Baltic J. Modern Computing, Vol.10, No. 1, 17-35
https://doi.org/10.22364/bjmc.2022.10.1.02

Metamodel Specialisation based Tool Extension

Paulis BARZDINS[1], Audris KALNINS[1], Edgars CELMS[1],
Janis BARZDINS[1], Arturs SPROGIS[1], Mikus GRASMANIS[2],
Sergejs RIKACOVS[2], Guntis BARZDINS[2]

[1]Institute of Mathematics and Computer Science,University of Latvia, Raiņa bulvaris 29, Riga, LV-1459, Latvia
[2]Innovation Labs LETA, Latvia, Riga, Satekles iela 2b, Riga, LV 1050, Latvia

{paulis.barzdins, audris.kalnins, edgars.celms, janis.barzdins, arturs.sprogis}@lumii.lv,
{mikus.grasmanis, sergejs.rikacovs, guntis.barzdins} @leta.lv

Abstract. This paper outlines our Deep Learning Lifecycle Data Management system. It consists of two major parts: the LDM Core Tool – a simple data logging tool; and an Extension Mechanism – this mechanism allows the user to extend the simple LDM Core Tool to match their specific requirements. Current extensions support adding new visualisations for data stored on the server. Our approach allows the Core Tool to be a complete black box; we need only a metamodel denoting the logical structure of the stored data. By then specialising this metamodel we can define an Extension Metamodel which, when communicated to the tool through configuration, allows us to define and thus add the extensions.

Keywords: model driven architecture, metamodel specialisation, MLOps, software extension, deep learning

1. Introduction

The Deep Learning process is long and tedious, with many parameters and code versions and other features to keep track of. Therefore, we need tools for Deep Learning lifecycle data management (DL LDM) to streamline the process, allowing to focus on the project, not how to remember what combination of features gave which results. There are many tools available that attempt to solve this problem. We look at these tools in more detail in Section 2. The general conclusion is that they fall into one of two pitfalls – they are either missing vital functionality or grow incomprehensibly complex when trying to fit all use cases.

This is similar to the situation in the system modelling area. There the Universal Modeling Language (UML) was developed, and it was highly complex. Many UML tools (WEB, a; WEB, b; WEB, c; WEB, d; WEB, e; Streinberg et al., 2008; etc.) were developed which were very complicated as well. At the same time, most real tasks either needed a tiny subset of these many possibilities, or something was still missing for tasks in the given domain. As a result, in the world of system modeling the idea of Domain Specific Languages (DSL) and tools emerged. What was needed was not a single

universal tool, but a DSL tool building framework. By using such a framework an expert of the given problem domain could relatively easily build the required tool himself. Many commercial development environments contain such frameworks for DSL creation – Microsoft DSL Tools (Cook et al., 2007) and various Eclipse DSL tools, such as GMF (Gronback, 2009) and Obeo Designer (Sirius + Acceleo + EMF) (WEB, f), MetaCase (Kelly et al., 2008; Tolvanen et al., 2007). These frameworks typically create the relevant code (in C#, Java, etc.) from a DSL.

There are two ways to build such frameworks. One is to create a configuration language and implement it directly; this is complicated to do. The other is to use a model-based approach. For the relevant domain a metamodel is built, call it a Universal Metamodel (UMM), then the specific DSLs for this domain are obtained from the UMM by either instantiation (OMG, 2015; Barzdins et al., 2008; WEB, u) or specialisation (Kalnins et al., 2019). To get the tools themselves, the UMM needs an engine that would understand the DSL models. It would then either compile or interpret the models to get the corresponding DSL tool. There is also a third option, where the engine, upon receiving the DSL model, interprets it to then itself act as the respective DSL tool (Sprogis, 2016). This third option is the one used in this paper.

The main problem investigated in this paper is the development of DSL tool building technology for the DL LDM field. The first option for achieving this is adopting the same DSL path taken in the modeling field. But the DL LDM field differs significantly with the most important difference – in this field, any sane DSL tool has to offer the base necessities of communication between the data server and the workstations, and data storage on the data server. The parts that are changed and specified could then be different views and actions with the data stored on the data server. Thus, this field requires its own specific DSL tool building technology. The goal of this paper is the development of such technology.

First, we define a base LDM Core Tool, which implements the mentioned minimum necessities on the data server. We also define a LDM Core Library which facilitates the exchange of data between Workstations and the Data Server. The LDM Core Tool and LDM Core Library together we will call the LDM Core System. Section 3 describes this system; it already has practical uses since it satisfies the base necessities.

As a result, the DL LDM DSL tool building problem is reduced to the problem of building LDM Core System extensions. The main contributions of this paper are:

1) The development of an advanced LDM Core System extension definition method.
2) The development of the corresponding extension building framework.

These contributions are described in detail in Section 4; the following is an overview of techniques. First is the construction of a Universal Metamodel (UMM) for describing possible extensions (Section 4.2); then through metamodel specialisation (Section 4.1) we obtain models of specific extensions (Section 4.3); lastly an engine that is defined for the UMM (Section 4.4) can accept the extension specifications (Section 4.6) and then change its workings accordingly (Section 4.5).

The extension mechanism as described in authors' previous works (Celms et al., 2020a; Celms et al., 2020b) was based on partially revealing the inner data structure of the Core Tool; we had defined the Core Tool as a "grey box". But feedback from practical applications revealed that having the Core Tool be a grey box was inconvenient for the end user. In this paper we propose a different approach to extending the Core Tool – we now regard the tool as a complete black box (no longer minding its

implementation and data structure), instead we ask for the black box to understand the extension language that we will propose later in this paper. This way the black box is made "wiser" and as a result serves as an extension building framework named LDM Core Tool Plus (see Section 4.3 to 4.5).

2. Related Work

There are many tools available that attempt to solve the lifecycle data management task, but they fall into one of two pitfalls – they are either missing vital functionality (Miao et al., 2017; Bisong, 2019; WEB, g; WEB, h; WEB, i; WEB, j; Haifeng et al., 2019; etc.), or grow incomprehensibly complex when trying to fit all use cases (WEB, k; WEB, l). A more detailed comparison and analysis of these tools is available in papers (Celms et al., 2020a; Celms et al., 2020b; Barzdins et al., 2020a).

In the last couple of years, the existing tools for machine learning lifecycle support have become more mature, as well as new tools have emerged. The whole Machine Learning (ML) tools market has grown from general tools for Directed Acyclic Graph (DAG) execution, logging, and dashboards to a multibillion-dollar industry. The term MLOps has been coined. A lot of tools exist under this umbrella, many of them promising to cover the full spectrum of the ML lifecycle.

All-in-one tools promise a swiss knife but are usually expensive and often over-promise and under-deliver. Open-source tools offer solutions at zero cost but have maintenance and integration overheads. All of them lack in customisation possibilities if the out-of-the-box solution does not fully fit the needs.

