A Proposal Towards Discovering Metamodels from Low-Code Application Platforms

Fernando Moreira[1] [a], Nuno Bettencourt[2] [b], Alexandre Bragança[1] [c] and Isabel Azevedo[3] [d]
[1] Institute of Engineering of Porto, Polytechnic of Porto, Portugal
[2] Interdisciplinary Studies Research Center, Institute of Engineering of Porto, Polytechnic of Porto, Portugal
[3] GILT Research Unit, Institute of Engineering of Porto, Polytechnic of Porto, Portugal

Keywords: Model Driven Engineering, Meta-Model Discovery, Low-Code Applications, Model to Model Transformations, Ecore, Mapping, REA.

Abstract: Low-Code platforms enable users to quickly create apps of various types, 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, restraining users to use the original platform. Thus, we investigated the feasibility of discovering these platforms' meta-models by using some exported models as a starting point. This possibility could enable apps to be migrated, for example, to a new version of the platform or a different one by describing transformations using the discovered meta-models. A proposal for solving this issue is described, also its evaluation. By analysing the obtained test results, the proposal was considered successful.

SCITEPRESS SCIENCE AND TECHNOLOGY PUBLICATIONS

1 INTRODUCTION

This article is part of the Business Application Modeling Language – Low-Code Platform (BAMoL) project, whose main goal is designing and implementing a Domain Specific Language (DSL) for the Low-Code Application Platform (LCAP) named Omnia.

This is a joint project between the team behind LCAP, Instituto Superior de Engenharia do Porto (ISEP), and Faculdade de Engenharia da Universidade do Porto (FEUP).

At the moment, two approaches for building the DSL are being considered. The traditional method of manually constructing the DSL is the first option. The second option is a technique that seeks to automate as much of the DSL generation as possible, using exported models from LCAP-based applications as a starting point. This work focused on contributions to the latter, particularly the generation of a meta-model from a set of models.

The interoperability of low-code platforms (Sahay et al., 2020) is lacking. For instance, it is impossible to reuse artefacts from one low-code development platform to another. Although this work's objective is to obtain a meta-model, a DSL would follow, allowing to achieve some level of interoperability.

The remainder of this paper is structured as follows. The next section describes the context and existent problem. Section 3 presents some related work. Section 4 proposes a solution for the enunciated problem, and Section 5 describes its evaluation. The paper concludes in Section 6, where some limitations and some work for the near future are also presented.

2 CONTEXT AND PROBLEM

LCAP is a term used to describe a class of applications that facilitate the development of other applications that require little to no code being written. Users of these platforms are typically called *citizen developers*.

At their core, these platforms implement Model-Driven Engineering (MDE) principles, allowing citizen developers to model the applications they are developing, as opposed to explicitly writing their code with a general-purpose programming language. Such a feature is enabled using advanced graphical user in-

[a] https://orcid.org/0000-0002-9268-2007
[b] https://orcid.org/0000-0003-1767-8240
[c] https://orcid.org/0000-0002-4882-9497
[d] https://orcid.org/0000-0003-2172-633X

Moreira, F., Bettencourt, N., Bragança, A. and Azevedo, I.
A Proposal Towards Discovering Metamodels from Low-Code Application Platforms.
DOI: 10.5220/0011688800003402
In *Proceedings of the 11th International Conference on Model-Based Software and Systems Engineering (MODELSWARD 2023)*, pages 168-175
ISBN: 978-989-758-633-0; ISSN: 2184-4348
Copyright © 2023 by SCITEPRESS – Science and Technology Publications, Lda. Under CC license (CC BY-NC-ND 4.0)

terfaces and visual abstractions (Sahay et al., 2020).

The Omnia Platform is an LCAP whose modelling follows the Resources, Events, Agents (REA) economic modelling system (Hruby, 2006).

However, LCAPs are typically closed-source solutions that do not adhere to standards, hence each individual platform probably has its own approach to designing its meta-model. Furthermore, due to their closed-source nature, usually the meta-model is not explicitly provided or properly versioned.

This issue results in the inability to migrate applications modelled with one LCAP to another, causing the user to become dependent on one platform.

Another issue that occasionally 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 in more recent versions since the migration mechanisms might not have been implemented.

Both issues challenge proper maintenance and evolution of applications. In a lot of cases, the only solution is to redo and remodel the application.

Since developing in an LCAP is, at its core, just creating and manipulating a model of the final application, each LCAP requires a meta-model that sets the boundaries of what is possible to develop through the platform.

