- **Attribute** - Represents a **name/value pair**. It has a name and two boolean properties, **isArray** and **isNullable**. Attributes can be:
  - **Simple Attribute** - Represents a primitive attribute, i.e. the value is either **null**, **boolean**, **number** or **string**;
  - **Reference** - The value of this attribute is a Concept, either by **containment** (the concept is defined as the value of the attribute) or a simple text attribute whose value identifies a Concept, which is indicated by the **isContainment** property;

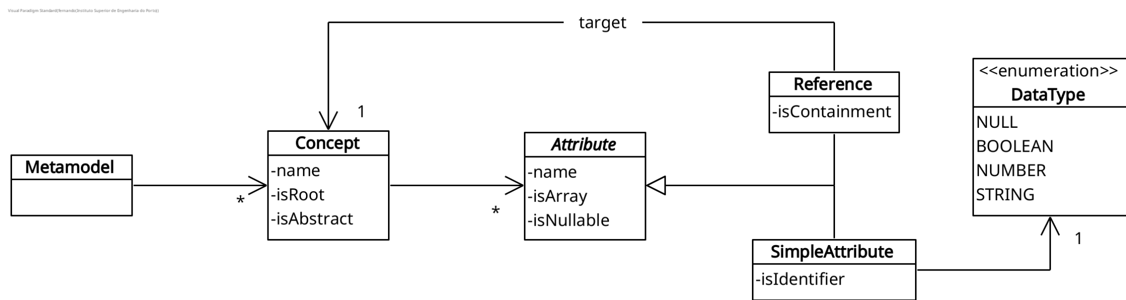


FIGURE 4.4: Class diagram dipicting the target meta-model.

4.3.2 Definition

The working principle of this algorithm is gradually building the meta-model with each model inspected, adding or updating Concepts with the new information found. This is illustrated by Figure 4.5, where initially the *Agent* Concept with the *name* Attribute is found in a model and then, when processing the next model, a new *description* Attribute is found and the Concept is updated accordingly. Finally, when another model is processed, an array Attribute with name *properties* is found. This time, the type of this Attribute is the *Property* Concept, with *name* and *type* Attributes.

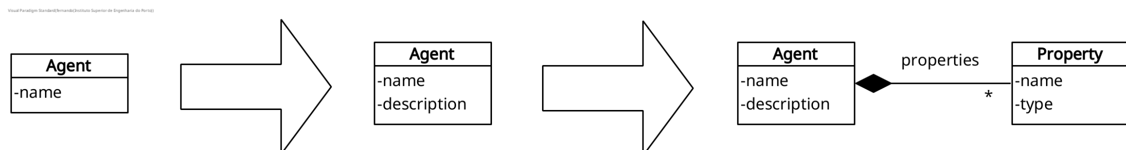


FIGURE 4.5: Evolution of a meta-model as more models are inspected, represented using a Class Diagram.

The algorithm has two major steps:

- Structure Inferrence - The goal of this step is to create an initial meta-model by inspecting the input models. The information gathered is relative to which data structures exist in the models, including their fields, respective types and other nested data structures. In MDE terms, this step can be seen as a **model to model transformation**.
- Reference Deduction - This step's goal is to identify which simple attributes are in fact references to a concept. In MDE terms, this step is a model refinement.

Structure Inferrence

This step consists in iterating over all the JSON files an recusively apply the following set of rules:

- An **ObjectNode** is mapped to a **Concept**:
  - If the ObjectNode is the root element of the file, i.e. it is referenced by a JsonFile, then the name in the resulting Concept is the name of the parent directory of the file and the **isRoot** flag is set to **true**. It is assumed that only ObjectNodes can be root nodes.

- If the ObjectNode is the value of a Property. In that case, the name of the resulting Concept includes the path from the **root Concept** to it. For example, if the name *Agent.attributes.multiplicity* is given to a Concept, that would imply the hierarchy of concepts **Agent -> Agent.attributes -> Agent.attributes.multiplicity**;
- Each **Property** of the JsonNode is mapped to an **Attribute** in the resulting **Concept**;
- The **isAbstract** flag is always set to false, since the fact that a JsonObject exists means the concept can't be abstract.

• The result of mapping an **ArrayNode** is an **Attribute**. Each element of the ArrayNode is mapped to to an Attribute according to the rules defined for that specific element and then the resulting Attributes are merged/reduced into a single Attribute, whose **isArray** flag is set to **true**;

• A **Property** pair is mapped to an **Attribute**:
  - The name of the resulting Attribute is the key of the pair and the **isIdentifier** flag is set to **true** if the name is equal to the identifier name and set to **false** if otherwise;
  - The **isArray** flag in the resulting Attribute is set to **false** if no other mapping rule is applied.
  - If the value is a primitive node (NullNode, NumberNode, BooleanNode or TextNode), then the resuling Attribute is a **SimpleAttribute** with the respective datatype. If the value is null, then the **isNullable** flag is set to **true**. Otherwise, it is set to **false**;
  - If the value is an ObjectNode, then the resuling Attribute is a **Reference** with the **isContaiment** flag set to **true** and the ObjectNode is mapped using its specific rules;

```
1 {
2   "name": "Company",
3   "attributes": [
4   {
5     "name": "code",
6     "type" "Text"
7   }
8   ]
9 }
```

LISTING 4.1: Example model in JSON format.

