

DSL Approach to Deep Learning Lifecycle Data Management

Edgars CELMS¹, Janis BARZDINS¹, Audris KALNINS¹,
Paulis BARZDINS¹, Arturs SPROGIS¹, Mikus GRASMANIS²,
Sergejs RIKACOVŠ²

¹Institute of Mathematics and Computer Science, University of Latvia, Raiņa bulvaris 29, Riga,
LV-1459, Latvia

²Innovation Labs LETA, Latvia, Riga, Marijas iela 2, Riga, LV 1050, Latvia

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

Abstract. A new approach to Deep Learning (DL) lifecycle data management tool support is presented: a very simple DL lifecycle data management tool, which however is usable in practice (it will be called Core tool) and a very advanced extension mechanism for this Core tool which in fact converts the Core tool into a DSL tool building framework for DL lifecycle data management tasks. The extension mechanism is based on the metamodel specialisation approach to Domain Specific Language (DSL) modelling tools introduced by the authors. 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 the domain define a Specialised Metamodel (SMM). The paper concludes with a detailed description of future research directions, concerned with defining a more general UMM and its usage.

Keywords: DSL, DL, metamodel specialisation, DL lifecycle data management

1. Introduction

This paper is an extended version of the Baltic DB&IS 2020 paper (Celms et al., 2020). As in (Celms et al., 2020) we start with a description of the problem to be studied.

The Deep Learning process is long and tedious. It has many parameters, data, code versions, and more to keep track of during the many runs of training and model development. All this emphasizes the need for effective training lifecycle data management. Currently there already exist several systems for supporting this process. We will explore them in more detail in the following section; here we only note that for different Deep Learning (DL) tasks support requirements may vary greatly. Due to this aspect the existing DL lifecycle data management systems cover either a very small part of these requirements or become excessively complicated in order to cover as large as possible part of the requirements.

We borrow the conclusion arrived at in the system modelling area in similar circumstances – try a Domain Specific Language (DSL) approach to DL lifecycle data management.

The goal of this paper is to deeper investigate the DSL approach in this area and offer a possible solution for the DL lifecycle data management problem. The solution consists of two parts:

1. A very simple DL lifecycle data management tool, which however is usable in practice (in what follows it will be called Core tool);
2. Advanced extension mechanism for this Core tool which in fact converts the Core tool into a DSL tool building framework for DL lifecycle data management tasks.

When combined these two parts deliver the ability to create custom tools for specific DL tasks, without unnecessary bloat but with all the specific functionalities the user deems needed.

2. Related work

Machine learning has a complex lifecycle consisting of many phases and steps, from the preparation of the training and test data; via model development and training; to testing and, finally, deployment. Although the Machine Learning libraries themselves (like Scikit-Learn, Tensorflow, PyTorch, etc.) have already matured in the last few years, the pipeline around them and the corresponding tool landscape is still in development and is very fragmented. Each of the tools covers one or several aspects of model development, and for a complete pipeline you may need to select a whole rainbow of tools to then be combined. Current state of the art tools are many and varied, each with its own focus.

There are tools in the landscape which orchestrate ML workflows in the cloud (Bisong, 2019; WEB, a; WEB, b). Other tools like (WEB, c) extend the workflow orchestration with data warehouse integration, state transfer and meta-training. However, they do not provide any means for monitoring the training and propose to use Jupyter notebooks for this purpose.

Some tools like DVC (WEB, d) focus on version and dependency management for the data artefacts and models, but they do not track the experiments themselves.

One of the most important phases in ML is model development and training. Some tools (e.g. AutoKeras (Haifeng et al., 2019)) try to automate this phase providing means for automatic exploration of hyperparameter space and selection of the best parameter values for the given problem and training data as well as with selecting the best architecture for the model, making the ML available for non-ML experts. In some cases, these tools abstract the developer from directly running the ML experiments.

However, these techniques are not mature enough yet, and, in many cases, developers still have to run the machine learning experiments manually. This involves iterative running of experiments in search for the best model architecture and for the best hyper-parameters. Natural requirements here are the ability to *track, compare and reproduce* the experiments. Good tool support is essential for development productivity. To this goal these tools (WEB, e; WEB, f; WEB, g; WEB, h; WEB, i; WEB, j; WEB, k) provide a set of library functions for augmenting the training program with a tracking functionality. A natural counterpart then is a tool for visualisation and result comparison.

Some of the tools concentrate on the tracking functionality, and their support for visualization of ML experiments is limited (WEB, e; WEB, f), or they provide a command line interface for experiment comparison (WEB, l). Other tools (WEB, k) rely on complementary tools (e.g. (WEB, m)) for experiment visualization.

However, most tools have been built with support for visualization and comparison of experiment metrics, usually as a dashboard containing tables and sometimes charts.

There are cloud-based solutions (WEB, g; WEB, h), closest to ours being Weights&Biases (WEB, g), both commercially closed projects. They are built to support visualisation and comparison of experiment metrics, as a dashboard containing tables and charts. Unfortunately, users are limited to the visualisations provided, and though there are many, if something is missing the tool cannot be extended. You also can't add semantics to role names of logged content, meaning unless the tool has built in support for it, you can't have data depictions be determined by the type of data you're logging. What you end up with is a tool that tries to be all encompassing, but for most real tasks ends up having tons of unused functionality while not meeting all of your needs.

Open source tools like (WEB, i; WEB, j) can be extended, but such an extension, besides direct development of the required software, requires deep understanding of the existing software and, especially, the data structures used in it. What's missing is an easily usable extension possibility, for adapting to a specific deep learning task domain. This situation becomes very similar to the one in the system modelling area. There the Universal Modelling Language (UML) was developed and naturally this language was very complicated. Typically for any real tasks a very small subset of these many possibilities was applicable, while at the same time something was missing for tasks in the given domain.

Besides that, in real application domains there is a desire to obtain software implementation from the chosen language automatically. As a result, the idea of Domain Specific Languages and tools developed rapidly. Required was not a single universal tool, but a DSL tool building framework. By using such a platform an expert of the given problem domain could relatively easily build the required tool themselves.

