Exploring flexible models in agile MDE

Artur Boronat

School of Computing and Mathematical Sciences, University of Leicester, UK

Abstract

Semi-structured data has become increasingly popular in data science and agile software development due to its ability to handle a wide variety of data formats, which is particularly important in data lakes where raw data is often semi-structured or ambiguous. Model-driven engineering (MDE) tools can provide a high-level, abstract representation of a system or process, making it easier to understand and navigate data. However, relying on data models to describe metadata of raw data can create challenges when working with semi-structured data, which can contain errors, inconsistencies, and missing data.

In this work, we present a pragmatic approach to data-centric application development using MDE that complements current MDE practices. Our approach uses flexible models to enable agility and adaptability in working with data, compared to traditional metamodel-based methods. We propose a new metamodel for characterizing such models, with the aim of enabling the development of data-centric applications that do not require an explicit schema or metamodel. Our work demonstrates the feasibility of working with flexible models in a wide range of model to model transformation languages, particularly when handling semi-structured data sources.

Keywords

EMF, model transformation, flexible MDE

1. Introduction

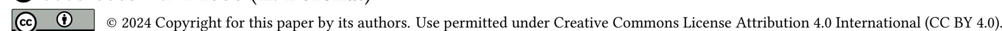
The increasing popularity of semi-structured data in agile software development and data science projects is due to its ability to handle a wide variety of data formats and heterogenous data. This is particularly important in data lakes, where raw data is often semi-structured or ambiguous, requiring data wrangling processes to infer metadata to aid in data analysis [1].

Model-driven engineering (MDE) is a software development approach that uses software models as a central artifact to capture the structure and behavior of systems. In the case of data, a software model can represent its structure, including the attributes, relationships, and constraints between them. MDE tools provide a high-level, abstract representation of a system or process, which can make it much easier to understand and navigate the data. This is especially important when working with semi-structured data, which can be highly variable and may not conform to a strict schema or data model. MDE tools also provide a more formal, rigorous approach to software development, with a focus on model validation and consistency checking. This can be helpful when working with semi-structured data, which can contain errors, inconsistencies and missing data.

However, such reliance on data models to describe the meta-data of raw data also hinders the application of MDE tools in this context, since the lack of such models is, precisely, what

Agile MDE 2024, co-located with STAF 2024, 08–11 July 2024, Enschede, Netherlands

artur.boronat@leicester.ac.uk (A. Boronat)
0000-0003-2024-1736 (A. Boronat)



© 2024 Copyright for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

CEUR-WS.org/Vol-3727/paper2.pdf

CEUR Workshop Proceedings ceur-ws.org ISSN 1613-0073

characterizes semi-structured data. This *rigidity* can create challenges in terms of modeling and integrating data from multiple sources [2]. Moreover, the majority of MDE practitioners who have been successful have either built their own modeling tools, made heavy adaptations of off-the-shelf tools, or spent a lot of time working around them. These adaptations introduce accidental complexity, which can add to the difficulties of using MDE tools [3].

In this work, we propose a pragmatic approach to data-centric application development using the Eclipse Modeling Framework. Our approach uses flexible models to enable agility and adaptability in working with data, compared to traditional metamodel-based methods. We propose a new metamodel for characterizing flexible models, with the aim of enabling the development of data-centric applications that do not require an explicit schema or metamodel. This approach allows for the loading of flexible models from a wide range of common data formats and can handle heterogeneous data. Our approach complements existing MDE approaches by focussing on data extraction and rapid prototyping, enabling their application in settings where a metamodel is not available or where data is simply too *messy*.

In this article, we start by explaining characteristics of semi-structured data and providing an outline of the model transformation languages used in section 2. We introduce the conceptual model of flexible metamodels in section 3. Next, we present a case study for analysing flexible models in section 4. Finally, we discuss related work in section 5, and conclude with a summary of our findings and directions for future research in section 6.

2. Background

This section provides an overview of two key areas: semi-structured data, which integrates elements of both structured and unstructured data, and model transformation languages (MTLs), which facilitate the mapping of input data to output data using defined rules and helper operations.