For example, take the input model shown in Listing 4.1, placed in the **Agent** directory under the name **Company.json**. When mapping this model to its respective meta-model, the process is started at its top-level ObjectNode, which has the Properties *name* and *attributes*. Since it's a top-level/root object, the name of its respective Concept is **Agent**, since it's the name of the directory where the file is located, and its *isRoot* flag is set to **true**. Then, its Properties are mapped to Attributes in the respective Concept. The **name** property is mapped to a SimpleAttribute, since its value is a text value and the isIdentifier flag is set to **true**, since the word *name* is assumed to be the designation for identifier properties. The value is a single value and it's not null so the **isArray** and **isNullable** flags are both set to **false**. As for the *attributes* property, since its value is an

ArrayNode which only contains one ObjectNode, the Property is mapped to a **Reference** whose *target* is the resulting Concept (mapped from the singular ObjectNode) and both its **isContainment** and **isArray** flags are set to **true**. The singular ObjectNode of the array is then mapped to a Concept, with **Agent.attributes** as its *name* and, since the object is a value of a property, its **isRoot** flag is set to **false**. Like the previous object, the Properties of the ObjectNode are then mapped to their respective Attributes in the Concept. The resulting meta-model instance represented using an object diagram can be found in Figure 4.6.

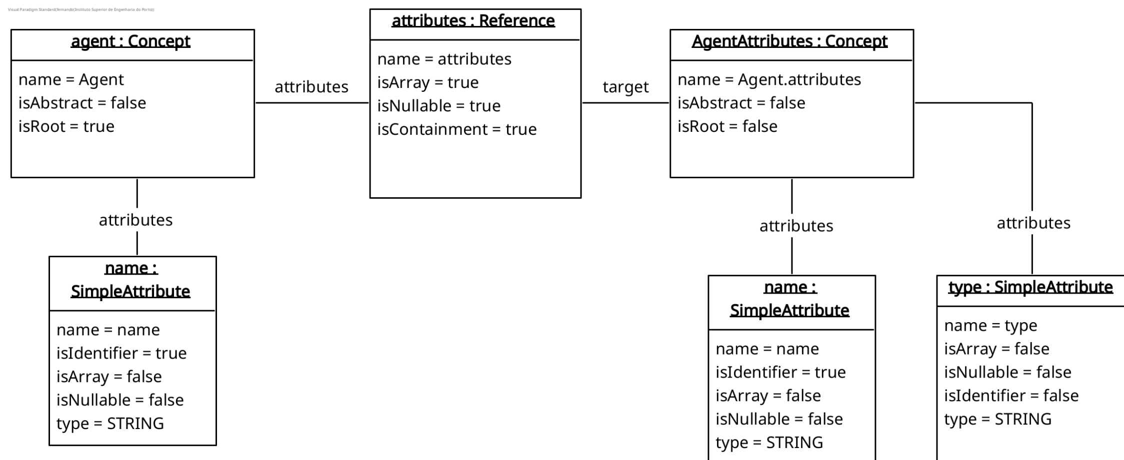


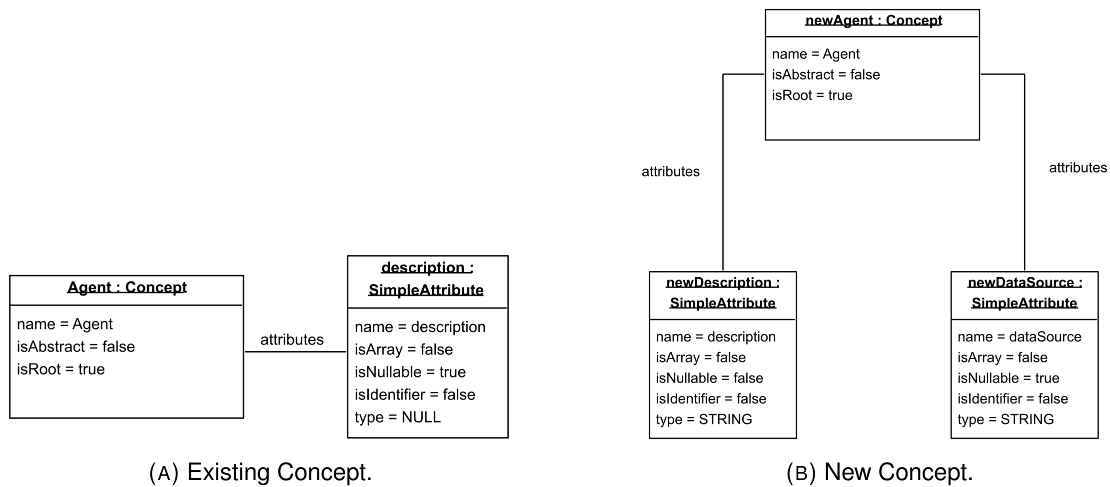
FIGURE 4.6: New Concept.