A question then arises: *could a Domain Specific language approach not be applied to the deep learning lifecycle data management area as well?*

In other words, no complicated universal tool, instead a DSL tool building framework, where the Deep learning domain expert themselves can build a tool for their specific needs.

3. DL lifecycle data management framework: general structure

The general structure and use of our framework is split amongst three “machines”, though they need not be separate physically (see Fig. 1):

- The Data Server – which hosts the tool and contains the database and all the files;
- The Workstation – a machine that wants to use our tool to track experiments run on it;
- Any Machine via web access – the way to access the platform to view the tracked information.

The framework itself consists of two components:

1. Core platform, which includes:
 - a. Core tool which works on the Data Server;
 - b. Core library which is placed on the Workstation.
2. Core extension mechanism which ensures the creation of the DSL tool for the given DL lifecycle data management scenario.

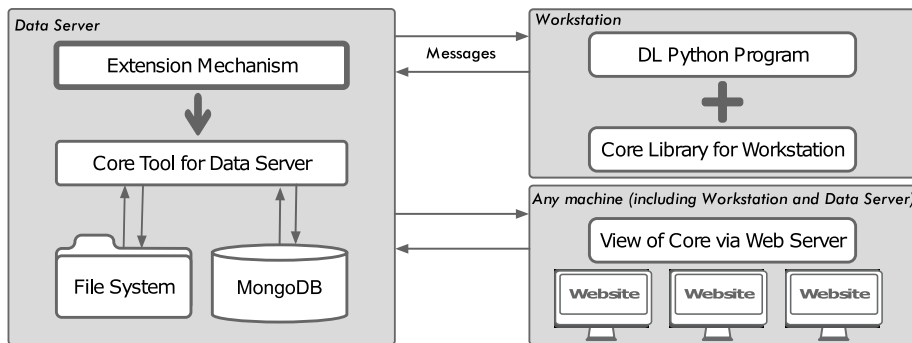


Fig. 1. General structure of LDM framework.

The Core tool is a simple but usable DL lifecycle data management tool. Website access lets you log in and then create and view projects. For each project you can upload training and test data, view your runs of the training program, under each seeing your logged hyper-parameters, gold and guessed captions, or whatever other string value you might want to log. You can also access uploaded images and graphs, checkpoints and code versions for recoverable progress. All the logs and files come from code augmented with the Core library tracking commands. All this is just the minimalistic base version.

The Core library is installed on the workstation. It consists of five possible messages that can be sent to the Data Server:

1. **Login** – to send your username and password so the Core tool knows to give access to your projects;
2. **StartRun** – to note that a new run has started, for the Core tool to know how to split and organise the logs and files sent;
3. **Log** – for sending String messages to the Core tool, which are then processed by the tool in accordance to the co-sent roleName;
4. **UploadFile** – for sending files to the Core tool, these can be anything ranging from images through graphs to whole code files, later accessible through the web access to the tool;
5. **FinishRun** – to tell the Core tool that a run has finished, useful for seeing that a run hasn't failed midway, gives ability to note how long the run took.

The Core Platform extension mechanism is the main contribution. It takes the basic Core tool, and allows the user to extend it into a specialised and as excessive as need be

tool, that perfectly fits the specific DL scenario. It works through revealing some of the inner structure at a consumable level, so the user doesn't have to familiarise with the whole code – like an API. Then the user can program any functionality they wish, while having access to all the structured data sent from the Workstation and saved on the Data Server.

4. End user view

When using the web access, you are met with the page seen below (Fig. 2). First you are required to log in using your username and password, then you gain access to all of your projects as well as the ability to create new ones. Under each project you can see some information on it, upload or download training and test data, as well as see all the runs.

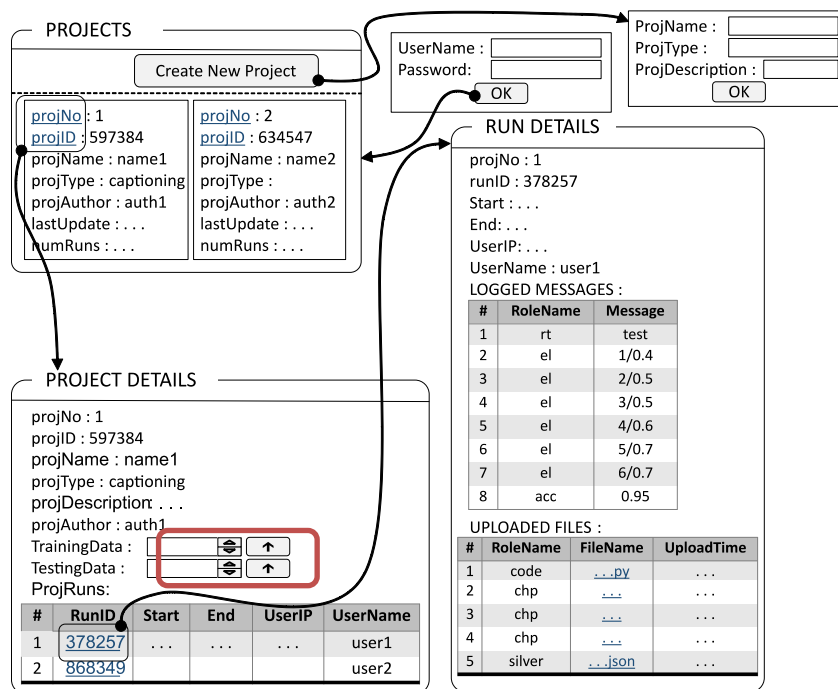


Fig. 2. Web view of Core tool

For the training and test data, when first adding them they have to be uploaded as zip files (using the upload button provided, circled red). These zip files will then be stored in the repository shown later in the paper (Fig. 8) in the folder TrainTestData. The unzipped folders are to be stored in the same repository folder and will have the same name as the corresponding zip file. In the case where the training and / or test data are already stored in the folder (i.e. they were manually placed in the repository folder before) the upload button should not be used, instead in the corresponding training and / or test data field the name of the folder should be entered.