2.1. Semi-structured data

Semi-structured data combines elements of structure, like tags, with unstructured components such as free text, without adhering to a fixed schema like a metamodel. This flexibility is beneficial in fields like big data and data integration, enabling the handling of diverse data sources without schema constraints. Despite its versatility, semi-structured data lacks the rigour of structured data models, which facilitate processing, manipulation, and analysis using Model-Driven Engineering (MDE) techniques.

It is common to encounter datasets that are stored in semi-structured formats, such as CSV files. The dataset in Table 1 represents a collection of physical activity data recorded for a group of patients. The data includes information about the date, time, distance, intensity, air quality, air temperature, and heart rate of each physical activity performed by the patients. This is an example where the dataset consists of *plain* tuples, where there are no further relationships between fields of the tuple other than belonging to the tuple. The dataset contains examples of heterogeneous data, with intensity and air quality measured using different scales and variations in date format and units of measurement for distance and temperature. Additionally, inconsistencies exist in the formatting of the data, with air temperature sometimes not written

Patient ID	Date	Time	Distance	Intensity	Air Quality	Air Temp	Heart (bpm)
1	01/03/2023	10:00	2.5 miles	Moderate	Good	20␣C	72
1	01/03/2023	11:00	3.0 miles	High	Excellent	22C	78
—	1 March 2023	10:00	1.5 km	2	1	—	—
2	1 March 2023	11:00	2.0 km	—	3	68␣F	—

Table 1
Dataset with physical activity.

consistently and the blank space between figures and units occasionally missing. Furthermore, the dataset includes gaps that indicate a lack of data for certain fields.

2.2. Model transformation languages at a glance

This work employs two flavours of YAMTL (Yet Another Model Transformation Language) [4, 5], which incorporate the most common features available in MTLs like ATL, ETL or RubyTL. YAMTL can be used as an internal domain-specific language for model transformation within JVM languages. Xtend[1] is used to represent statically-typed MTLs and Groovy[2] is used to represent the dynamically-typed MTLs.

The MTLs define model transformations via transformation rules that map input objects to output objects. Input and output patterns can comprise multiple object patterns. An input object pattern specifies the object type and a filter (or guard) condition that must be met for applying a rule. An output object pattern creates an output object and is specified by the output object type and a sequence of attribute bindings that initialize the object's features (attributes and references). Furthermore, transformation rules may use additional helper attributes and operations that encapsulate logic to promote reuse. An attribute is a global variable whose value is reused across transformation rule applications, while an operation is called over a list of arguments to perform a computation.

In the statically typed MTL (YAMTL Xtend), operation and attribute helpers are called using the operation `fetch`, whereas in the dynamically typed MTL (YAMTL Groovy), the syntax is more flexible and the name of operation and attribute helpers can be used directly without the need for an explicit operator `fetch`. This operation is not needed in the dynamically typed MTL.

In both MTLs, the operation `o = fetch(i)` is also used to obtain an output object `o` that has already been transformed from an input object `i` using a model transformation rule. Such an operation is common in MTLs and corresponds to `resolveTemp()` in ATL and to `equivalent()` in ETL.

3. Flexible models

MTLs use metamodels to ensure that models are well-formed, and that models are semantically-correct wrt any formal OCL-like constraints of the metamodel. For example, EMF models require a metamodel, represented as an `Ecore` model. Even when model management tasks are