When a Concept or Attribute is added to the Meta-model it can already exist, since it could have been found previously. In this situation, new information is added to the current Concept/Attribute, according to the following rules:

- When **merging a Concept**:
  - New attributes are simply added to the existing Concept;
  - Existing attributes are merged;
- When **merging an Attribute**:
  - If the new Attribute is a **Reference** and the previous is a **SimpleAttribute**, upgrade it to a **Reference**. This feature is useful for the Reference Deduction step;
  - For the boolean flags, the merged value is obtained by performing an **OR** operation between the old and the new value;
  - For the **type** of the simple attribute, the merged value is the more general type between the old and new type. The types from most general to least general are: string > number > boolean > null;

For example, Figure 4.7 shows two Concepts with the name "Agent". The preexisting Concept, shown in Figure 4.7a, has one Attribute, with name "description", type *null* and *isNullable* set to **true**

The two Concepts are then merged, resulting in the Concept instance shown in Figure 4.8. Since it is a new, The **dataType** attribute is added as is, while the **description**



(A) Existing Concept.

(B) New Concept.

FIGURE 4.7: Concepts before merge, represented using an Object Diagram.

Attribute is changed according to the new information. The **isNullable** flag was kept unchanged from the previous value because it was already **true**, and the other Attributes saw no chance, while the *type* Attribute was changed from *null* to *string*, since it is the more general type of the two.

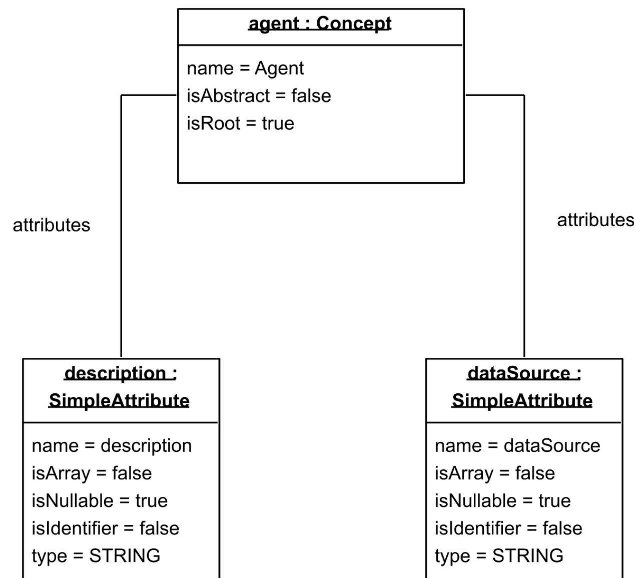


FIGURE 4.8: Merged Concept, represented using an Object Diagram.

Reference Deduction

The goal of this step is to determine which simple attributes are, in fact, a reference to another concept. The general rule for an attribute A to reference a concept C is that, given the set of values of A values(A) and the set of identifiers/names for the concept C names(C), the attribute A is a reference to C if names(C) contains all the values in values(A), i.e., values(A) is a subset of names(C). This relationship can be expressed in the form $values(A) \subset names(C)$.

This approach generally works well to identify references to a single concept, but if an Attribute references more than one Concept at the same time, that is, it references an abstract Concept that one or more concepts can extend, this method wouldn't find those kinds of references. One way to deal with that issue is to try to predict what the abstract concept could be.

In the context of metamodel inferrence, abstract Concepts are only relevant if at least one reference uses it as its target. Therefore, a way to find possible references to abstract concepts is to first consider a possible Concept hierarchy, including abstract and non-abstract Concepts, and, for each Concept, find the SimpleAttributes for which the relationship expressed by $values(A) \subset names(C)$ holds true. To find a Concept hierarchy, for each Concept C let ***attributes(C)*** be the the set which holds all the names of all the attributes belonging to Concept C. Then, for all the Concepts found in the 4.3.2 step, for each unique pair of Concepts C1 and C2, compute $attributes(C1) \cap attributes(C2)$. If the intersection of the sets is not empty, use it to define a new abstract Concept. Repeat this with the new abstract Concepts until no new Concept can be defined.

However, there is one issue with this approach: given a Concept C that extends abstract Concept AC, then it follows that $names(C) \subset names(AC)$. Transitively, any Attribute A for which $values(A) \subset names(C)$ holds true implies that $values(A) \subset names(AC)$ also holds true. This might result in wrongly deduced reference, which is why out of all the Concepts A might target, the most specific Concept (the one with the largest set of labels) must be chosen. This step of the algorithm can be summed in the following specific steps:

1. Find possible abstract Concepts by finding common attributes in the exsiting Concepts;
2. For each non-identifier SimpleAttribute in the meta-model find the Concepts for which the condition $values(A) \subset names(AC)$ holds true. If at least one Concept is found, select the **most specific one**;
3. For each SimpleAttribute in the last step, if one Concept was chosen, upgrade it to a **Reference**, with the selected Concept as its **target** and the *containment* flag set to *false*;

Chapter 5

Implementation

The solution was implemented as a Java 11 project, using Gradle as the built system. This chapter will describe how the project is structured and bridge the implementation with the design shown in Chapter 4. Initially, the package structure is shown, highlighting division of the packages by feature. Then, the implementation of each feature is further described in dedicated Section, including external dependencies, the implemented data model and the services that implement the logic of that feature.

5.1 Structure

The packages of the project are organized by feature, which are **Archive**, **Metamodel** and **Ecore**. Generically, each feature is then subdivided by **Model**, containing the data model for that feature, and by **Service**, which depends on Model and contains the logic of the feature. The services that existing inside the *services* package are each implemented as a single class with a single responsibility. Figure 5.1 shows a package diagram that reflects this structure.