The training and test data will still likely need to be stored locally as well for the DL expert running the experiments on the Workstation, but having them stored on the Data

Server allows for easy migration between Workstations or for other teammates to download the latest code, datasets, and checkpoints and be able to run and view the latest version of your project.

Moving on to the runs, under the project view you can see all of them listed with some basic information on them (runID, start and end time, IP and name of user that initiated it). Each of the runs can be clicked to display the run details. These extend the basic information with logged messages and uploaded files from the run. These are textual logs and files of any type uploaded from the Workstation during the run by augmenting the code with provided library functions. The available library functions are:

```
login(user_id, psw): a trial to authorize 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 by means of these functions the Workstation sends the following messages (shown in Fig. 3) to the Data Server.

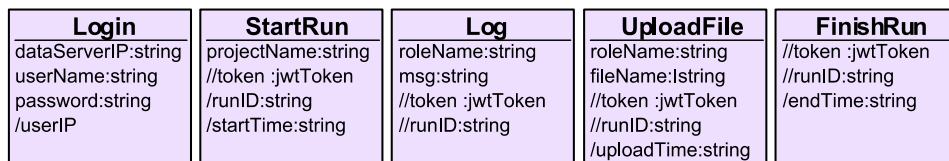


Fig. 3. Class diagram of messages.

“/” before the attribute name means that the value of this attribute is inserted by the Core library. “//” before the attribute name means that the value of this attribute is not sent from the Workstation but generated by the Core platform upon message reception.

The library is very user friendly as *login* and *startRun* both need to be called only once at the very start of the program, similarly *finishRun* is only called once at the very end. The rest of the time the user only has to use *log* and *uploadFile* – one for sending String (textual) messages to the Data Server, the other for sending files.

The first of the library functions that the user has to call is *login*. As parameters it is given the user ID and password. Same as how it is on the web access, you first have to log in before being given access to edit the projects.

Then comes the *startRun* function. As parameter it is given the name of the project that you are currently running. This also notes down the time that is later displayed in the run info, as well as used for the calculation of total run time.

Now follows the main body of the user’s code, within which only the *log* and *uploadFile* functions are used. The *log* function can be used quickly and constantly for saving small values that might be useful to remember. These can range from more abstract run wide logs like “run type” for noting whether this is a training or testing run,

to very narrow and detailed ones like seed used for selecting 10 random images from the testing set to be displayed. The only thing the user has to remember to do is add a suitable *roleName* for the logged value. This can be anything user chosen, but it has to stay consistent if later extension functionalities are to work, so if there were to be multiple DL experts working with the code, they would have to agree amongst themselves to be consistent with role names. The *uploadFile* function is very similar, a slight difference user experience wise is that a file does have to be created for whatever you might want to upload (experience wise different to *log* where any variable value can be cast to String and sent with ease). Although often some files will be created anyway in the user's workflow already (the current code version, checkpoint files used for saving progress, etc.). However, to use the tool's full functionalities now and later, the user might want to create and upload some more abstract files, that might only become useful once the extension mechanism is used. Examples might be uploading images or graphs that the user might usually view only in their Jupyter notebook. Uploading them means that their teammates or end users have easy access to these as well. More abstract examples might be uploading files containing many abstract values from the run; this might be done instead of logging them all to allow a later extension to display them in a user-friendly way, instead of simply being listed.