[1] https://eclipse.dev/Xtext/xtend/
[2] https://groovy-lang.org/

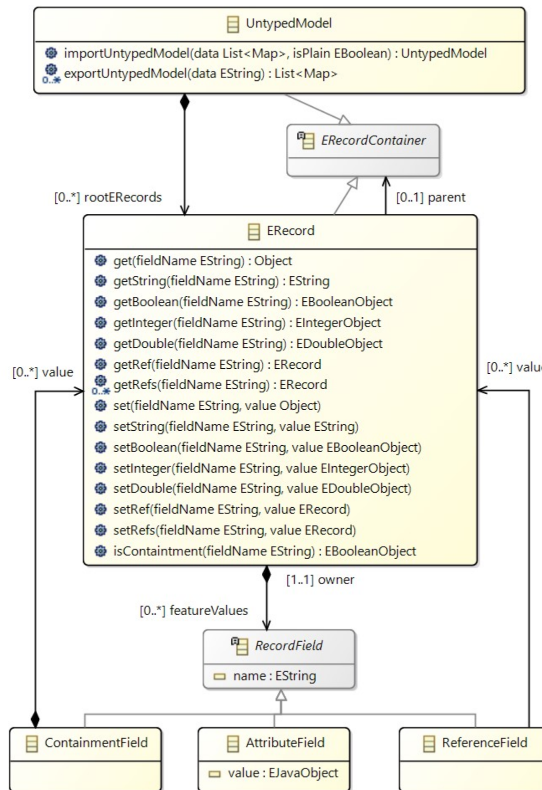


Figure 1: The metamodel of flexible models.

programmed generically using the EMF reflective API, the metamodel is still required to be able to access the EMF models.

In this work, we want to be able to work with EMF models independently of their metamodels, either from a statically typed context or from a dynamically typed context. This is achieved by means of the metamodel of flexible (untyped) models, shown in Figure 1, which defines a set of classes to represent the structure of models that do not have a fixed metamodel. We refer to this metamodel as the *flexible metamodel* in the remainder of the article. Semi-structured EMF models will be used to represent the type of semi-structured data shown in the previous section.

The `UntypedModel` class represents the overall model. The `ERecord` class represents an instance of a flexible (untyped) `EObject`, and the `RecordField` abstract class represents a field of such `EObject`, either an `AttributeField` containing a value of type `EJavaObject`[3], a `ReferenceField` that refers to another flexible `EObject`, or a `ContainmentField` that contains other `ERecord`'s, denoting hierarchical structures of objects in the model. Each `ERecord` points to its parent `ERecordContainer`, which is the `UntypedModel` instance for root records and another `ERecord` for non-root records. Attribute values can refer to collections of values. This metamodel captures the essential structure of models by segregating cross-reference references, which define graph structures in a model, from containment references, which define hierarchical structures of

[3]`EJavaObject` is the type use within an Ecore model to represent `java.lang.Object` in EMF.

objects in the model.

The `UntypedModel` and `ERecord` classes offer valuable functionalities that have been incorporated into the prototype, and their performance has been assessed in the case studies. The upcoming subsections provide a detailed explanation of these classes.

3.1. ERecord class

To declare a MT on flexible models in tools that require models to conform to metamodels, MTs need to consider the flexible metamodel in Figure 1. By using this metamodel and its model import `API`, tools can load semi-structured data from popular data formats, such as CSV, JSON, or XML, as flexible models.

The class `ERecord` offers a dynamically-typed interface that provides accessor and mutator methods that encapsulate its implementation. Accessor methods define how to retrieve field values. The accessor `get(fieldName)` returns the value of a field `fieldName` as a Java Object.

Since the flexible model operates with the absense of metamodel information, there is a mutator operation for each type of field, with *upsert semantics*. That is, whenever a mutator operation is called it either inserts the field value if not present in the `ERecord` and, if present, it overrides it. To create new `ERecords`, MDE tools built atop the EMF API can use the standard factory design pattern.

In the evaluation section, we have focused on the utilization of flexible models as input for MTs, exploring both the advantages and disadvantages of this approach.

3.2. UntypedModel class

The `UntypedModel` class serves as an intermediary abstraction layer between databinding libraries and model management APIs, specifically the EMF API in this study. This layer offers several design benefits:

- Firstly, it decouples the data sources from the model management API, enabling the extraction of semi-structured data from various heterogeneous sources in a unified way. This means that third-party object mappers can be used to produce lists of records from CSV, JSON, and XML serializations, among others.
- Secondly, it encapsulates design concerns that are commonly shared across various semi-structured data sources. For example, the ability to treat a dataset as a plain or hierarchical structure can be reused for different data formats, leading to a more modular and reusable design.
- Third, applications built using flexible models experience consistent performance across different data sources, as the flexible metamodel acts as a pivot language that hides the implementation databinding details. This uniformity is achieved by abstracting the specificities of each data format and providing a common interface, resulting in a more efficient and maintainable design.
- Lastly, for EMF-based tools, the notion of `ERecord` in flexible models enables interaction at an object-level without having to rely on extraneous third-party APIs.