The solution proposed by this article aims at discovering this meta-model from a set of existing models (*i.e.*, exported applications created in the target LCAP, represented in a serializable format, such as JavaScript Object Notation (JSON) or Extensible Markup Language (XML)). For the sake of simplicity, the approach depicted in this article only deals with models serialized in a JSON format. By applying this approach, the source and target meta-models can be constructed and then used to define transformation with the goal of, for example, migrating an application from a source LCAP to a target LCAP.

The main goal of this article is to test the hypothesis that it is possible to automatically deduce a meta-model for an existing application from its LCAP exported models.

The deduced metamodel must be precise enough to ensure the acceptance of agreeing models by LCAP. In other words, models instantiated by the deduced meta-model should be valid compared to the original one.

Moreover, it should be as complete as possible to cover the wide range of features that an LCAP can have. This means that every feature that exists in the initial models should also be covered in the deduced meta-model.

3 RELATED WORK

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

The authors (Cánovas Izquierdo and Cabot, 2013) demonstrate an approach to find the implicit domain model from a set of JSON-based services. To illustrate this, they use a Representational State Transfer (REST) Application Programming Interface (API) provided by a real case.

Their implementation consists of three stages: Pre-discovery; Single-Service Discoverer and Multi-service Discoverer.

In the Pre-discovery stage, JSON documents are transformed into models conforming to a JSON meta-model. During the Single-Service Discoverer stage, for each JSON service, the JSON models are transformed into `Ecore` models by applying defined mapping rules. Finally, the Multi-service Discoverer stage is performed. During this stage, all the `Ecore` models from the previous stage are merged according to a set of rules as well. The result is a model that is as close as possible to the complete domain model of the set of JSON-based services.

The paper by (Bragança et al., 2021) describes the experience of using DSLs to add support for a Software Product Line (SPL) engineering approach on an LCAP. For that purpose, a DSL capable of representing all the LCAPs modeling concepts is developed: the Low-Code DSL (LC-DSL).

Since LCAPs are usually closed source and do not provide access to the underlying meta-model but do provide ways for exporting models, a method was devised that allowed the meta-model to be deduced from exported models. This meta-model is used as input to create the LC-DSL.

The LC-DSL is then used to model groups of LCAP applications that can be managed as a SPL, in other words, the LC-DSL is a model that covers all LCAP applications that can be managed as a SPL.

The work includes an import/export process to obtain the LC-DSL from the models in the LCAP (and vice versa), to integrate the existing LCAP with the new LC-DSL.

Other works like (Javed et al., 2008) or (Zolotas et al., 2019) also exist, but their applicability in the context of this work is not clear.

4 PROPOSAL

A Model to Model (M2M) Transformation was fundamental for the approach. This section describes the source (LC-Metamodel) and the target meta-models

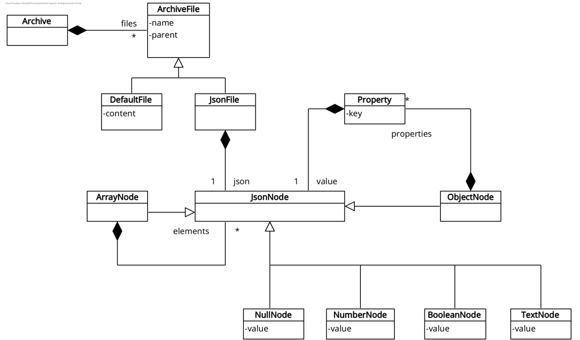

Figure 1: Class diagram of the LC-Metamodel.

(LC-MetametaModel). Then it presents the existing mapping workflow, detailing the Basic Structure Inference and Reference Deduction stages.

4.1 LC-Metamodel

The source meta-model is the **meta-model of LC-Model**, which shall be referred to as **LC-Metamodel**. Its class diagram can be found in Figure 1 and the description of its elements is as follows.

An **Archive** is a collection of **ArchiveFiles**. An **ArchiveFile** represents a file. Each **ArchiveFile** is identified by a `name` and a `parent path`, which is relative to the root of the archive. When it contains JSON it is a **JsonFile**, otherwise a **DefaultFile**. Both types of files are extensions and specifications of an **ArchiveFile**. While the first, a **DefaultFile** represents a generic file by adding the file's contents on the `content` property, the latter, a **JsonFile** represents a JSON file and adds a reference to a **JsonNode**.

A **JsonNode** represents a JSON value that can be specialized as: an **ArrayNode**; an **ObjectNode**; a **TextNode**; a **BooleanNode**; a **NumberNode**; or a **NullNode**;

Each **TextNode**, **NumberNode**, **BooleanNode** and **NullNode** represent a JSON primitive value, whose type depends on the instance.