At the very end the user is only left to call *finishRun*, that doesn't take any parameters and simply signals to the Data Server that the run has ended; the end time is now noted and can be displayed (as well as total run time calculated). There can be cases where the program fails mid-run, resulting in a scenario where the finish run is never sent (unless the user has appropriate error catching in place). In these cases, the run simply displays no end time, but the run can be viewed all the same.

In the conclusion of this section we note that the information already presented is more or less adequate for a simple Workstation user to start using the Core platform seriously: to extend their DL program in Python with Core library function calls for sending data to Data Server as well as for downloading files from Data Server to their Workstation, e.g. Code or Checkpoint from an earlier Run. Such file download is planned to be done via the web page (see Fig. 2) using the traditional web browser facilities – open the web page on their Workstation and download the reachable from this page files via the Save As option.

Let us remind you that the time-consuming DL process occurs exclusively on the Workstation and the Data Server is only used for relatively infrequent storing of training results.

5. Inner structure

We start here with the presentation of the Logical Metamodel for this platform in the form of a class diagram (see Fig. 4). We will call this model the Core Logical MM. In this MM and others following the slash symbol before the attribute name marks that those values are set by the Core platform itself. The values of other attributes come from the messages sent by the Workstation (or are directly entered via the relevant web page, e.g. project name for a new project).

Now about the semantics of this MM. Class and attribute names already explain the semantics to a great degree. We only note that instances of the *Run* class (together with classes *LoggedMessage* and *UploadedFile*) mean the information sent to the Data server from Workstation during the training run.

We assume that the neural network training process consists of many runs. In addition, the neural net and code itself may be modified between consecutive runs.

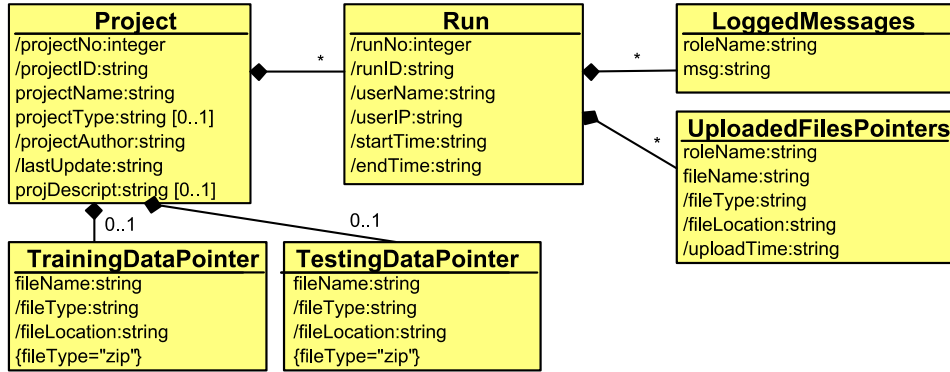


Fig. 4. The Core Logical MM.

To clarify this situation Fig. 5 presents an instance of this MM which corresponds to two training runs.

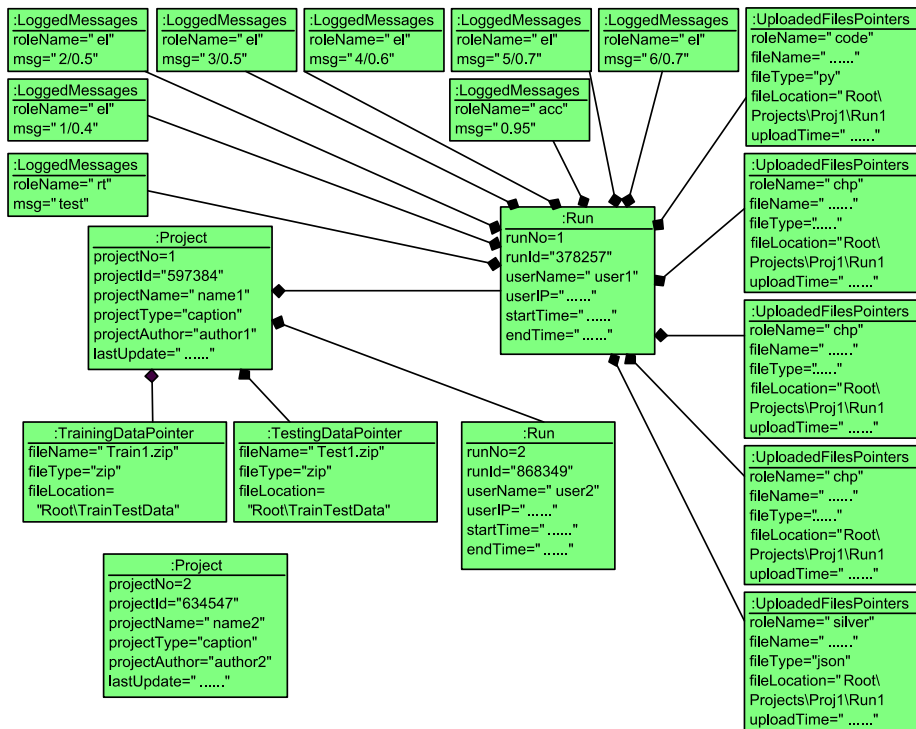


Fig. 5. Instance of Core Logical MM representing two runs.

Fig. 6 shows the same instance in the form of web pages that should now seem familiar, which should be built by our offered Core platform.

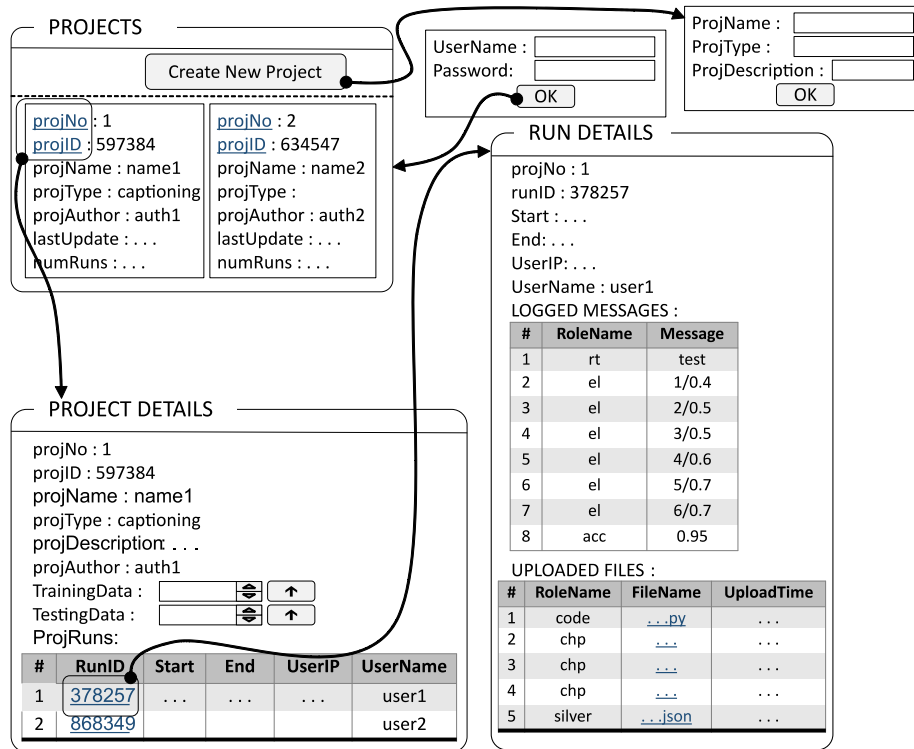


Fig. 6. Instance shown as web pages.

6. Core platform extension mechanism: basic ideas

The biggest contribution of this project is the extension mechanism. There are two ways to allow extension:

1. Reveal the Core tool code – this is complicated for the user as they have to familiarise with too much information;
2. Reveal only some necessary parts that are easily understandable, while still giving enough knowledge of the tool internals for extension.

The first thing we reveal is the Core Logical MM (Fig. 4.). It shows the structure of data within the tool. Physical representation of data corresponding to the Core Logical MM is done via two complementary facilities, the structures of which are the other two parts we are revealing:

- a. The Logical MM and its instances are stored in Mongo DB according to the structure shown in Fig. 7 (ProjectID and RunID are generated automatically);
- b. The files themselves (including the Core platform software) are stored in the Data server file system according to the repository shown in Fig. 8.