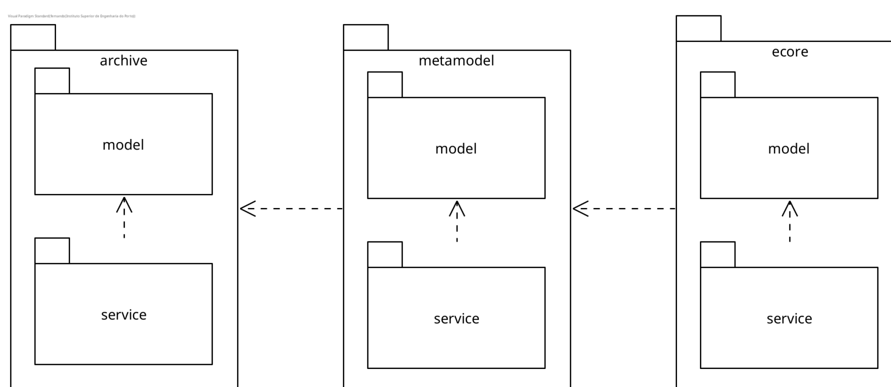


FIGURE 5.1: Package structure of the project.

5.2 Archive

This feature is where the data and logic responsible for implementing the **Archive to LC-model** step. Since the raw input model is assumed to be a ZIP archive containing JSON files, the dependencies in Listing 5.1 are required. The **org.apache.commons:commons-compress** dependency is a library that handles opening ZIP files and **com.fasterxml.jackson.core:jack databind** is a library for serializing, deserializing and manipulating JSON.

```
...
dependencies {
  implementation 'org.apache.commons:commons-compress:1.21'
  implementation 'com.fasterxml.jackson.core:jackson-databind:2.13.2.2'
}
...
```

LISTING 5.1: Archive feature dependencies

Figure 5.2 shows a class diagram of the implemented LC-Metamodel. It might seem incomplete when compared to Figure 4.3, but in reality, the **JsonNode** class is implemented by the **Jackson** library, so it is treated as a black box in this diagram.

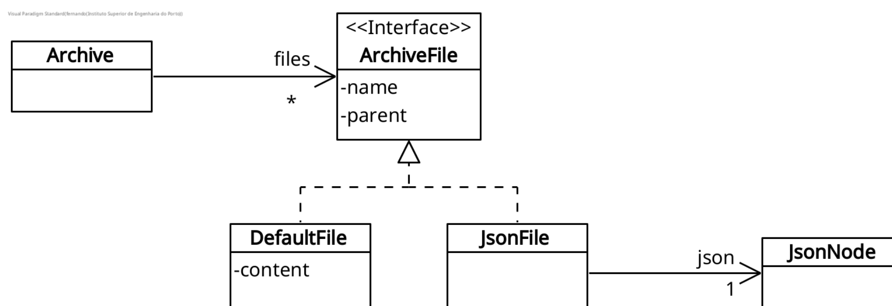


FIGURE 5.2: Class diagram representing the implementation of LC-Metamodel.

This feature has one service implemented, being the **ArchiveReader**, which is responsible for mapping a model directly exported from the LCAP to an Archive (LC-Model).

5.3 Metamodel

This feature is responsible for implementing the data and logic behind the **LC-Model to Meta-model** step. The implemented **LC-Metametamodel** can be seen in Figure 5.3. While it differs from the one presented in Figure 4.4, some of it is the same, so its description will focus on the major changes:

- A Concept can now be a **ConcreteConcept** or an **AbstractConcept**, removing the need for the *isAbstract* flag;
- An AbstractConcept, as the name implies, is a concept that cannot have instances of it in the model. The *subConcepts* association refences which Concepts extend the AbstractConcept. Its only purpose is to allow References to target more than one Concept;
- A ConcreteConcept is similar to the previous Concept, but it does not extend other Concepts, nor does it have the *isAbstract* flag;
- The SimpleAttribute has a *values* property, which is a list with all the values that Attribute had in the input models. Similarly, the *types* association's cardinality was changed to allow all the types that Attribute has in the input models. This allows for the deduction of references to simpler to implement;

- An Attribute can now also be an **EmptyArray**, since empty arrays weren't possible in the previous model.

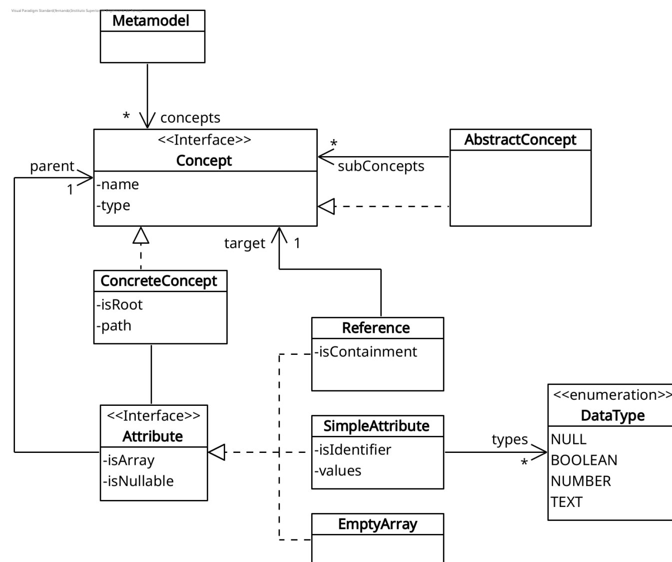


FIGURE 5.3: Class diagram representing the implementation of LC-Metametamodel.