An **ArrayNode** represents a JSON array and has the `elements` property, which supports a collection of **JsonNodes**. An **ObjectNode** represents a JSON object and has one or more **properties**, which are **Key-ValuePairs**.

A **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**.

4.2 LC-MetametaModel

The target meta-model is the **meta-model of Metamodel**. Since it is a meta-model of a meta-model, the target meta-model is a **meta-metamodel for all LCAPs**, referred from now on as **LC-MetametaModel**.

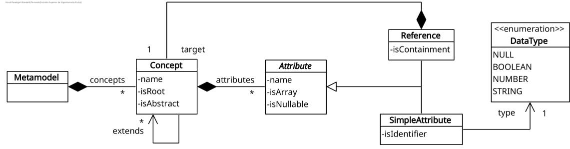

Figure 2: Class diagram portraying the target meta-model.

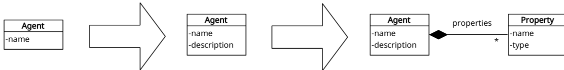

Figure 3: Evolution of a meta-model as more models are inspected, represented using a Class Diagram.

Figure 2 presents a class diagram representing the LC-MetametaModel. Its elements are described as follows.

The **Metamodel** is a collection of **Concepts**.

Each **Concept** is essentially a collection of **Attributes**, identified by a `name`, and described by two boolean properties `isRoot` and `isAbstract`.

Each **Attribute** represents a **name/value** pair. It is identified by a `name` and has two boolean properties, `isArray`, and `isNullable`. Attributes can be further specialized by the **SimpleAttibute** and **Reference**. A **Simple Attribute** represents a primitive attribute, *i.e.*, its value is either `null`, `boolean`, `number` or `string`. Each **Reference** represents 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.

4.3 Workflow

The working principle of this algorithm is gradually building the meta-model with each inspected model, adding, or updating Concepts when new information is found.

This is illustrated by Figure 3, 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. This workflow has two major stages.

On the first stage (**Basic Structure Inference**), the goal 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, in-

cluding their fields, respective types, and other nested data structures. In MDE terms, this stage can be seen as a **M2M transformation**.

On the second stage (**Reference Deduction**), the goal is to identify which simple attributes are in fact references to a concept. In MDE terms, this step is a model refinement.

4.3.1 Basic Structure Inference

This stage consists in iterating over all the JSON files and recursively apply a set of rules that are further described.

The first rule states that 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**, 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 property flag `isAbstract` is always set to `false`, since the fact that a `JsonObject` exists means the concept cannot be abstract.

The second rule states that an **ArrayNode** is mapped as an **Attribute**. Each element of the **ArrayNode** is mapped to an **Attribute** according to the rules defined for that specific element and then the resulting **Attributes** are merged/reduced into a single **Attribute**, which property `isArray` is set to `true`.

The third rule states that a **Property** pair is mapped to an **Attribute**. The name of the resulting **Attribute** is the key ti the pair and the `isIdentifier` property is set to `true` if the name is equal to the identifier name and set to `false` if otherwise. The **isArray** property in the resulting **Attribute** is set to `false` if no other mapping rule is applied. If the value is a primitive node (*e.g.*, **NullNode**, **NumberNode**, **BooleanNode** or **TextNode**), then the resulting **Attribute** is a **SimpleAttribute** with the respective data type. If the value is `null`, then the `isNullable` property is set to `true`. Otherwise, it is set to `false`. If the value is an **ObjectNode**, then the resulting **Attribute** is a **Reference** with the `isContaiment` property set to `true` and the **ObjectNode** is mapped using its specific rules.

For example, take the input model shown on Listing 1 and the resulting meta-model instance repre-

Listing 1: JSON Model.

```
{
    "name": "Company",
    "attributes": [
        {
            "name": "code",
            "type" "Text"
        }
    ]
}
```

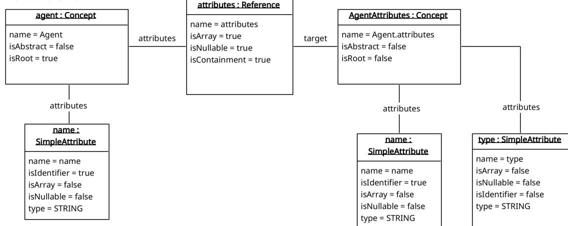

Figure 4: New Concept.

sented using an object diagram that can be seen in Figure 4.