```

db.createCollection("project")
db.project.insertOne({projectId:597384, projectNo:1,
  projectName:"name1", ... })
db.project.insertOne({projectId:634547, projectNo:2,
  projectName:"name2", ... })
db.createCollection("run")
db.run.insertOne({runId:378257, projectId:597384,
  runNo:1, ... })
db.run.insertOne({runId:868349, projectId:597384,
  runNo:2, ... })
db.createCollection("loggedMessages")
db.loggedMessages.insertOne({runId:378257,
  roleName:"rt", msg:"test"})
db.loggedMessages.insertOne({runId: 378257, roleName:
  "e1", msg:"1/0.4"})
db.createCollection("uploadedFilesPoint")
db.uploadedFilesPoint.insertOne({runId: 378257,
  roleName:"code", fileName:"..." ...}) ... ..
db.createCollection("trainingDataPoint")
db.trainingDataPoint.insertOne({projectId:597384,
  filename:"Train1.zip", fileType:"zip", fileLocation:
  "ROOT\TrainTestData"}) ... ..

```

Fig. 7. Mongo DB presentation of the given instance.

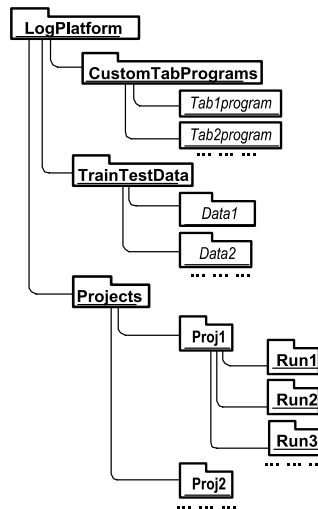


Fig. 8. Repository structure

Now some comments on this structure. LogPlatform is the root folder of this structure. In this folder the Core platform software is stored. For instance, the full address of this folder could be C:\Programs\LogPlatform. This address will be denoted by ROOT.As a result the full name of folder Proj1 would be ROOT\Projects\Proj1.

The opening of Core platform internals mentioned above is sufficient for our offered extension mechanism.

7. Extension mechanism: our specialisation approach

The proposed extension mechanism will, to a great degree, be based on the metamodel specialisation approach to DSL modeling tools introduced by authors (Barzdins et al., 2009; Sprogis and Barzdins, 2012; Kalnins and Barzdins, 2016; Kalnins and Barzdins, 2019). The main idea of metamodel specialisation is that we at 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's classes, as many as we need. The subclasses are defined according to UML rules, but with some restrictions. Class attributes may be assigned with new fixed values, but new attributes may not be added. Similarly, for associations the role names may be redefined (subset) and multiplicities may be changed (shrunk). In our new domain we also allow attribute names and types to be redefined. We illustrate all this on a very simple example from a workflow domain. This example is taken from our paper (Kalnins and Barzdins, 2019), but slightly extended to also show our new specialisation features. Fig. 9 shows the UMM for this workflow.

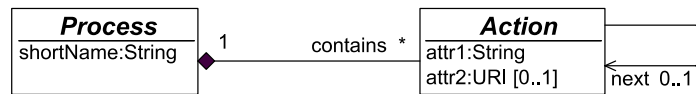


Fig. 9. UMM for simple workflow.

Fig. 10 shows a specialisation of this UMM for business trip workflow in an enterprise in a standard UML notation. Similar to (Kalnins and Barzdins, 2019), in order to make diagrams more compact and readable, we use a custom notation for specialisation (only slightly extended with respect to (Kalnins and Barzdins, 2019)).

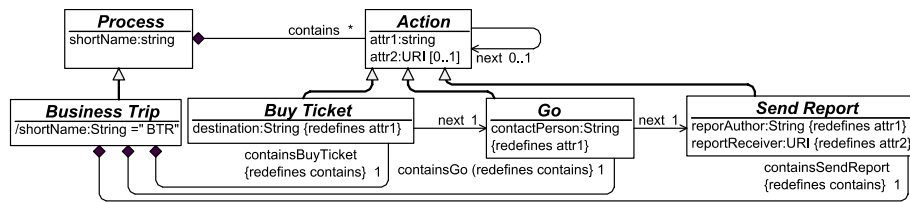


Fig. 10. Workflow specialisation in standard UML notation.

This notation uses the super-object (class, attribute or association) name in braces to reference it in the sub-object. See this notation for business trip in Fig. 11. In the general case not all UMM classes may be specialised. In order to define which UMM classes

may be specialised we show their class names in *italic* (in our example these classes are *Process* and *Action*).

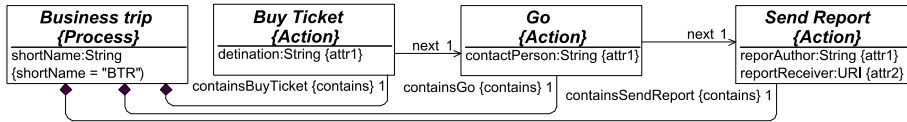


Fig. 11. Custom notation for Business Trip specialisation.