As shown in Subsection 4.3.2, the **LC-Model to Meta-model** step is subdivided into two substeps and this feature has one service implemented for each of these substeps, which are the following:

- **ArchiveToMetamodel** - Responsible for implementing the logic behind the **Structure Inferrence** substep, mapping an Archive (LC-Model) to a meta-model instance.
- **MetamodelReferenceFinder** - Responsible for implementing the **Reference Deduction** substep, refining the meta-model by adding potential References.

5.4 Ecore

This feature is responsibe for implementing the logic of the **Meta-model to Ecore** step. For that, dependencies to Ecore are required, as shown in Listing 5.2;

```
1  ...
2  dependencies {
3    implementation 'org.eclipse.emf:org.eclipse.emf.ecore:2.26.0'
4    implementation 'org.eclipse.emf:org.eclipse.emf.ecore.xmi:2.16.0'
5  }
6  ...
```

LISTING 5.2: Project dependencies

No implemented model resides in this feature's package model, because Ecore's official meta-model implementation, available through the **org.eclipse.emf:org.eclipse.emf.ecore** dependency, is used instead.

This feature has one service implemented, which is the **MetamodelToEcore** class, which is in charge of mapping an input meta-model to an Ecore representation.

Chapter 6

Experimentation and Evaluation

Testing is essential for assuring software quality and dependability. After a product has been built, it is important to review it in order to verify if it achieves its goal and if it has enough quality.

This chapter starts with Section 6.1, where the hypothesis that this dissertation is trying to prove is presented, followed by the set of criteria that are used to prove the hypothesis, presented in Section 6.2. Next, Section 6.3 showcases the tools developed in order to help validate the defined criteria, which includes model generators, a mechanism to import the generated models into the LCAP and a validator that automatically validates the meta-model using a metamodel provided by the Omnia Platform. Then the results are shown and discussed in Section 6.4, where the solution is tested in a practical use case, using models exported through the Omnia platform to deduce a meta-model, which is then validated using the aforementioned testing tools, ensuring that the criteria are met.

6.1 Hypothesis

The hypothesis that this dissertation is trying to prove are the following:

H1 From exported models of applications built through LCAPs, it should be possible to deduce a meta-model which covers all the features of the input models;

H2 It should be possible to suggest, from the exported models, which properties might be references to a concept;

6.2 Evaluation Criteria

Besides the fulfilling of the requirements, some evaluation measures are used in order to validate the hypothesis, namely:

- Using the generated meta-model, it should be able to instantiate the same models that were used for deduction;
- Instances of the generated meta-model should be able to be imported into the LCAP;
- The coverage of the features of the input models.

6.3 Testing Tools

This section gives an overview of the tools used to help validate the solution. They include a **Input Model Validator**, used to validate the input model against its deduced meta-model, **Generated Model Validator** which is used to validate models generated used a inferred meta-model, a **Metamodel Validator** used to validate the meta-model against OMNIA's provided meta-model in JSON format, a **Model Generator** for automatically generating models adhere to the inferred meta-model.

6.3.1 Input Model Validator

The goal of this tool is to validate the the inferred meta-model by validating the input models against it. This assumes that the input models are valid from the outset and, as such, if the meta-model can successfully validate them, then the meta-model should be valid as well. However, since the meta-model only captures structural features of the models, its validity is limited to that and does not include semantical relationships between Concepts/Attributes of the meta-model.

6.3.2 Generated Model Validator

This tool is actually part of the OMNIA platform, which accessible through their REST API, whose documentation can be found in Omnia's Swagger UI page, or through the platform's modeler. In other to evaluate the validity of the models, both of these features can be used to import a model into the Omnia platform. This assumes that model is checked for its valitity, meaning that if the process is successful, then the model is valid.

6.3.3 Metamodel Validator

The purpose of this tool is to automatically validate the meta-models deduced by the solution. For that, the target meta-model needs to be available in a format that can easily be read by an algorithm. Thankfully, Omnia makes its meta-model available via a GitHub repository (https://github.com/OMNIALowCode/omnia3/tree/master/docs/pages/omnia3/Languages), an example of which is seen in Listing 6.1.

```
1  [
2    {
3      "Name": "Agent",
4      "Description": "...",
5      "Properties": [
6      {
7        "Name": "Name",
8        "Description": "The name of the entity (unique identifier).",
9        "TypeKind": "Primitive",
10       "TypeName": "Text",
11       ...
12     },
13     {
14       "Name": "Description",
15       "Description": "The textual explanation of the entities' purpose.",
16       "TypeKind": "Primitive",
17       "TypeName": "Text",
18       ...
19     }
```

```
20          ]
21        }
22      ...
23    ]
```

LISTING 6.1: Excerpt of OMNIA's meta-model.

Figure 6.1 shows the meta-metamodel that the Omnia meta-model follows. Therefore, the work performed by this tool is to compare Omnia's meta-model found in GitHub with the one that was inferred (see 4.4), particularly comparing analogous features, such as comparing **OmniaConcepts** with inferred **Concepts**.