From the cloud-based solutions prominent are SageMaker ToolKit (WEB, m) and Azure Machine Learning (WEB, n), which now also include the MLflow toolkit (WEB, o) which by itself is an open-source platform. They follow the platform-as-a-service paradigm and are tightly integrated with model training on their own clouds.

From stand-alone platforms which support machine learning lifecycle management for model training on-premises notable are Comet (WEB, l), Neptune (WEB, p), ClearML (WEB, q), Aim (WEB, r), PiniTree (Barzdins et al., 2020b) and cnvrg.io CORE (WEB, s). Most of them are commercial solutions containing an open-source subset. All of them support experiment and run tracking and some sort of dashboards for comparing and visualising experiment runs. Comet even supports custom visualisations, although it requires writing custom JavaScript code to extend the built-in classes. Neptune allows for easy creation of custom charts from the built-in chart types, visualising hyperparameters and metrics. ClearML can show Tensorboard visualisations as well as custom ones, created via Matplotlib and Seaborn. For Aim and cnvrg.io as well, some easy views can be created visually, but further customisation would involve digging into the source and writing some code. Finally, PiniTree is a graph database with universal low-level visualisation and API resembling the LDM Core Tool described in this paper, it was used for prototyping the described techniques.

3. LDM Core System

The LDM Core System is what realises the base necessities, described in the Introduction, which any tool in the DL LDM ought to offer – communication between workstations and the data server, as well as data storage on the data server. The

following subsections will explain the approach taken to achieve these base functionalities.

3.1. General Structure of the LDM Core System

The LDM Core System consists of the LDM Core Tool and LDM Core Library. These components and access to them are then split amongst three "machines", though they need not be physically separate ones (Fig. 1).

The LDM Core Tool is installed on a Data Server by the tool *Configurator*. This server then hosts the tool and stores uploaded data. The LDM Core Tool is minimalistic in functionality, but usable in practice.

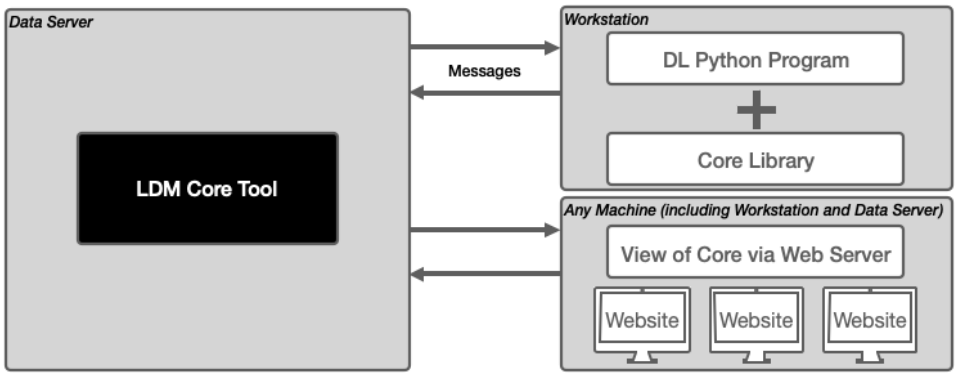

Fig. 1. General structure of the LDM Core System.

The Core Library is installed by the *Programmer* on his Workstation, where the DL program is run, and from which experiment data and results will be sent to the server.

Finally, there is the web access from any machine, which allows the *End-User* to view the experiment data and results. This could be the same or a different programmer looking to analyse or replicate work, or a client wanting to track results and progress.

3.2. LDM Core Library

The Workstation communicates with the Server using the following five library commands.

login(user_id, psw): a trial to authorise the user with user_ID using the password psw, in case of success returns the token_id
startRun(project_name): start new run in the project project_name
log(msg, role_name): store on DS the message msg and the corresponding role_name in the current run
uploadFile(file_name, role_name): upload the file file_name and the corresponding role_name in the current run
finishRun(): finish the current run

From a higher abstraction level, we can say that with these functions the Workstation sends the messages seen in Fig. 2 to the Data Server. "//" before the attribute name means that the value of this attribute is inserted by the LDM Core Library. "/" before the attribute name means that the value of this attribute is not sent from the Workstation but generated by the LDM Core Tool upon message reception.

The most important here for the content of this paper are commands Log and UploadFile; these are used to send data from the workstation to the server, data which extension programs will then visualise (explained in Section 4). The *log* function is more capable than it might appear, as the string message could be any JSON encoded data, thus encoding well-structured information of almost any scale.

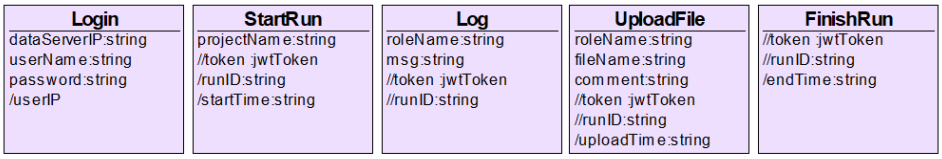

Fig. 2. Class diagram of messages.

3.3. LDM Core Tool

The messages and files sent by the end user are organised according to the structure shown in Fig. 3. It is a class diagram, named here the Core Logical Metamodel (Core LMM).

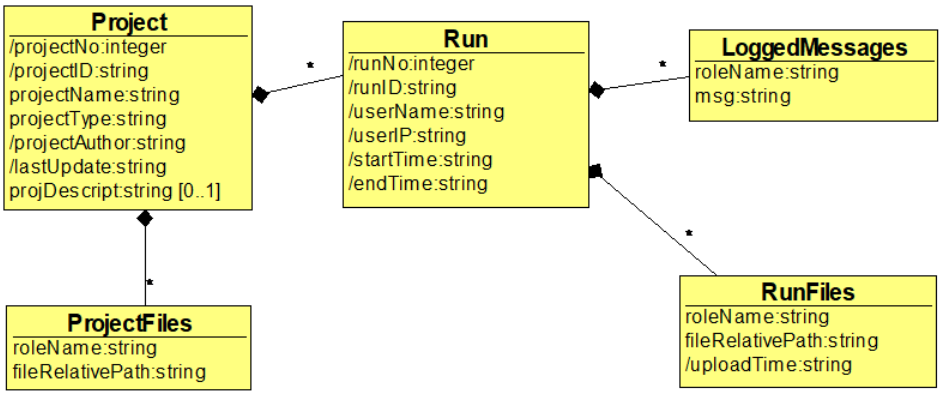

Fig. 3. Core Logical Metamodel.

It shows that the data is organised by Projects, each of which can have multiple ProjectFiles and Runs. Each Run can have its own RunFiles and LoggedMessages (the ones sent by the DL program execution with the commands Log and UploadFile). This model doesn't enforce anything for the server black box program itself, it can store this data in whatever structure the programmer decides. And for our later extension (in Section 4) we have no need to reveal the physical structure that the black box is actually using for storage, we just need the end user and the server program to be aware of this Core LMM.