Now the main new idea in this paper. In comparison with the paper (Kalnins and Barzdins, 2019) the specialisation concept itself will be extended. In (Kalnins and Barzdins, 2019) it was assumed that the semantics of specialised classes are directly determined by their attribute values or a default attribute values set. We will call such specialisation a simple specialisation. But in 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 our domain we will use true custom extension of the Core tool functionality specially adapted for the given DL use case by means of invoking an additional custom program at appropriate points of the Core tool functioning. The functional specialisation defines how such custom programs can be found. Both simple and functional specialisation will be used in this paper.

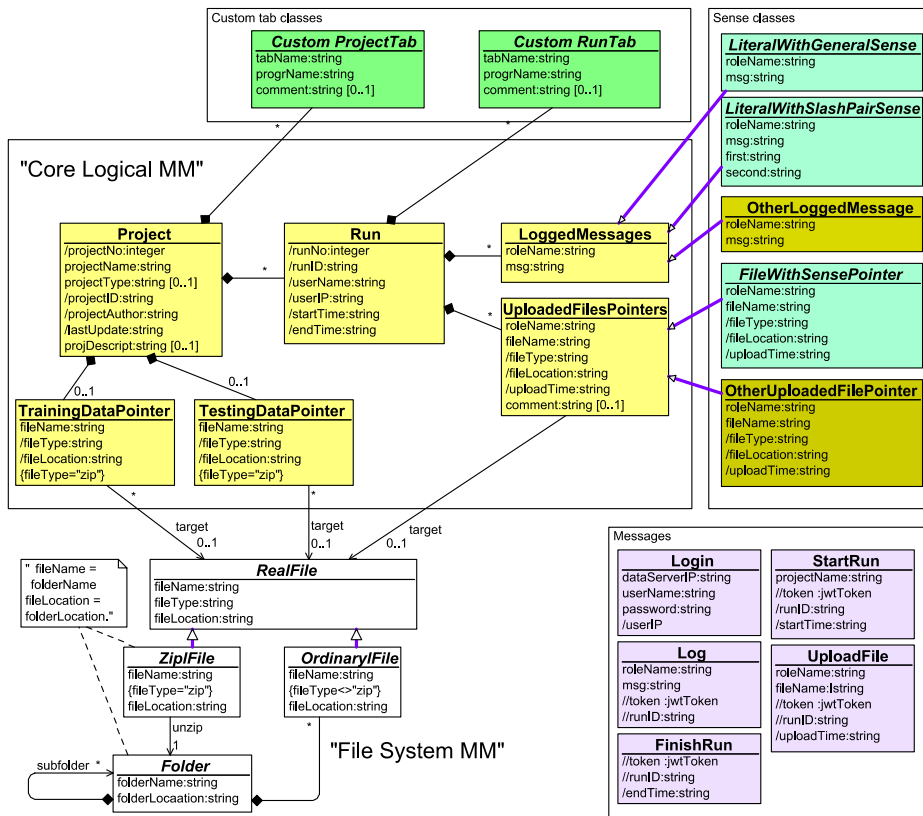


Fig. 12. Universal metamodel (extended Core Logical MM).

We start the definition of our extension mechanism by extending the Core Logical MM with new classes called extension classes (their names are shown in italic font). The extended MM is shown in Fig. 12, further on we will call it the Universal Metamodel (UMM). As Fig. 12 shows, there are two kinds of extension classes corresponding to two kinds of extensions supported by our LDM framework:

- Sense classes: *LiteralWithGeneralSense*, *LiteralWithSlashPairSense*, *FileWithSensePointer*
- Custom tab classes: *CustomProjectTab* and *CustomRunTab*

The LDM framework we offer is based on the idea that by means of specialisation of the mentioned extension classes we will define concrete DL lifecycle data management tools (called DL DSL tools) which provide new additional possibilities in comparison with the Core platform.

Now let us explain this idea in a greater detail.

For sense classes we will apply simple specialisation – see classes *RunType* and others coloured light blue in Fig. 13.

However, the main heavyweight DL DSL tool building facility is the specialization of custom tab classes. For these classes we will apply the functional specialisation. These classes have an attribute *progrName*, which after specialisation must point to an independent executable program, also called an Extension program. This program should be inserted in the repository, shown in Fig. 8. This means that simultaneously with defining a specialised tab class we have to build the corresponding extension program. Namely due to this reason we need the above described opening of the internals of the Core platform. We note that our UMM also includes the File system MM. Its specialisation will be used to describe more precisely the source data for extension program building. We also have to add two more details on extension programs:

- Any extension program has just one parameter. Its value in the corresponding extension call will be the corresponding *projectID* value (in the case of *Custom Project Tab*) or the corresponding *runID* value (in the case of *Custom Run Tab*).
- Any extension program call will open a new web page presenting the result of this program execution (e.g. Loss graph).

8. Extension example explanation

Now let us go to the explanation of Fig. 13 presenting one concrete specialisation of the UMM. This specialisation defines a concrete DL DSL tool.

First, let us stress that our extension mechanism (via the extension class specialisation) refers only to additional features of Data Server (DS) which permit to view in a more understandable way the information sent by Workstation to Data Server (in comparison with the bare Core platform). The custom tabs permit to view this information in a completely new format, e.g. in the form of various graphs. For this explanation we assume that from a Workstation to Data Server the same messages have been sent as in the example mentioned in Section 4 (see the Fig. 5). According to the Fig. 5 the following message roles are used: *acc*, *rt*, *lp* for ordinary messages; and *code*,