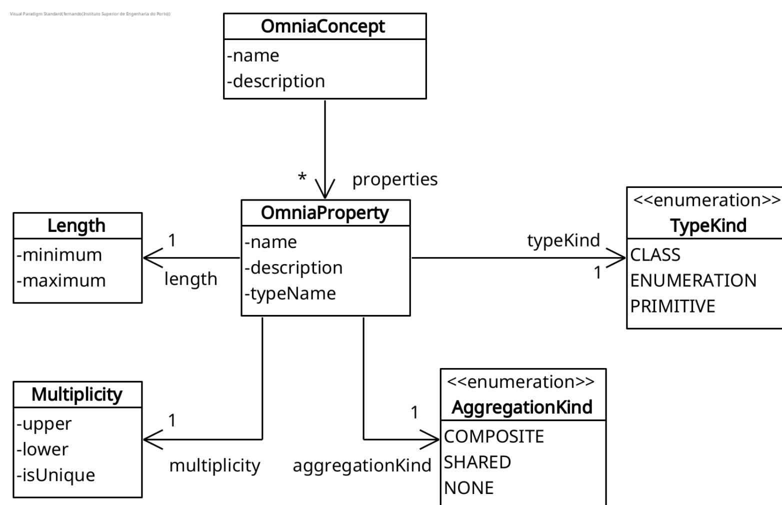


FIGURE 6.1: Omnia Meta-metamodel.

The algorithm behind this tool will essentially check if for each of the deduced Concepts, there is also a**matching OmniaConcept**, i.e. an OmniaConcept with the same name or with the same set of attributes/properties. Only non abstract Concepts are considered at this stage, since Omnia's meta-model makes no mention of them. This results in the following metrics:

- Total number of OmniaConcepts;
- Total number of Concepts existing in the input models;
- Total number of inferred Concepts;
- Number of Concepts that had a matching OmniaConcept;
- Number of Concepts that had did not have a matching OmniaConcept;
- Number of OmniaConcepts that had no matching Concept;

For each Concept and OmniaConcept compared, their attributes/properties are also compared. Similarly, for each Attribute, the algorithm will check if there is a **OmniaProperty with the same name** in the OmniaConcept. Attributes are then validated by asserting the following criteria:

- If the Attribute is a **SimpleAttribute**, the matching OmniaProperty should have *typeKind* of **PRIMITIVE** or **ENUMERATION**;

- If the Attribute is a **Reference**, the matching OmniaProperty should have *typeKind* of **CLASS** and the following requirements must be met:
  - If the *isContainment* flag is set to **true**, the *aggregationKind* of the OmniaProperty should be **COMPOSITE**;
  - The Reference target should match the OmniaProperty's *typeName* value;

6.3.4 Model Generators

This tool, as the name indicates, is responsible for generating models according to the specified meta-model. This process starts by recursively navigating the generated meta-model, starting with the root Concepts, and applying a set of generation rules.

If generating the value for a Concept:

- A Concept will originate a **JSON object**;
- If it is a **root Concept**, generate a configurable amount of JSON files. Inside each file the Concept's generated JSON object is placed, differing for each file;
- The Concept's Attributes will each originate a key/value pair inside the respective JSON object;

If generating the value for an Attribute:

- The attribute will originate a key/value pair inside its respective parent JSON Object;
- The key is the **name** of the Attribute;
- If the *isArray* flag is set to **true**, the generated value shall be an array and its has parameterized length;
- If the Attribute is a **SimpleAttribute**:
  - If the name of the Attribute is **lower**, **upper**, **aggregationKind** or **type** treat it as an ENUM and its value is one of the values that the Attribute assumed previously;
  - If the Attribute requires no special treatment, then it is generated randomly, with its type taken into account;
- If the Attribute is a **Reference**:
  - If the *isContaiment* flag is set to **true**, the value of the key/value pair is the JSON Object that the target Concept is mapped to;
  - If the *isContaiment* flag is set to **false**, the value of the key/value pair is the name of one of the generated instances of the Concept;

6.4 Results

Three models were exported from the OMNIA platform and used as input for the meta-model inferrence. Using the tool presented in Subsection 6.3.3, the data shown in Table 6.1 was obtained. The **Input model** column identifies the **model** used for inferring the

meta-model, the **Valid** column indicates whether the input model is valid when compared to the inferred meta-model, the **No. of Concepts in input models** column displays the number of unique Concepts existing in the input model, the **Percentage found** column shows the percentage of Concepts that were correctly deduced out of the total number of Concepts existing in the OMNIA meta-model, which is **56**. The **Relative percentage found** column represents the percentage of Concepts correctly deduced out of the total number of Concepts existing in the input models, the **Attributes found average percentage** represents the average percentage of Attributes found for each Concept and the **Not matched Concepts** is a list of deduced Concepts that did not have a matching OmniaConcept. All the percentages of this table are rounded to the nearest decimal unit.

TABLE 6.1: Meta-model validation results

Model	Valid	No. of Concepts in input models	Percentage found	Relative percentage found	Attributes found average percent	Not matched Concepts
#1	Yes	29	51.8%	100.0%	98.6%	N/A
#2	Yes	20	35.7%	100.0%	98.5%	N/A
#3	Yes	29	52.7%	100.0%	100.0%	
Combined	Yes	33	58.9%	100.0%	99.7%	

The models used as input did not cover the entirety of OMNIA's meta-model, since the goal here is not to find what OMNIA's meta-model is but rather if the algorithm can extract all the concepts present in the input models. Given the data in Table 6.1, all the Concepts that were present in the input models were found with no exception. This claim is supported by the **Valid** column, which indicates whether the input model follows the structure of the deduced meta-model, and is further validated by the **Relative percentage found** column. Furthermore, shown by column **Attributes found average percent**, practically all Attributes were found, due to the fact that input models contain all the Attributes, for each of the Concepts they contain. However, one exception to this fact was, for the input models except **#3**, the serialized instances of *DataSource*, which had its *attributes*, which is an array of containment references to the *Attribute* Concept, with less fields than what an *Attribute* is defined in the Omnia meta-model.