Flexible models can be imported using two main methods. The first method is from semi-structured data that is represented as List<Map>, a format commonly obtained from widely used databinding libraries. The import service uses the List<Map> type in its signature, which allows third-party libraries to reuse this service with other formats. The transformation from List<Map> to a flexible model involves a linear traversal of the data structure, creating an ERecord for each map. The runtime complexity is $\mathcal{O}(n)$, where n is the number of maps in the list.

4. Case study: plain datasets

The goal of this case study is to demonstrate the effectiveness of using MDE tooling for transforming raw and semi-structured data into a more meaningful and structured model. We will be transforming a CSV dataset into an instance of the data model in Figure 2 that describes physical activities recorded for patients. The transformation normalizes the data into Patient's and PhysicalActivity's, filtering away some fields that are not relevant. In addition, free-form strings are converted into enumeration literals. In the resulting data model, it is easier to perform advanced analysis and gain insights into the data. The use of a data model also enables us to validate the data.

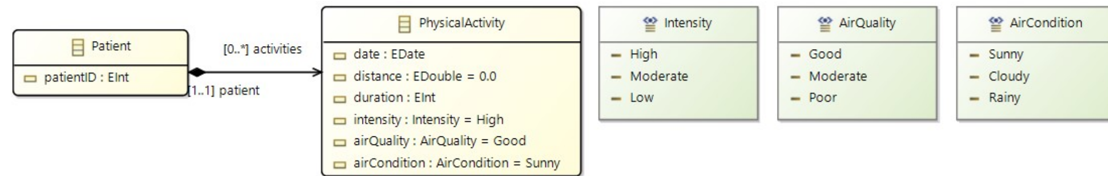


Figure 2: Simplified data model of a health tracker app.

This MT is used to analyze the overhead for loading semi-structured data and for assessing the flexibility that it provides in MT engines. The transformation CSV2PA is unidirectional and out-of-place, meaning that the source model is not altered during the transformation. The MT loads semi-structured data in CSV format and extracts data as instances of classes in the domain model of Figure 2. Datasets of different sizes along a logarithmic scale have been generated, and the sizes are reported in Table 2. The MT consists of one rule and one static operation, shown below. This transformation is defined using the dynamically typed MTL. Object variables may refer to objects matched by in pattern elements, e.g., r, and by out pattern elements, e.g., a.

In the MT CSV2PA, shown in the Listing 1, each row is translated into a PhysicalActivity instance, and each unique patient_id in a row gives rise to a Patient instance that is related to the PhysicalActivity instance corresponding to the row. The MT assumes that that each row of a dataset is available in memory as an ERecord. The rule matches each row in the dataset and creates Patient's and PhysicalActivities.

Missing data can be dealt with using filter conditions and data processing helpers. For example, we have decided that only those activities that correspond to a patient will be processed. The filter r.patient_id.isNotEmpty() performs this check. To initialize the structural features of a Patient, the transformation leverages the dynamic capabilities of the MTL, using the field name as the accessor. Data wrangling at the attribute level is delegated to helpers, which handle

CSV2PA	
size factor	**rows**
1	10^1
2	10^2
3	10^3
4	10^4
5	10^5
6	10^6

Table 2
Input dataset sizes (CSV dataset for CSV2PA).

Listing 1: Rule transforming an `ERecord` into a domain-specific class.

```
1  rule('Activity')
2      .in('r').filter { r.patient_id.isNotEmpty() }
3      .out('a', PA.physicalActivity) {
4          def p = getPatient(['patient_id' : r.patient_id])
5          p.activities += a
6          a.date = processDate(r.date, r.time)
7          a.distance = processDistance(r.distance)
8          a.intensity = processIntensity(r.intensity)
9          a.airCondition = processAirCondition(r.air_condition)
10         a.airQuality = processAirQuality(r.air_quality)
11     }
```

tasks such as checking null values, data splitting, analysis, and conversion. These helpers use common programming language features and require no further elaboration in this study. When data is missing helpers can be defined to supply a default value. Alternatively, a guard can be used to initialize the corresponding structural feature if the corresponding field `isNotEmpty()`.