chp, *silver* for files. The semantics of these roles are, to a great degree, explained by sufficiently expressive names of specialised classes used in Fig. 13: *Accuracy*, *RunType*, *LossPairs*, *Code*, *Checkpoint* and *SilverCaptions*. Further explanation is needed for the class *LossPairs* and its parent class *LiteralWithSlashPairsSense*: the attribute *msg* of this class is used to code message pairs (first, second) separated by the slash character. For example, if *msg*="25/0.95", then first =25 and second =0.95. At defining the class *LossPairs*, the first is renamed to *epochNo* and second to *loss* (such a redefinition is permitted at class specialisation).

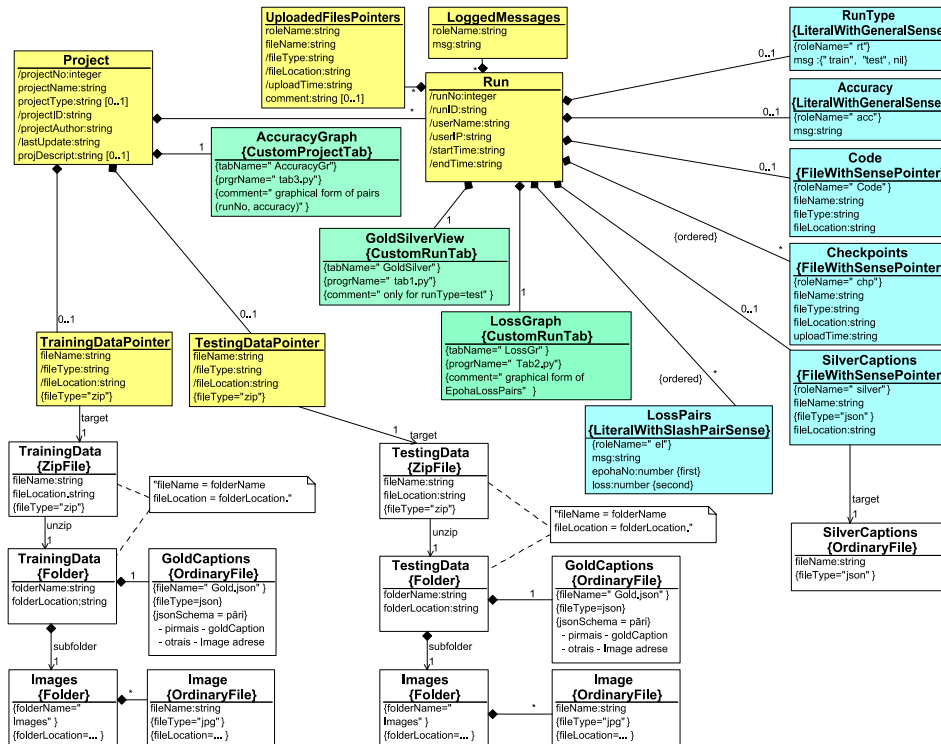


Fig. 13. One concrete specialisation of the UMM.

The results of this specialisation are illustrated in Fig. 14 where the Run Details web page is presented. The first difference from Fig. 8 where the corresponding web page is also visible is such that the messages sent to DS are grouped according to the specialised classes and thus have become more understandable (in addition also the possibilities related to *LiteralWithSlashPairSense*).

Now let us discuss the Custom tabs. Fig. 13 shows three Custom tabs: *AccuracyGr*, *LossGr* and *GoldSilver*, the first is a specialisation of *CustomProjectTab*, the remaining two are specialisations of *CustomRunTab*. As already mentioned, the feature there is the extension program: in the case of *AccuracyGr* it is a program with the name *tab3.py*, in the case of *LossGr* it is a program with the name *tab2.py*, but in the case of *GoldSilver* it is a program with name *tab1.py*.

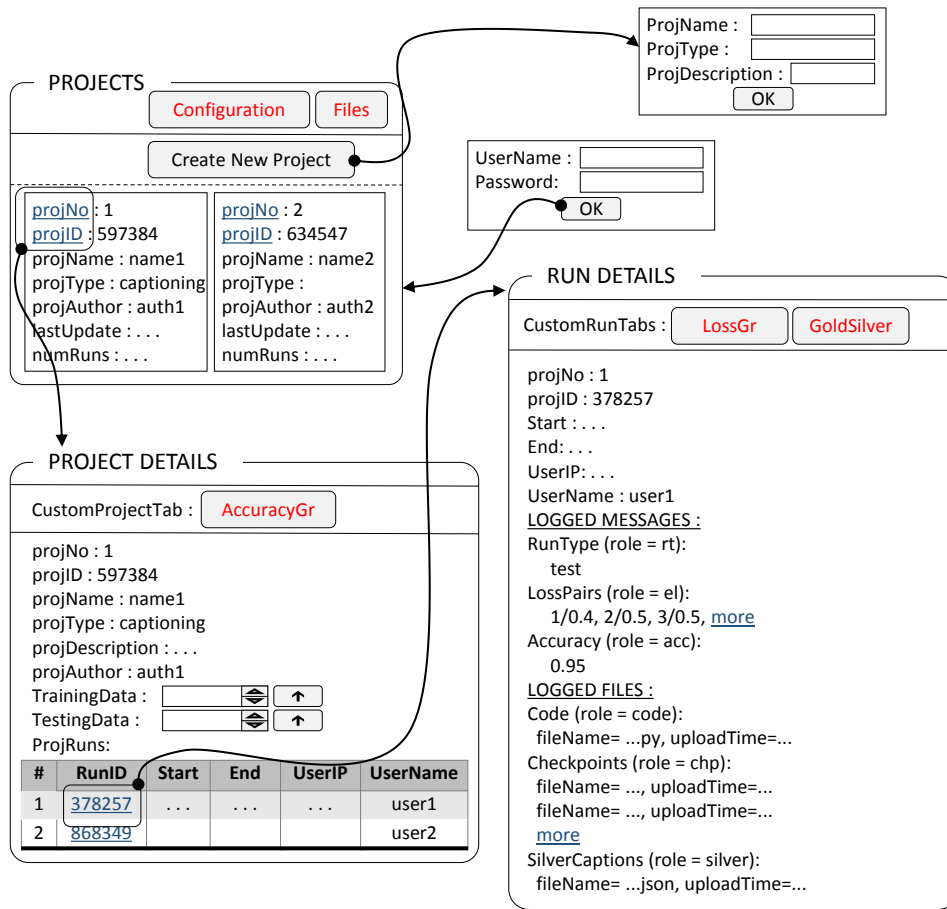


Fig. 14. The result of specialisation as a web page.

Now let us explain what these programs are doing. The program `tab3.py` when invoked with a parameter value equal to the `projID` generates a web page for the given *Project* instance containing an Accuracy graph from the *AccuracyPairs* (*runNo*, *accuracy*). In this graph *runNo* is mapped on the x-axis, *accuracyValue* on the y-axis. The program `tab2.py` for the given *Run* instance (when invoked with the parameter value equal to the `runID` of this instance) generates a web page with a Loss graph from the *LossPairs* instances related to the given *Run* instance. In this graph *epochNo* is mapped on the x-axis, but *lossValue* on the y-axis. The program `tab1.py` performs a slightly more complicated action. This program (invoked with `runID` value of a *Run* instance) finds the corresponding *RunType* instance and if its `msg` attribute value is equal to "test", generates a web page which for the corresponding *TestingData* contains the table visualized in Fig. 15. For this purpose this program first uses the folder *TestingData* visible in Fig. 13. This folder in turn contains the *Images* folder and the *GoldCaptions* file. Secondly, this program finds the *SilverCaptions* file corresponding to this *Run* instance.



IMAGE	GOLD	SILVER
	three men fishing	a man is riding a boat
	Riga city skyline	a city filled with lots of buildings
	cars standing in traffic	a car is parked on a street
.....

Fig. 15. Results of the GoldSilver extension program execution.