Fig. 4 shows an instance of the Core LMM, to help understand the relation of projects and runs, as well as the files and logged messages. Fig. 5 shows this same instance as web pages; this is the view that the users accessing the LDM Core Tool via the web would see.

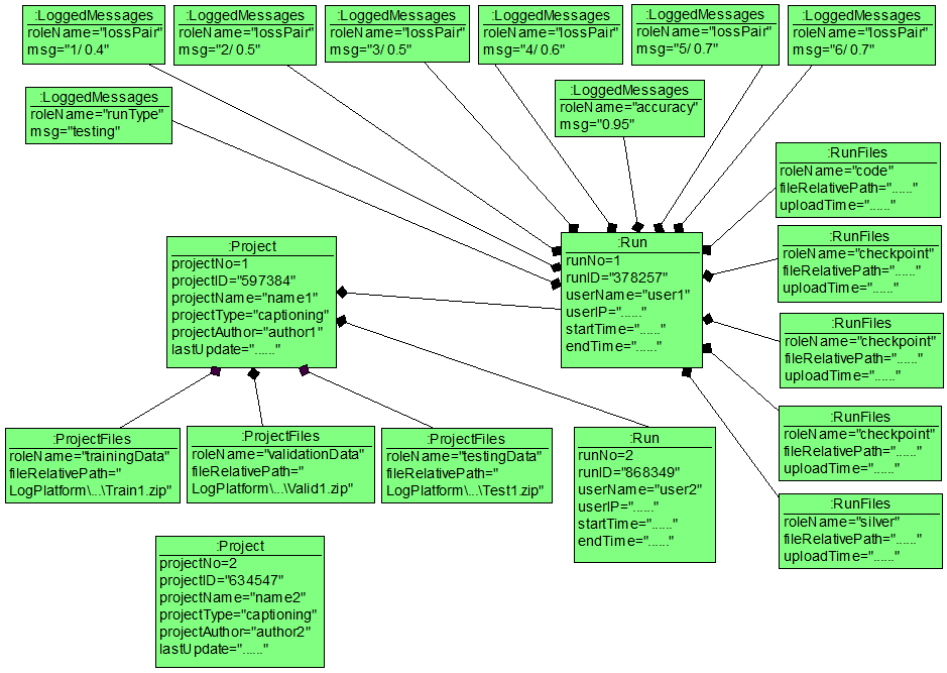

Fig. 4. Instance of the Core Logical Metamodel.

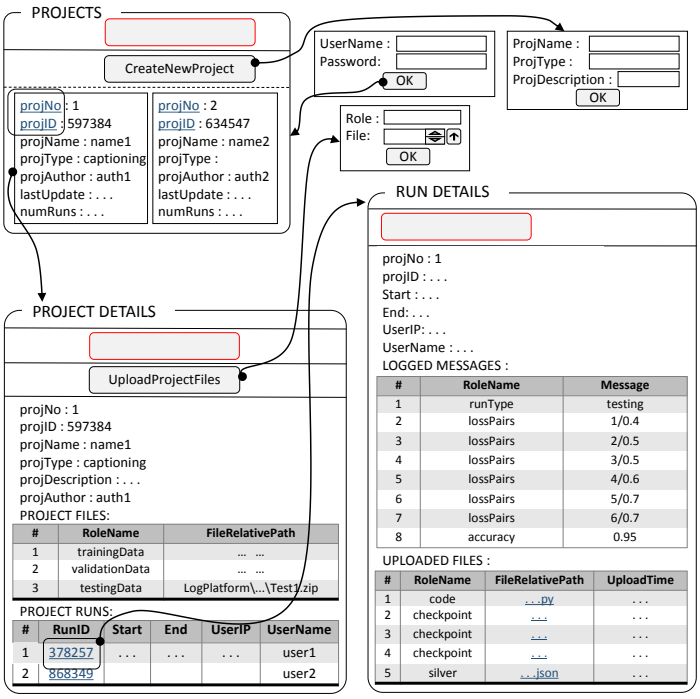

Fig. 5. Web views of the instance of the Core LMM.

The LDM Core Tool requires every user to log in, which is how it knows which projects to show to the user and allow them to edit. New projects are added via the web access with the "Create New Project" button. Project files are added in the "Project Details" view, using the "Upload Project Files" button. This would then ask the user to upload the file and give it a role-name.

Thus, the LDM Core System satisfies communication between workstations and the data server through the LDM Core Library, then the LDM Core Tool stores data on the data server, as well as allows web access to browse the data.

4. LDM Core System Extension Framework

So far, we've defined the LDM Core System, which by itself is a minimalistic but usable solution for DL LDM. What we will introduce now is an extension framework, which will build on top of this minimalistic tool to allow users to create specialised solutions for their DL LDM problems, with functionalities extended beyond the ones possessed by the pure LDM Core Tool.

When describing the extension framework, we only mention the LDM Core Tool, not the LDM Core Library. This is due to the library already being rather universal and unlikely to require new additions. Thus, the extension framework deals only with the Tool end – creating new visualisations for gathered data.

4.1. Metamodel Specialisation

The extension framework proposed in the following subsections will, to a great degree, be achieved through the medium of metamodel specialisation – an approach to DSL modeling tools introduced by the authors (Kalnins et al., 2016; Kalnins et al., 2019). The main idea of metamodel specialisation is that we first define the Universal Metamodel (UMM) for a domain and then for each use case in this domain define a Specialised Metamodel (SMM). The SMM contains a set of subclasses of the UMM classes, however many we need. The subclasses are defined according to UML rules, but with some restrictions. New fixed values may be assigned to class attributes, but new attributes may not be added. Similarly, for associations the role names may be redefined (as sub- role names) and multiplicities may be changed (shrunk). In our new domain we also allow attribute names and types to be redefined. If necessary, OCL expressions (OMG, 2014) can be used as well. Fig. 6 demonstrates a simple UMM example – a class diagram defining graphical diagrams.

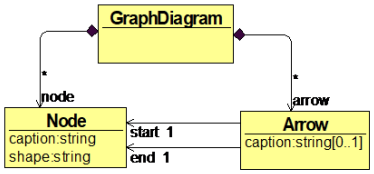

Fig. 6. UMM example.

This UMM is purposefully simplified, compared to metamodels used in practice, to make it easily understandable. In turn, Fig. 7 demonstrates a possible specialisation of this metamodel – a simple flowchart SMM. Fig. 7 also demonstrates the shorthand notation used for SMM presentation: class notation "Aaa {Bbb}" means that "Aaa" is a subclass of class "Bbb" (e.g., "Flowchart {GraphDiagram}"). Similarly, association notation "ccc {ddd}" means that association "ccc" is a subclass of association "ddd" (e.g., "condFlow {arrow}").