The operation `getPatient` in the listing below creates a `Patient` instance for a given `patient_id`. This operation is called using the expression `getPatient([ 'patient_id' : r.patient_id ])`, which indicates the operation symbol `getPatient` and a list of parameter bindings `name : value`. Static operations are functional ensuring that is `patient_id` corresponds to one single `Patient` instance. In the body of the operation, the parameter bindings can be fetched by name, as declared in the invocation of the operation.

```
1  operation('getPatient') {
2      def p = PAFactory.createPatient()
3      p.patientID = Integer.valueOf(patient_id)
4      p
5  }
```

Notable syntactic advantages of the dynamically typed MT with respect to the statically-type one are that `ERecord` fields can be accessed using regular accessor methods and cast expressions are not required.

The MT CSV2PA has been executed over models that were generated using a size factor of N. The size factor N determines the number of rows in the models, such that the total number of rows is equal to 10^N. To generate values for the `patient_id` field, a random range between 0 and 10^{N-1} is used. As a result, each patient typically has about 10 associated activities.

In the CSV2PA case study, we established a performance baseline by evaluating the efficiency

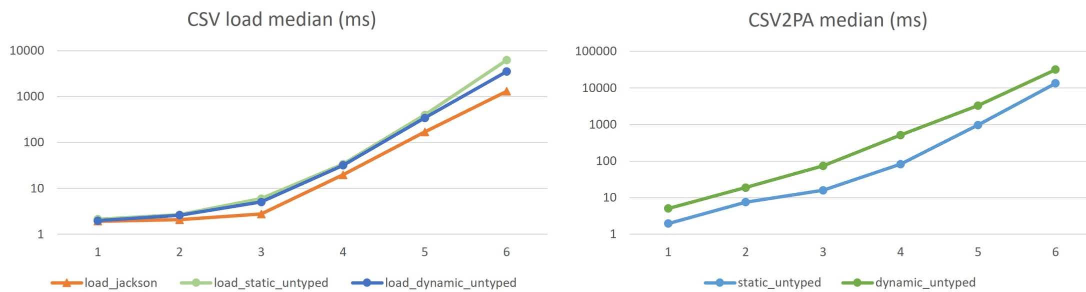


Figure 3: Experimental results of CSV2PA with size factor and execution time (ms).

of the Jackson library[4] for loading datasets in CSV format. To map the loaded records to untyped objects, we used the method `importUntypedModelFromCSV(path: String)`, which uses the Jackson library to parse the CSV file and then uses the method `importUntypedModel(data: List<Map>)`, from Figure 1, to create an untyped model in memory in the specified domain. This allowed us to quantify the overhead introduced by the mapping process. Figure 3 shows the baseline performance of the Jackson library to load the dataset in memory and the run-time of loading untyped models. The run time of loading untyped models includes the run time of loading CSV files with the Jackson library. The experiments show that datasets containing up to 10^5 examples can be loaded as untyped models in less than a second. The dataset with a million examples (53 MB) took around six seconds to load. The experimental results suggest that loading large untyped models is feasible and can be done in a reasonable amount of time. Furthermore, loading datasets as untyped models scales well relative to the load run-time with the Jackson library.

Figure 3 shows a slight overhead (measured in milliseconds) when transforming untyped models using dynamically-typed MTLs. This is due to the fact that accessor/mutator methods need to be resolved at run time. Nevertheless, the scalability of the transformation remains consistent with respect to the size of the input model, as compared to the transformation utilizing the CD metamodel. This suggests that using untyped models as input for MTs has minimal runtime penalties.

5. Related work

In recent years, there have been several approaches proposed for managing heterogeneous data sources for models. Below, we explore representative examples of metamodel-based approaches for dealing with heterogeneous data sources and flexible approaches that do not enforce the use of metamodels.