Importing a model generated using the tool described in Subsection 6.3.4 using the REST API yields an **HTTP 500** error message, displayed in Listing 6.2. Unfortunately, this does not give much room for discussing, however it is possible to speculate on the reasons the models might be invalid. For example, the value of an Attribute could be limited by a certain pattern, by length or by the value of another Attribute. Or the relationship between two Concepts could be constrained in the number of relationships for example. These constraints are not explicit, neither in the documentation nor the models themselves, which highlights one of the biggest limitations of generating a meta-model from models, being that such constraints are not trivially identifiable, and even if some are identified, their accuracy is not guaranteed.

```
1 {
2   "code": "InternalError",
3   "message": "The server encountered an internal error. Please retry the
    request."
4 }
```

LISTING 6.2: Error message when importing generated models

The deduced references for the combined model were the following:

- Form.entity -> Document, Agent, Serie, Event, GenericEntity, DataSource, Commitment and Resource
- Form.dataSource -> DataSource;
- List.query -> Query;
- List.dataSource -> DataSource;
- Agent.dataSource -> DataSource;
- Query.dataSource -> DataSource;
- Querytable.definition -> Document, Agent, Serie, Event, GenericEntity, DataSource, Commitment and Resource;
- Serie.dataSource -> DataSource;
- Document.dataSource -> DataSource;
- Resource.dataSource -> DataSource;
- Commitment.dataSource -> DataSource;
- StateMachine.definition -> Document, Agent, Serie, Event, GenericEntity, Commitment and Resource;
- AuthorizationPolicy.permissions.roles -> AuthorizationRole
- Event.dataSource -> DataSource
- GenericEntity.dataSource -> DataSource

While there is no way to determine if the references are 100% correct, from using the OMNIA Platform, it is possible to deduce that all of the references are correct.

Chapter 7

Conclusion

This chapter describes in full all of the discoveries made during the dissertation. Its objective is to establish a connection between the accomplishments made and the goals. This chapter will show how the goals were met and then present solution's limitations and future work. Then, the contributions of the work presented in this dissertation are explained. Finaly, the document will end with a section devoted to final considerations.

7.1 Goals Achieved

Comparing the initial goals with the obtained results, it is possible to conclude that generated meta-model covers all the structural features of the input models. Furthermore, it is possible to suggest possible references which are, for the most part, accurate.

7.2 Limitations and Future Improvements

It is a complex challenge to infer a meta-model from a collection of models. While it is undoubtedly possible and generally successful to infer a meta-structure, its an entirely different matter when attempting to infer the implicit relationships between its elements and their constraints. For example, with this approach it is not possible to determine whether a field's value is dictated by another field or follows a pattern.

Areas of the work that could be improved are adding a mechanism to predict the name of the identifier attribute, removing the need for it to be provided as a parameter and further automating the meta-model discovery process. Additionally, features that allow for manual changes in the generated meta-model could be implemented, including adding constraints or adding, updating and removing elements.

Currently, the application only supports models that are serialized in JSON, although some LCAPs might not provide models with such format. The solution's usefulness would increase if it could handle several data formats. This leads to another area for improvement, which is the evaluation of the approach using models from various LCAPs to further ensure its adaptability and broad applicability.

7.3 Contributions

The approach presented in this dissertation provides contributions towards the BAMoL project as well as towards the area of meta-model discovery. To that end, the work

presented in this document was adapted and submited as an article in the MODELSWARD 2023 conference (Moreira et al. 2022). This conference provides a venue for participants to present their research findings and application experience with model-based techniques for developing various types of systems. As of the writing of this document, the article is still pending review.

7.4 Final Considerations

Last but not least, the author views this dissertation as a success because the majority of the goals have been met. This approach has a lot of potential and can be used to solve other issues.

This was an interesting project that forced the author to consider the most effective methods for conducting research as well as how to effectively manage the thesis and implementation components. In the end, the author sees this project as an important step for Meta-model discovery, one with the potential to scale to other fields, and one that has been very rewarding because it gave him the chance to consider his own development.

References

Bragança, Alexandre, Isabel Azevedo, Nuno Bettencourt, Carlos Morais, Diogo Teixeira and David Caetano (17th Oct. 2021). 'Towards supporting SPL engineering in low-code platforms using a DSL approach'. In: *Proceedings of the 20th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences*. GPCE '21: Concepts and Experiences. Chicago IL USA: ACM, pp. 16–28. ISBN: 978-1-4503-9112-2. DOI: 10.1145/3486609.3487196. URL: https://dl.acm.org/doi/10.1145/3486609.3487196 (visited on 06/02/2022).

Brambilla, Marco, Jordi Cabot and Manuel Wimmer (30th Mar. 2017). *Model-Driven Software Engineering in Practice*. Google-Books-ID: dHUuswEACAAJ. Morgan & Claypool Publishers. 207 pp. ISBN: 978-1-62705-708-0.