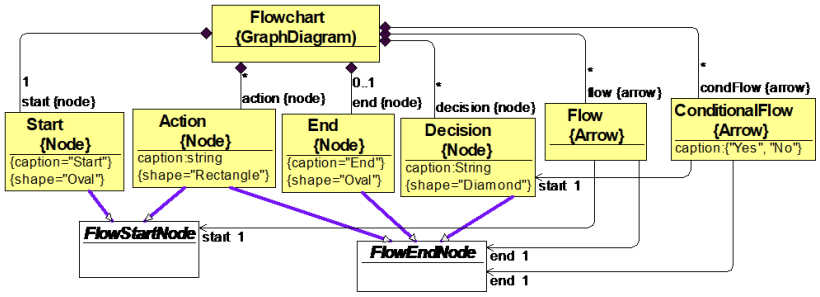

Fig. 7. SMM example.

However, in this paper, the notation of specialisation will be extended. In (Kalnins et al., 2019) it was assumed that the semantics of specialised classes are directly determined by their attribute values or a default attribute value set. We will call such specialisation – simple specialisation. But for this paper, we need a broader specialisation concept where

the semantics are determined by some additional information as well. Such specialisation will be called functional specialisation. In the context of this paper, additional custom programs will be invoked by the LDM Core Tool at appropriate times. Formally, the functional specialisation will only describe how such custom programs can be found, but semantically the program now influences the workings of the framework engine. Both simple and functional specialisation will be used in this paper.

4.2. First step towards the Extension Framework: Universal Metamodel

The first problem we must solve is how to precisely define the set of valid LDM Core Tool extensions. Within the context of this paper, we will limit these to extensions that create new visualisations for data stored on the server. One powerful method for doing this is with UML class diagrams. As mentioned in the Introduction, this was the path taken in the field of modeling DSL tools; only here, instead of the modeling field, we deal with the more complex DL LDM.

The main result of this section is the class diagram shown in Fig. 8. It precisely defines a wide set of valid extensions that the framework should support. Following DSL tradition, the diagram is named Universal Metamodel (UMM).This diagram consists of two parts – the light classes (in yellow) and the bold classes (in pink). The light classes precisely match the Core LMM from Fig. 3, but the additional bold classes are the most important ones, as they together with the relevant associations precisely describe the supported extensions for the LDM Core Tool. The Project and Run classes have new Tab classes associated with them (more precisely, when accounting for the cardinalities, every Project and Run instance can have multiple Tab instances associated with it). Every Tab then is associated with an Extension Program, the parameters of which can be files that are either Project Files or Run Files.

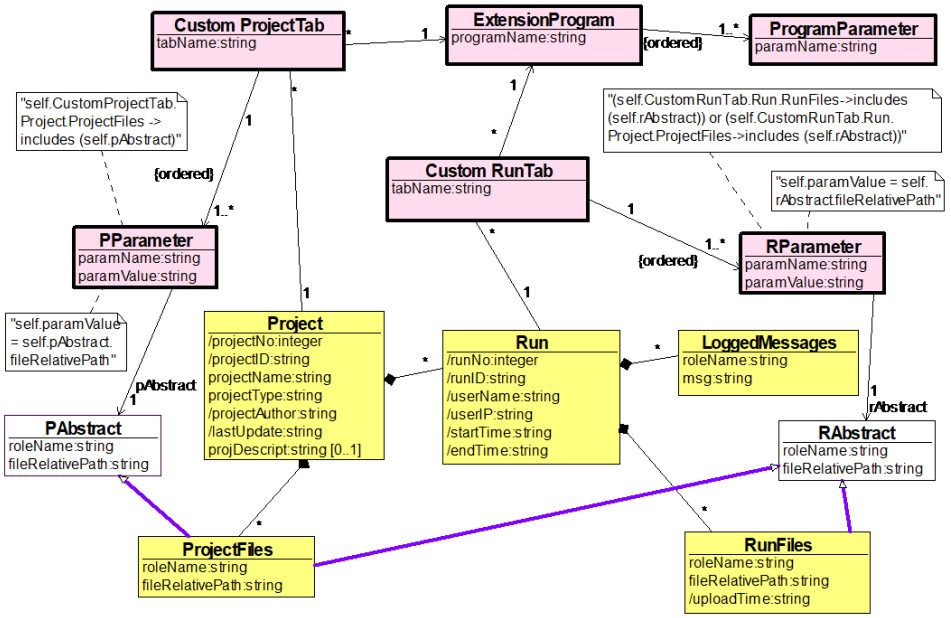

Fig. 8. Universal Metamodel (UMM)

To be precise, what is passed to the Tab are the file paths of these files (this can also be seen in the UMM). The associated Extension Program will lead to a web page that shows the results of running the program, thus serving new visualisations of the Core LMM instances for the end-user. What follows in Section 4.3 is the definition of specific extension models using metamodel specialisation.

4.3. From the UMM to LDM Core Tool Extension Metamodels

LDM Core Tool Extension Metamodels (EMM) are metamodels obtained from the UMM (in Fig. 8) by way of metamodel specialisation (described in Section 4.1). One such EMM example is shown in Fig. 9, but many other versions are possible for describing different extensions. The specialisation only affects the bold classes, as only those are relevant to the extension. The non-bold classes are the Core LMM classes and those will remain unchanged, as the base functionality of the LDM Core Tool needs to be maintained (the aforementioned base necessities). Fig. 9 shows an example of a LDM Core Tool EMM, which, besides the Core LMM in yellow, also includes the additional Run Tab titled TestGoldSilverView, together with its extension program and parameters.

Fig. 10 shows an instance of this metamodel. This instance corresponds to the instance of the Core LMM shown in Fig. 11. Fig. 10 differs from Fig. 11 with the addition of the Tab instances and their specific Extension Program and Parameters, as well as all the respective associations. By looking at the instances of Par1 and Par2, one can see that they now contain the specific file paths of the files they are associated with (e.g., Par2 contains the file path ending in Silver1.json, which is the same as the file path of the Silver instance it is associated with).