The input model was 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 is a top-level/root object, the `name` of its respective **Concept** is **Agent**, since it is the name of the directory where the file is located, and its `isRoot` property 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` property is set to `true`, since the word `name` is assumed to be the designation for the identifier properties. The value is a single value, and it is not `null` so the `isArray` and `isNullable` properties 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` property is the resulting **Concept** (mapped from the singular **ObjectNode**) and both its `isContainment` and `isArray` properties 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 property value property, its `isRoot` property is set to `false`. Like the previous object, the **Properties** of the **ObjectNode** are then mapped to their respective **Attributes** in the **Concept**.

SCITEPRESS SCIENCE AND TECHNOLOGY PUBLICATIONS

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** or **Attribute**, according to some predefined rules.

When **merging a Concept**: new attributes are added to the existing **Concept** and existing attributes are merged.

When **merging an Attribute**, if the new **Attribute** is a **Reference** and the previous is a **SimpleAttribute**, it upgrades it to a **Reference**. This feature is useful for the **Reference Deduction** stage. For the boolean properties, 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 broader type between the old and new type. The types are order from broader to narrower accordingly: string -> number -> boolean -> null.

4.3.2 Reference Deduction

The goal of this stage is to determine which simple attributes are, in fact, a reference to another concept. As an example, 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) contain all the values in values(A), *i.e.*, values(A) are 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 would not be able to find those kinds of references. One way to deal with this issue is to predict what the abstract concept could be.

In the context of metamodel inference, 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 $values(A) \subset names(C)$ hold true.

To find a **Concept** hierarchy, for each **Concept** C let *attributes(C)* be the set which holds all the names of all the attributes belonging to **Concept** C. Then, for all the **Concepts** found in the 4.3.1 stage, 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 hold 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) is chosen. This stage of the algorithm can be summed in the following specific steps.

1. Find possible abstract **Concepts** by finding common attributes in the existing **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** is chosen, upgrade it to a **Reference**, with the selected **Concept** as its `target` and the `containment` properties set to `false`.

5 EVALUATION

In this section we discuss what guided the assessment, the tools used and how, and the results obtained.

5.1 Evaluation Criteria

Besides the fulfilling of the requirements, some evaluation measures are used to validate the hypothesis, namely: (i) using the generated meta-model, it should be able to instantiate the same models that were used for deduction; (ii) instances of the generated meta–model should be able to be imported into the LCAP; (iii) the coverage of the features of the input models.

5.2 Test Tools

Four tools (*i.e.*, Input Model Validator, Generated Model Validator, Metamodel Validator, and Model Generators) were used for testing the proposal. Their usage and procedures are explained in this section.

5.2.1 Input Model Validator

The goal of this tool is to validate 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

Listing 2: Omnia meta-model.

```
[{
    "Name": "Agent",
    "Description": "...",
    "Properties":[
    {
        "Name": "Name",
        "Description": "The name of the
            entity (unique identifier)
            .",
        "TypeKind": "Primitive",
        "TypeName": "Text",
        ...
    },
    {
        "Name": "Description",
        "Description": "The textual
           explanation of the entities
           ' purpose.",
        "TypeKind": "Primitive",
        "TypeName": "Text",
        ...
    }]
}...]
```

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 semantic relationships between **Concepts/Attributes** of the meta-model.

5.2.2 Generated Model Validator

This tool is vendor-specific, and part of the Omnia platform being used for this proposal. It is accessible through the platform's REST API, whose documentation can be found at the Omnia's Swagger UI webpage[1], or through the platform's modeller. To evaluate the validity of the models, both features can be used to import a model into the Omnia platform. This assumes that the model is checked for its validity, meaning that if the process is successful, then the model is valid.

5.2.3 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 (cf. Figure 2)[2].

Figure 5 shows the meta-metamodel that the Omnia meta-model follows. Therefore, the work performed by this tool is to compare Omnia's meta-

[1]https://platform.omnialowcode.com/api/docs/index.html

[2]https://github.com/OMNIALowCode/omnia3/tree/master/docs/pages/omnia3/Languages

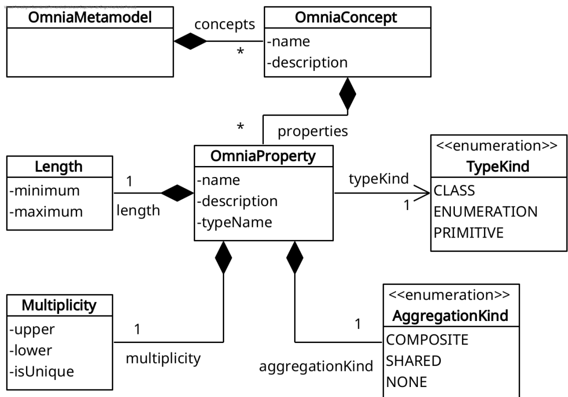

Figure 5: Omnia Meta-metamodel.

model found in GitHub with the one that was inferred (cf. Figure 2), particularly comparing analogous features, such as **OmniaConcepts** and the inferred **Concepts**.