[4]https://github.com/FasterXML/jackson

5.1. Metamodel-based approaches

Epsilon [6] is a family of interoperable languages for model management built atop the Epsilon Object Language (EOL), an OCL-based imperative language that provides support for querying and manipulating models. Epsilon provides the Epsilon Model Connectivity (EMC)[7, 8] layer that offers a uniform interface for interacting with models of different modelling technologies, including semi-structured data formats like XML and CSV. EMC makes minimal assumptions about the structure and the organization of the underlying modelling technologies and it does not include concepts such as model element, type and metamodel. The main drivers for these design decisions are to facilitate future extensions, and to avoid degradation of run-time performance. However, each driver has to implement databinding and a load/save strategy that is specific for each data format/domain. This means that the performance of EOL, and Epsilon-based tools, is likely to be strongly influenced by the driver choice. Flexible models can be regarded as an intermediate abstraction layer decoupling serialization methods from model management APIs, where common tasks for processing semi-structured data, like inferring the hierarhical structure of semi-structured data, can be reused across data formats. Flexible models are EMF models themselves and the EMC layer is more abstract. On the other hand, once a flexible model has been imported the runtime performance of model transformation tasks is independent of the data source used.

Pinset [9] is a DSL that enables the extraction of CSV datasets from metamodel-based models in the Epsilon platform, facilitating concise scripts and domain-specific constructs, like aggregation, normalization and replacement of null values. Pinset is therefore used to specify transformations that are opposite to our case study. In our approach there are two options to implement similar tasks: 1) If the objective is to save all objects of a class as a CSV file, the solution involves using `importUntypeModelFromEmf`, a generic operation that translates any EMF model to a List<Map>, and then `exportUntypedModelToCsv`, which performs the reverse operation of `importUntypeModelFromCsv`; alternatively 2) if the goal is to perform data preprocessing, it can be defined as a transformation (MT) from a given metamodel to a flexible model, followed by `exportUntypedModelToCsv`. Using flexible models eliminates the need for a new metamodel or DSL for the intermediate results, potentially reducing accidental complexity.

NeoEMF [10] is a model persistence framework designed to address scalability issues associated with storing, querying, and transforming large and complex models. It provides a modular architecture enabling model storage in multiple data stores, including graph, key-value, and column databases. NeoEMF achieves scalability using a lazy-loading mechanism, which loads objects into memory only when they are accessed. It benefits from datastore optimizations and maintains only a small amount of elements in memory. NeoEMF is primarily designed for the persistence of metamodel-based models, possibly using data formats like JSON, but it does not provide support for importing semi-structured data.

5.2. Flexible MDE

Flexible or bottom-up MDE is an approach to domain and systems modeling that emphasizes the ability to adapt to changing requirements and to work with models that are not strictly defined by a pre-existing metamodel [11, 12, 13]. This line of research is based on the idea that

models should be able to evolve and adapt over time, rather than being fixed and rigid. This approach can be useful in situations where requirements are rapidly changing, or where domain experts are not familiar with formal modeling languages.

Flexmi [14] is a flexible and fuzzy parser algorithm for parsing XML documents into in-memory models that conform to EMF metamodels. The algorithm uses an approximate matching method to map XML element and attribute names to metamodel types and their structural features. This approach is designed to be more forgiving and can resolve inconsistencies between the structure and naming of metamodel elements and the contents of serialized models. Flexmi is a metamodel-independent approach and is not limited to any particular modeling language or metamodel. The main advantages of Flexmi are its flexibility and scalability. While the algorithm has been developed for XML, it should be easily portable to other data formats, like JSON and YAML. The algorithm scales linearly with the size of the XML document, and its overhead of flexible parsing is not prohibitive for practical use. One of the main limitations is that the current implementation does not allow for programmatically modifying and persisting models back into XML or other structured formats. Additionally, because Flexmi relies on fuzzy matching, there is a possibility that some mismatches may occur, leading to errors or inaccuracies in the parsed models.

FlexiMeta [15] is a dynamic meta-modeling framework that allows for the creation and manipulation of objects at runtime without the need for a pre-defined class or schema. It uses JavaScript, a prototype-based, weakly-typed programming language, to instantiate and serialize model elements. FlexiMeta also provides an optional code generation process to generate meta-objects from a metamodel, which improves model validation and conformance. The advantage of FlexiMeta is that its host language enables the use of models in web user interfaces. Flexible models provide a similar level of flexibility and the ability to import and export with EMF models, making typing optional.