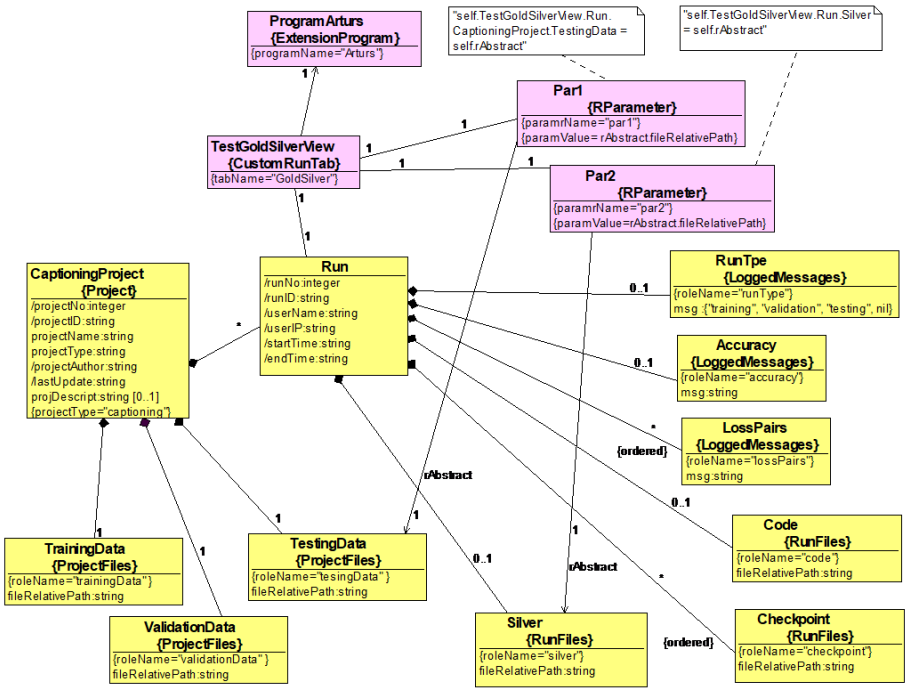

Fig. 9. Example of a LDM Core Tool Extension Metamodel

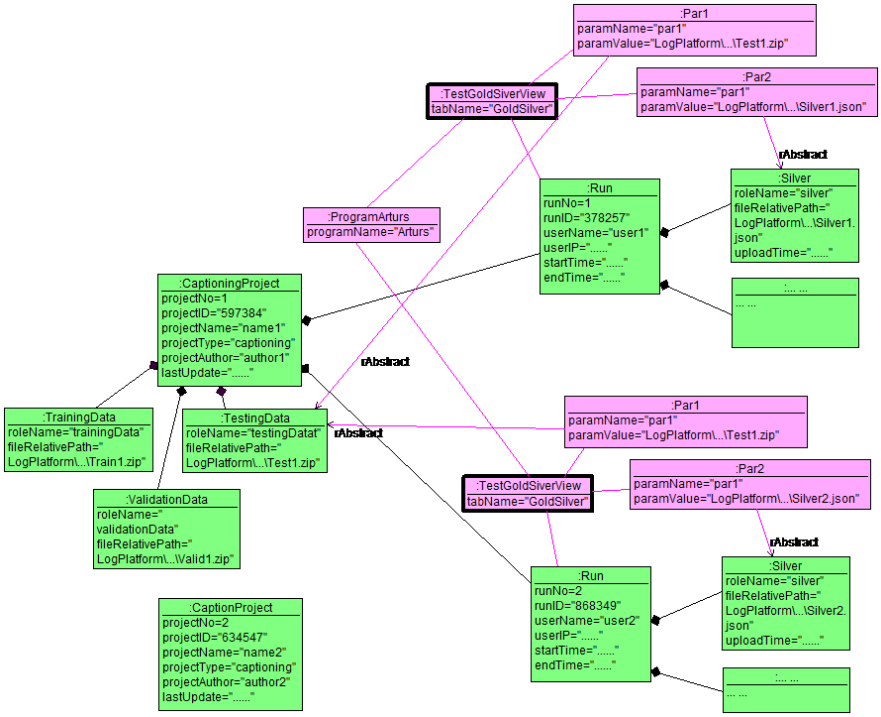

Fig. 10. Instance of the Extension Metamodel.

4.4. A basic remark concerning the building of the LDM Core System Extension Framework

Assume we are given a Core Tool Extension Metamodel (e.g., the one in Fig. 9) and an instance of the Core LMM (e.g., the one in Fig. 11) – is this knowledge sufficient to update the instance with the pink classes, which concern the extension (seen in Fig. 10)? The answer is yes! From Fig. 9 we see what Tabs are to be added to each project and run, as well as what Extension Programs they connect to (in our example a run tab TestGoldSilverView with the program Arturs). The non-trivial part is correctly updating the attributes and associations of the Parameter classes. For this both the EMM and the Core LMM instance are needed. From the EMM (in Fig. 9) we know which run and project files each parameter is associated with. But, vitally, the instance (in Fig. 11) has the precise file paths, which we add to the attributes of Parameter instances. Thus, we've managed to accurately update the instance with the extension classes.

The LDM Core System Extension Framework can now be defined a meta tool (named LDM Core Tool *Plus*), which upon receiving an Extension Metamodel (e.g., the one in Fig. 9) through configuration, starts to work as an extended LDM Core Tool according to the specifications of the received EMM.

What remains is only the implementation of the visualisation of the EMM instance (see Section 4.5), part of which requires calling the Extension Program, and communicating the EMM itself to the tool (see Section 4.6).

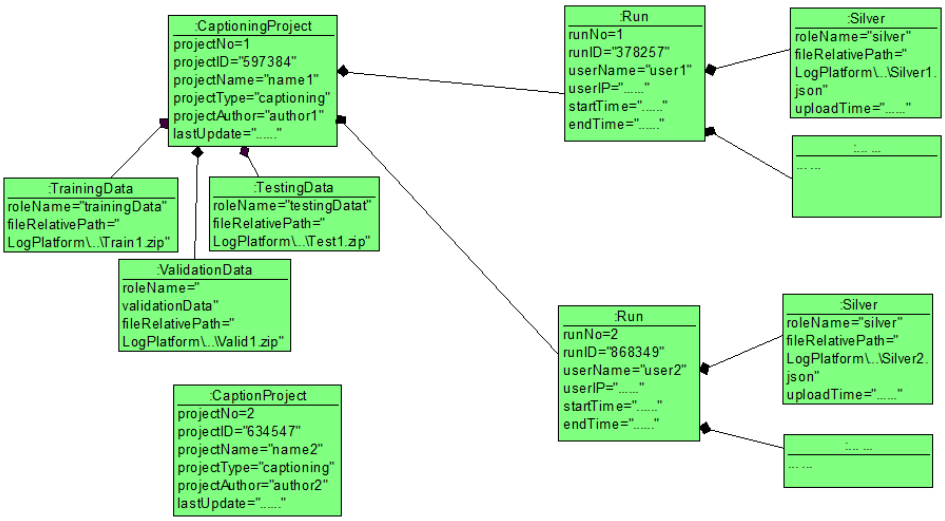

Fig. 11. Instance of the Core LMM (a shortened version of Fig. 4).