The algorithm behind this tool will essentially check if for each of the deduced **Concepts**, there is also a **matching OmniaConcept**, *i.e.*, a **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 **OmniaConcept** 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` property is set to `true`, the `aggregationKind` of the **OmniaProperty** should be `COMPOSITE`.
  - The Reference target should match the **OmniaProperty's** `typeName` value.

5.2.4 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` property is set to **true**, the generated value shall be an array and its parametrized length.
- If the **Attribute** is a **SimpleAttribute**:
  - If the `name` of the **Attribute** is `lower`, `upper`, `aggregationKind` or `type` it will 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 considered.
- If the **Attribute** is a **Reference**:
  - If the `isContaiment` property 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` property is set to `false`, the value of the key/value pair is the name of one of the generated instances of the Concept.

5.3 Results

Three models were exported from the Omnia platform and used as input for the meta-model inference.

Using the “Metamodel Validator” tool presented in Subsection 5.2.3, the data shown in Table 1 was obtained. All the percentages are rounded to the nearest decimal unit.

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**

The **Not matched Concepts** is a list of deduced **Concepts** that did not have a matching **OmniaConcept**.

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 1, all the **Concepts** that were present in the input models were discovered 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, since the input models contain all the **Attributes**, for each of the **Concepts** they contain.

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 the references are correct.

6 CONCLUSION

It is a complex challenge to infer a meta-model from a collection of models. This works depicts the problem related to migrating models from one version of a Low-Code Development Platforms to an updated one or even to another platform. It presents a proposal for partially fixing the problem by exposing a detailed process explained through different stages.

Table 1: Meta-model validation results.

Input 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 procedures for testing the proposal are described. By comparing the initial goals with the obtained results, it is possible to conclude that the generated meta-model covers all the structural features of the input model. Furthermore, it is possible to suggest potential references which are, for the most part, accurate.

While it is undoubtedly possible and generally successful to infer a meta-structure, it is an entirely different matter when attempting to infer the implicit relationships between its elements and their constraints. Yet, with the envisaged proposal it is not possible to determine whether a field's value is dictated by another field or follows a pattern.

Moreover, there are some parts of this work that could be improved by adding a mechanism to predict the identifier's attribute name and removing the need for it to be provided as a parameter, therefore automating the meta-model discovery process.

Additionally, features that allow for manual changes in the generated meta-model could be implemented (*e.g.*, adding constraints; adding, updating, and removing elements).

The evaluation of this approach using models from various LCAPs to further ensure its adaptability and broad applicability will begin shortly.

Discussing "counteracting vendor lock-in" (Di Ruscio et al., 2022), Di Ruscio et al. refer to the inability to export development artefacts enabling their importing into other Low-Code Development Platforms. This paper discussed a possible first step towards this ambitious and complex goal, particularly important when initiatives aimed at any regulation in the sector are missing.

ACKNOWLEDGEMENTS

This work is supported by "Fundo Europeu de Desenvolvimento Regional (FEDER)" funds through the "Programa Operacional Competitividade e Internacionalização and Portugal2020" program, under the project BAMoL Low-Code Platform and the consortium BAMoL – LCP (POCI-01-0247-FEDER-39661).

REFERENCES

Bragança, A., Azevedo, I., Bettencourt, N., Morais, C., Teixeira, D., and Caetano, D. (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*, pages 16–28. ACM.

Cánovas Izquierdo, J. L. and Cabot, J. (2013). Discovering implicit schemas in json data. In *International Conference on Web Engineering*, pages 68–83. Springer.

Di Ruscio, D., Kolovos, D., de Lara, J., Pierantonio, A., Tisi, M., and Wimmer, M. (2022). Low-code development and model-driven engineering: Two sides of the same coin? *Software and Systems Modeling*, 21(2):437–446.

Hruby, P. (2006). *Model-driven design using business patterns*. Springer Science & Business Media.

Javed, F., Mernik, M., Gray, J., and Bryant, B. R. (2008). MARS: A metamodel recovery system using grammar inference. 50(9):948–968.

Sahay, A., Indamutsa, A., Di Ruscio, D., and Pierantonio, A. (2020). Supporting the understanding and comparison of low-code development platforms. In *2020 46th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, pages 171–178. IEEE.

Zolotas, A., Matragkas, N., Devlin, S., Kolovos, D. S., and Paige, R. F. (2019). Type inference in flexible model-driven engineering using classification algorithms. 18(1):345–366.