Cánovas Izquierdo, Javier Luis and Jordi Cabot (2013). 'Discovering Implicit Schemas in JSON Data'. In: *Web Engineering*. Ed. by Florian Daniel, Peter Dolog and Qing Li. Red. by David Hutchison, Takeo Kanade, Josef Kittler, Jon M. Kleinberg, Friedemann Mattern, John C. Mitchell, Moni Naor, Oscar Nierstrasz, C. Pandu Rangan, Bernhard Steffen, Madhu Sudan, Demetri Terzopoulos, Doug Tygar, Moshe Y. Vardi and Gerhard Weikum. Vol. 7977. Series Title: Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 68–83. ISBN: 978-3-642-39199-6 978-3-642-39200-9. DOI: 10.1007/978-3-642-39200-9_8. URL: http://link.springer.com/10.1007/978-3-642-39200-9_8 (visited on 06/02/2022).

Gartner, Inc (2022). *Enterprise LCAP (Low-Code Application Platforms) Reviews 2022 | Gartner Peer Insights*. Gartner. URL: https://www.gartner.com/market/enterprise-low-code-application-platform (visited on 14/02/2022).

Gronback, Richard (2022). *Eclipse Modeling Project | The Eclipse Foundation*. URL: https://www.eclipse.org/modeling/emf/ (visited on 01/02/2022).

Javed, Faizan, Marjan Mernik, Jeff Gray and Barrett R. Bryant (1st Aug. 2008). 'MARS: A metamodel recovery system using grammar inference'. In: *Information and Software Technology* 50.9, pp. 948–968. ISSN: 0950-5849. DOI: 10.1016/j.infsof.2007.08.003. URL: https://www.sciencedirect.com/science/article/pii/S0950584907000882 (visited on 06/02/2022).

Koen, Peter, Greg Ajamian, Scott Boyce, Allen Clamen, Eden Fisher, Stavros Fountoulakis, Albert Johnson, Pushpinder Puri and Rebecca Seibert (2002). 'Fuzzy Front End: Effective Methods, Tools, and Techniques'. In.

MarketsandMarkets (2022). *Low-Code Development Platform Market Size, Share and Global Market Forecast to 2025 | MarketsandMarkets*. URL: https://www.marketsandmarkets.com/Market-Reports/low-code-development-platforms-market-103455110.html (visited on 14/02/2022).

Moreira, Fernando, Alexandre Bragança, Nuno Bettencourt and Isabel Azevedo (1st Nov. 2022). 'A Proposal Towards Discovering Metamodels from Low-Code Application Platforms'. In: *Proceedings of the 11th International Conference on Model-Driven Engineering and Software Development*. 11th International Conference on Model-Based Software and Systems Engineering. Lisbon, Portugal.

OMNIA - Low-Code Business Application Development Platform (2022). OMNIA - Low-Code Business Application Development Platform. URL: https://omnialowcode.com/ (visited on 14/02/2022).

Richardson, Clay and John Rymer (2014). *New Development Platforms Emerge For Customer-Facing Applications*. Cambrige, MA, USA: Forrester.

Zolotas, Athanasios, Nicholas Matragkas, Sam Devlin, Dimitrios S. Kolovos and Richard F. Paige (1st Feb. 2019). 'Type inference in flexible model-driven engineering using classification algorithms'. In: *Software & Systems Modeling* 18.1, pp. 345–366. ISSN: 1619-1374. DOI: 10.1007/s10270-018-0658-5. URL: https://doi.org/10.1007/s10270-018-0658-5 (visited on 24/02/2022).

Appendix A

Value Analysis

Appendix for the Value Analysis chapter.

A.1 TOPSIS Calculations

This section contains the full applications of the TOPSIS method present in 3.1.4.

Initial parameters

TABLE A.1: Weight of each criteria.

Criteria	Accuracy	Applicability	Difficulty	Time
Weight	50%	30%	10%	10%

TABLE A.2: Values for each alternative and criteria.

	Accuracy	Applicability	Difficulty	Time
Alternative A	5	5	7	5
Alternative B	9	1	4	7

Step 1

Each value is normalized according to formula xpto. The results are shown in Table A.3.

TABLE A.3: Normalized values.

	Accuracy	Applicability	Difficulty	Time
Alternative A	0.486	0.981	0.868	0.581
Alternative B	0.874	0.196	0.496	0.814

Step 2

Each value is multiplied with the respective weight of the criteria. The results are shown in Table A.4.

TABLE A.4: Normalized weighted values.

	Accuracy	Applicability	Difficulty	Time
Alternative A	0.243	0.294	0.087	0.058
Alternative B	0.437	0.059	0.050	0.081

Step 3

The ideal (A*) and negative ideal solutions (A') are determined. The results are shown in Table A.5.

TABLE A.5: Ideal and negative ideal solutions

	Accuracy	Applicability	Difficulty	Time
A*	0.437	0.294	0.050	0.058
A'	0.243	0.059	0.087	0.081

Step 4

For each alternative the separation from the ideal solution (S*) and negative ideal solution (S') is found. The results are shown in Table A.6.

TABLE A.6: Separation from S* and S'.

	S*	S'
Alternative A	0.198	0.236
Alternative B	0.236	0.198

Step 5

Finally, the relative closeness is calculated. The results are shown in Table A.7.

TABLE A.7: Relative closeness.

	C*
Alternative A	0.545
Alternative B	0.455