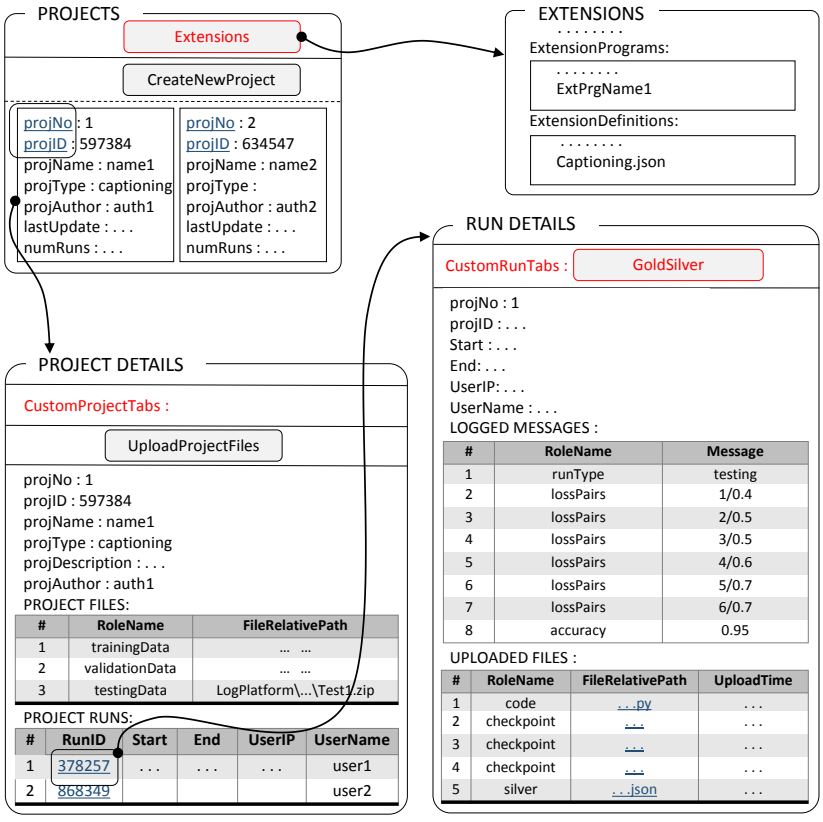

Fig. 12. Web views of instance of the EMM

4.5. LDM Core Tool Plus: Visualising the EMM instance

Fig. 12 shows the instance from the previous section (Fig. 10) displayed in the form of a web page, as it would be by the Plus component of the LDM Core Tool Plus. This is one of the main responsibilities of the Plus component. Compared to Fig. 5 the only visual difference is the addition of a single Tab (labelled TestGoldSilverView under Run Details). But this single tab plays a significant role, which also needs to be facilitated by the Plus component! A click on the tab must open a web page on which the result of the relevant Extension Program is shown when it is passed the specified attribute values.

Fig. 13 shows one possible way the web page resulting from executing the Extension Program might look. In this example the DL project is image captioning, and the tab displays the results of the program on the data sets. The tab takes as parameters the data sets, as well as the results sent to the server by the Workstation. Under the images the names of the flowers are in red if they were labelled incorrectly; the names in gold were labelled correctly.

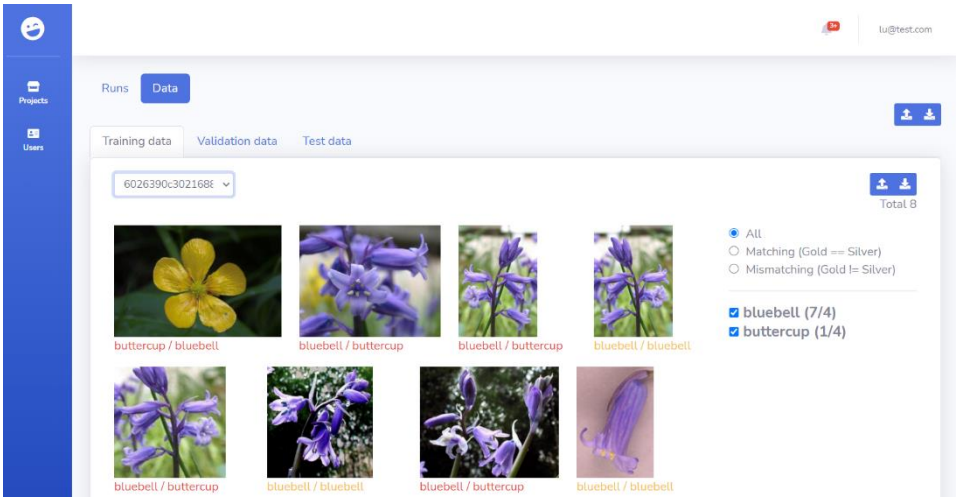

Fig. 13. Results of executed GoldSilver tab extension program.

4.6. LDM Core Tool Plus: Extension Configuration

For the LDM Core Tool Plus to work according to the chosen extension specification, we need a mechanism that facilitates passing this specification to the LDM Core Tool Plus. For this we need a configuration tab (Fig. 12 already has this tab, titled Extensions under the Projects view). As can be seen in Fig. 14, this tab leads to a web page with the option of entering configuration specification. Two things must be entered – the extension specification (named Extension Definition) and extension program (thus named).

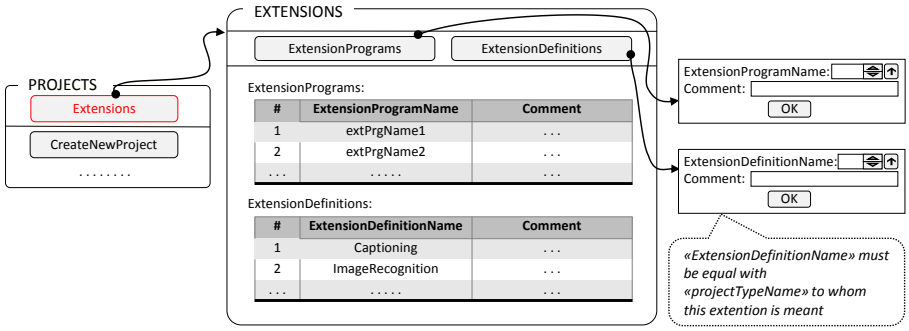

Fig. 14. Web views of the LDM Core Tool Plus extension configuration pages.

As mentioned before, the Extension Program the *Configurator* writes themselves, and uploads on this configuration page. But a question arises – how do we pass the Extension Specification to the LDM Core Tool Plus as we only have its graphical form? What is needed is to serialise the metamodel, which we can do by using JSON. Fig. 15 shows a metamodel serialised into the JSON format. This then is a JSON file which should be uploaded on the configuration page as the Extension Definition.

```
Captioning.json
---
{
projectType: Captioning,
projectFiles: [
  TrainingData,
  ValidationData,
  TestingData
],
runFiles: [
  Code,
  Checkpoint,
  Silver
],
runLogs: [
  RunType,
  Accuracy,
  LossPairs
],
projectTabs: [],
runTabs: [
  {
    tabName: GoldSilver,
    extensionProgram: Arturs,
    parameters: {
      Par1: {
        level: Project,
```

```
        rolename: TestingData
      },
      Par2: {
        level: Run,
        rolename: Silver
      }
    }
  }
]
}
```

Fig. 15. Specialised Metamodel serialised with JSON.

This example clearly shows how metamodels of this style can be serialised with the help of JSON. The UMM in a way acts as our JSON schema.