6. Conclusions

While MDE provides a formal approach to software development, the lack of explicit schema or metamodel makes it challenging to work with semi-structured data. In this article, we introduce a conceptual model for semi-structured data in the form of flexible models that is independent of the EMF-based model transformation technology of choice. The new metamodel of flexible models enables the development of data-centric applications that do not require an explicit schema or metamodel. Our approach allows for the loading of flexible models from a wide range of common data formats and can handle heterogeneous data. From a design perspective, flexible models offer an intermediate abstraction layer between databinding libraries and model management APIs that helps encapsulating persistence logic that can be reused across semi-structured data sources.

There is potential for the current import mechanisms to be improved with more efficient loading strategies, such as parallelization, lazy loading, and incremental propagation of changes. The current prototype is not capable of exporting flexible models as metamodel-based models, as doing so would require reconciling data inconsistencies and missing data with the metamodel.

References

[1] I. G. Terrizzano, P. M. Schwarz, M. Roth, J. E. Colino, Data wrangling: The challenging journey from the wild to the lake, in: Conference on Innovative Data Systems Research, 2015.

[2] J. Whittle, J. E. Hutchinson, M. Rouncefield, H. Burden, R. Heldal, A taxonomy of tool-related issues affecting the adoption of model-driven engineering, Softw. Syst. Model. 16 (2017) 313–331.

[3] J. Whittle, J. E. Hutchinson, M. Rouncefield, H. Burden, R. Heldal, Industrial adoption of model-driven engineering: Are the tools really the problem?, in: MoDELS, volume 8107 of *LNCS*, Springer, 2013, pp. 1–17.

[4] A. Boronat, Expressive and efficient model transformation with an internal dsl of Xtend, in: MoDELS, ACM, 2018, pp. 78–88.

[5] A. Boronat, Incremental execution of rule-based model transformation, International Journal on Software Tools for Technology Transfer 1433-2787 (2020).

[6] R. F. Paige, D. S. Kolovos, L. M. Rose, N. Drivalos, F. A. C. Polack, The design of a conceptual framework and technical infrastructure for model management language engineering, in: ICECCS, IEEE Computer Society, 2009, pp. 162–171.

[7] D. S. Kolovos, L. M. Rose, J. R. Williams, N. D. Matragkas, R. F. Paige, A lightweight approach for managing XML documents with MDE languages, in: ECMFA, volume 7349 of *LNCS*, Springer, 2012, pp. 118–132.

[8] M. Francis, D. S. Kolovos, N. D. Matragkas, R. F. Paige, Adding spreadsheets to the MDE toolkit, in: MODELS, volume 8107 of *LNCS*, Springer, 2013, pp. 35–51.

[9] A. de la Vega, P. Sánchez, D. S. Kolovos, Pinset: A DSL for extracting datasets from models for data mining-based quality analysis, in: QUATIC, IEEE Computer Society, 2018, pp. 83–91.

[10] G. Daniel, G. Sunyé, A. Benelallam, M. Tisi, Y. Vernageau, A. Gómez, J. Cabot, NeoEMF: A multi-database model persistence framework for very large models, Sci. Comput. Program. 149 (2017) 9–14.

[11] J. Sánchez-Cuadrado, J. de Lara, E. Guerra, Bottom-up meta-modelling: An interactive approach, in: R. B. France, J. Kazmeier, R. Breu, C. Atkinson (Eds.), Model Driven Engineering Languages and Systems, Springer Berlin Heidelberg, Berlin, Heidelberg, 2012, pp. 3–19.

[12] H. Cho, J. Gray, E. Syriani, Creating visual domain-specific modeling languages from end-user demonstration, in: Proceedings of the 4th International Workshop on Modeling in Software Engineering, MiSE ’12, IEEE Press, 2012, p. 22–28.

[13] A. Zolotas, N. Matragkas, S. Devlin, D. S. Kolovos, R. F. Paige, Type inference in flexible model-driven engineering, in: Modelling Foundations and Applications, Springer, Cham, 2015, pp. 75–91.

[14] D. S. Kolovos, N. Matragkas, A. García-Domínguez, Towards flexible parsing of structured textual model representations, in: 2nd Workshop on Flexible MDE, CEUR-WS.org, 2016, pp. 22–31.

[15] N. Hili, A metamodeling framework for promoting flexibility and creativity over strict model conformance, in: FlexMDEMoDELS, 2016.