5. Discussion and Conclusion

The results of this paper can be looked at in two ways. The first is the viewpoint we used in the paper: how to build an easily extendable DL Lifecycle Data Management system. The second way to view the results is the Model Driven Architecture (MDA) viewpoint. The basic idea of MDA, according to (Kleppe, 2007), can be described with Fig. 16. PIM is the Platform Independent Model of the system; usually this model is a class diagram, and it is conceptually the same as what we called the Logical Metamodel of the system. Next is the Platform Specific Model (PSM) and finally the system Code. Currently we aren't concerned with the PSM, thus Fig. 16 is replaced by Fig. 17.

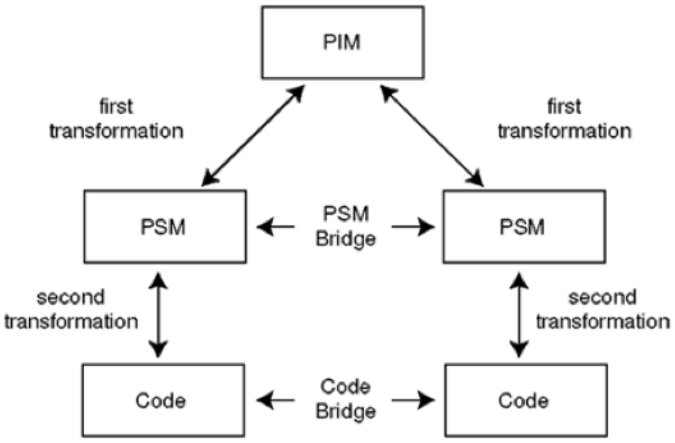

Fig. 16. MDA conceptual schema (Kleppe, 2007).

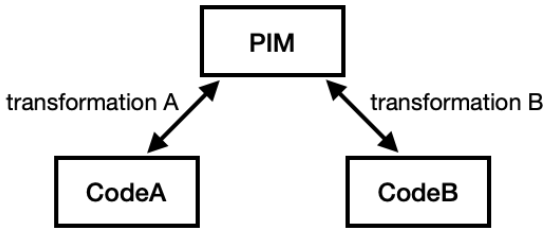

Fig. 17. Simplified MDA conceptual schema.

The extension idea offered in this paper is applicable to systems whose PIM models match the schema shown in Fig. 18.

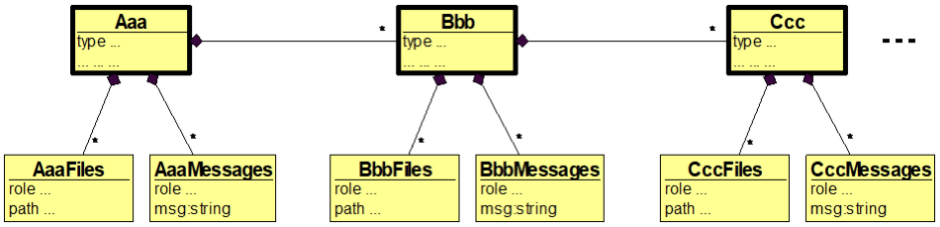

Fig. 18. PIM schema.

Classes Aaa, Bbb, Ccc will be called main classes (having bold frames in Fig. 18). The other classes we will call auxiliary classes (in Fig. 18 they are AaaFiles, AaaMessages, BbbFiles, etc.). Many data servers work according to such PIM models, including the DL LDM data server considered in this paper (Aaa corresponds to Project, Bbb to Run, Ccc could be Epoch). A hospital data server could also work according to such a PIM model, where Aaa corresponds to Patient, Bbb to Diagnosis, Ccc to Treatment.

In Fig. 19 we see the website schema which naturally follows from the PIM schema in Fig. 18, using which a user can view the contents of the data server. In this example CodeA is a program which manages the data server, supports both data input as well as the data visualisation according to Fig. 19.

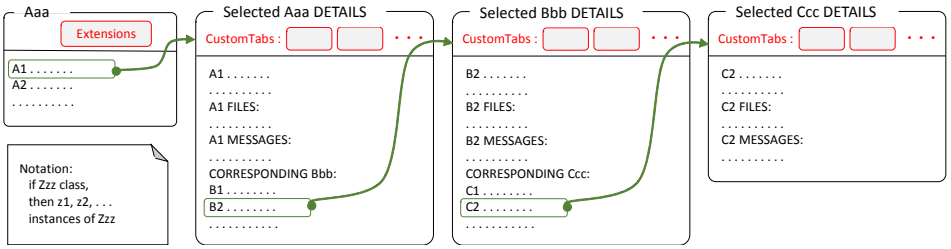

Fig. 19. Website schema.

The data input mechanism, thanks to the Role concept, seems to be sufficiently universal (from the server side) and therefore will not be discussed in more detail. However, the standard view of the server data according to Fig. 19 may be insufficient for use in specific domains (e.g., DL LDM). In this paper we have offered a sufficiently universal mechanism for defining new visualisations by using the Custom Tab concept. This mechanism consists of two parts. From one side there is a new program written by the tool configurator, which displays the new view from the supplied parameter values. From the other side CodeA Plus "understands" the offered language for tab definition, i.e., it understands which program is assigned to the given tab and what parameter values should be supplied to this program when we click on the given tab in Fig. 19.

In this paper we only described extensions that support visualisations, but this mechanism, with some additions, can handle more. For example, imagine a Custom Run Tab "Repeat Run"; in a dialogue it would ask if any hyper-parameters are to be changed, then execute the code of the run again with the new hyper-parameters. This is a much-sought functionality, offered by Facebook's Hydra (WEB, t). The new run would log as usual, as the executed code still contains all the logging commands from the Core Library, but getting to that point would require two things:

1) That the run log contains all required logs and files (can be specified with the Extension Metamodel).
2) That there is one or more workstations on which to execute the program.

A problem arises, where the tab could be clicked ten times in a row, each tab starting in a separate execution thread, but the workstations will not manage to keep up – a queue is needed. If the extension followed the often-used RabbitMQ approach, it would have to form a job, enter it into a queue, later a worker would pull the job from the queue, then even later the extension can receive back the finished job. The authors have developed a simpler queue implementation named "Token Queue", which mimics the real-life post office and similar places' system of taking a ticket and then waiting to be called. A dedicated ticket server is set up (this is what we call the Token Queue); all that is left for the extension to do is ask the Token Queue for a URL of a free worker, and then just wait on this query until a worker is free, and a URL is then returned. Then the extension simply executes the run on that worker and after it is done returns the URL back to the token queue, so that it can be served to the next in line.

The addition of the MDA viewpoint and Token Queue shows how our described Extension Mechanism can support quite powerful extensions, far beyond the visualisations described in this paper.

Acknowledgements

The research was supported by ERDF project 1.1.1.1/18/A/045 at Institute of Mathematics and Computer Science, University of Latvia, and by the H2020 project SELMA (under grant agreement No. 957017).

References

Barzdins, J., Rencis, E., Kozlovics, S. (2008), The Transformation-Driven Architecture. *In: Proceedings of DSM'08 Workshop of OOPSLA 2008*, Nashville, Tennessee, pp. 60–63, University of Alabama at Birmingham.

Barzdins, J., Cerans, K., Grasmanis, M., Kalnins A., Kozlovics, S., Lace, L., Liepins, R., Rencis, E., Sprogis, A., Zarins, A. (2009). Domain Specific Languages for Business Process Management: a Case Study. *Proceedings of 9th OOPSLA Workshop on Domain-Specific Modeling*, pp. 34–40.

Barzdins, P., Celms, E., Barzdins, J., Kalnins, A., Sprogis, A., Grasmanis, M., Rikacovs, S. (2020a). Metamodel specialization based DSL for DL lifecycle data management. *Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings*, Article No.16.

Barzdins, G., Gosko, D., Cerans, K., Barzdins, O. F., Znotins, A., Barzdins, P. F., Gruzitis, N., Grasmanis, M., Barzdins, J., Lavrinovics, I., Mayer, S. K., Students, I., Celms, E., Sprogis, A., Nespore-Berzkalne, G., Paikens, P. (2020b). Pini Language and PiniTree Ontology Editor: Annotation and Verbalisation for Atomised Journalism. *In: ESWC 2020 Satellite Events.* LNCS, Volume 12124, pp. 32-38.

Bisong, E. (2019). Kubeflow and Kubeflow Pipelines. *In: Building Machine Learning and Deep Learning Models on Google Cloud Platform*. Apress, pp. 671-685.

Celms, E., Barzdins, J., Kalnins, A., Sprogis, A., Grasmanis, M., Rikacovs, S., Barzdins, P. (2020a). Towards DSL for DL Lifecycle Data Management. *Communications in Computer and Information Science*, Volume 1243 CCIS, 2020, Pages 205-218, 14th International Baltic Conference on Databases and Information Systems, DB and IS 2020, Tallinn, Estonia, 2020.

Celms, E., Barzdins, J., Kalnins, A., Barzdins, P., Sprogis, A., Grasmanis, M., Rikacovs, S. (2020b). DSL approach to deep Learning lifecycle data management. *Baltic Journal of Modern Computing*, Volume 8, Issue 4, pp. 597-617.

Cook, S., Jones, G., Kent, S., Wills, A. C. (2007): Domain-Specific Development with Visual Studio DSL Tools. Addison-Wesley Professional, 568 p.

Gronback, R. C. (2009). Eclipse Modeling Project: A DomainSpecific Language (DSL) Toolkit. AddisonWesley Professional.

Haifeng, J., Qingquan. S., Xia, H. (2019). Auto-Keras: An Efficient Neural Architecture Search System. *Proceedings of 25th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pp. 1946-1956.

Kalnins, A., Barzdins. J. (2016). Metamodel Specialisation for Graphical Modeling Language Support. *Proceedings of 19th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems*, MODELS 2016, pp. 103-112.

Kalnins, A., Barzdins. J. (2019). Metamodel specialisation for graphical language support. *Software and Systems Modeling Journal*, vol. 18, no. 3, pp. 1699-1735.

Kelly, S., Tolvanen J. P. (2008). Domain-Specific Modeling: Enabling Full Code Generation. John Wiley & Sons.

Kleppe, A. G., Bast, W., Warmer, J. (2007). MDA explained: The model driven architecture: practice and promise. Addison-Wesley.

Miao; H., Li; A., Davis, L. S., Deshpande, A. (2017). Towards Unified Data and Lifecycle Management for Deep Learning. *Proceedings of IEEE 33rd International Conference on Data Engineering (ICDE)*, pp. 571-582, San Diego, USA.

OMG (2014). Object Constraint Language (OCL), version 2.4, OMG document formal/14-02-03, https://www.omg.org/spec/OCL/2.4/PDF.

OMG (2015). Meta Object Facility (MOF) Core Specification, version 2.5, OMG document formal/2015-06-05, https://www.omg.org/spec/MOF/2.5/PDF.

Sprogis, A. (2016). ajoo:Web Based Framework for Domain Specific Modeling Tools. *Frontiers in Artificial Intelligence and Applications*, Volume 291, IOS Press, pp. 115-126, Selected Papers from the Twelfth International Baltic Conference, DB&IS 2016.

Streinberg, D., Budinsky, F., Paternostro, M., Merks, E. (2008). EMF: Eclipse Modeling Framework, 2nd Rev. ed. Addison-Wesley.

Tolvanen, J. P., Pohjonen, R., Kelly, S. (2007). Advanced tooling for domain-specific modeling: MetaEdit+. *Proceedings of 7th OOPSLA Workshop on Domain-Specific Modeling*, Montreal, Canada.

WEB (a). *IBM Rational Software Architect and Modeler*. https://www.ibm.com/docs/en/rational-soft-arch/9.7.0

WEB (b). *Sparx Systems Enterprise Architect*. https://sparxsystems.com/

WEB (c). *MagicDraw*. https://www.3ds.com/products-services/catia/products/no-magic/ magicdraw/

WEB (d). *Modelio*. https://www.modelio.org/

WEB (e). *Eclipse MDT UML2 project*. https://projects.eclipse.org/projects/modeling.mdt.uml2

WEB (f). *Obeo Designer: Domain Specific Modeling for Software Architects*. http://www.obeodesigner.com/

WEB (g). *Flyte - Cloud Native Machine Learning and Data Processing Platform*. https://flyte.org

WEB (h). *Dagster - System for building modern data applications*. https://github.com/dagster-io/dagster

WEB (i). *Metaflow - Framework for real-life data science*. https://metaflow.org

WEB (j). *DVC - Open-source Version Control System for Machine Learning Projects*. https://dvc.org

WEB (k). *Weights&Biases*. https://www.wandb.com

WEB (l). *comet*. https://www.comet.ml

WEB (m). *Amazon SageMaker*. https://aws.amazon.com/sagemaker/

WEB (n). *Microsoft Azure Machine Learning*. https://azure.microsoft.com/en-us/services/machine-learning/

WEB (o). *mlflow - An open source platform for the machine learning lifecycle*. https://mlflow.org

WEB (p). *Neptune*. https://neptune.ai

WEB (q). *ClearML*. https://clear.ml/products/clearml-experiment/

WEB (r). *Aim*. https://aimstack.io

WEB (s). *cnvrg.io CORE*. https://cnvrg.io/platform/core/

WEB (t). *Facebook AI - Reengineering Facebook ai's deep learning platforms for interoperability* (Testuggine, D., Subramaniam, A., Perez, L., Nguyen, B., Liu, S., Ott, M., Adcock, A.). https://ai.facebook.com/blog/reengineering-facebook-ais-deep-learning-platforms-for-interoperability

WEB (u). *Graphical Modeling Framework (GMF) Tooling*. https://eclipse.org/gmf-tooling/

Received October 25, 2021, revised February 6, 2022, accepted February 21, 2022