Had our example more Custom tabs, they should be explained as well and our memorandum would be longer. Now on the topic where these Custom tabs are visible in the Data Server website and how to invoke them. Let us look at Fig. 14. There we see that Custom Run tabs are visible in the Run Details web page, but Custom Project tabs in the Project Details web page. By clicking on these tabs, the following Core platform action occurs: the corresponding ID value is found (in the case of *Run* it is runID, in the case of *Project* it is projID). Then the extension program corresponding to this tab is invoked with the selected ID value as the input parameter. As it was mentioned above, program execution result is to show the corresponding web page.

9. DSL memorandum and configuration

Before proceeding we want to stress the following aspect: our DL DSL framework is related to the following 4 human actors:

1. Framework developer
2. DSL tool configurator/developer
3. DSL tool user, i.e. DL programmer at a Workstation
4. End user, e.g. LETA data expert

The DSL tool developer is the actor who performs the corresponding UMM class specialisation and, if required, develops the corresponding extension programs and places these programs in executable form on Data Server in the repository shown in Fig. 8. Beforehand however the tool developer has to perform one more job – together with the DSL tool users fix the list of permitted role names and the semantics of these roles for messages to be sent from a Workstation to the Data server. Together with the role list it is also necessary to agree on the structure of messages corresponding to these roles, including the files to be uploaded. The agreement on the file structure should be sufficient for developing the corresponding extension programs. Such an agreement for a DSL will be called a DSL memorandum. Let us stress once more that such a memorandum stands outside of the formalism used by us (the above-mentioned file structure MM only helps in writing such a memorandum). In fact, there is a need for

such a memorandum for the development of any DSL tool, e.g. to explain in natural language the semantics of used DSL symbols (but it is not referred to so formally). In our case the agreement covers a wider area and therefore we use such a term.

Finally, how the Core platform knows which tabs must be shown in the given web page and what values must be passed as parameters. For this purpose a special configuration tab named Configuration is placed in the PROJECTS page. By clicking on the Configuration tab, the page visible in Fig. 16 is opened.

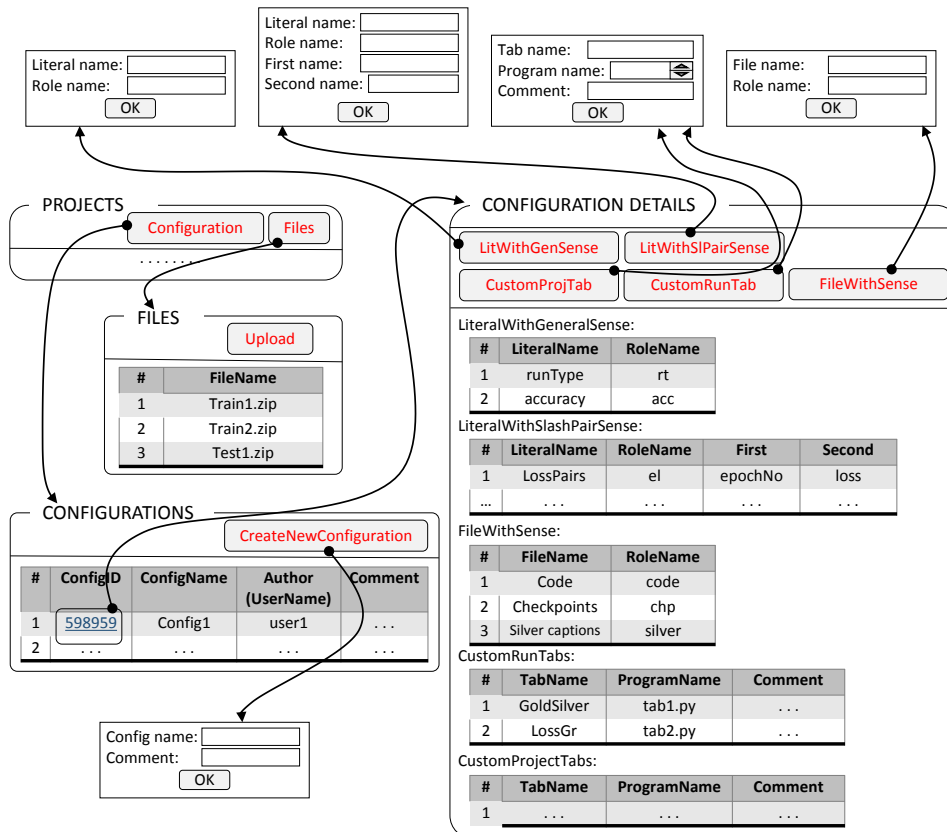


Fig. 16. Tables in the web page opened via Configuration tab.

By executing the actions visible in this page (i.e. by filling in the corresponding tables) we pass to the Core platform all the required information – both for sense class specialisation and custom tab definition. In other words, we pass to the Core platform the information that the UMM specialisation of your DL DSL tool would contain, for example the information contained in Fig. 13.

Now briefly about the Core platform implementation. The implementation of the “bare” Core platform is quite straightforward: the Core Logical MM is stored in the Mongo DB (see Fig. 7), but the corresponding files are stored in the file system according to Fig. 8. Then follows a resource demanding, but quite simple from the logical point of view programming of the used web pages. Certain comments are required on the Extended Core platform implementation, which ensures the described Extension mechanism. In order to make the use of the Extension mechanism simpler, we

will use not the code generation method (as it is typically done for building traditional DSL tools), but an interpretation method. Namely the information entered via the Configuration tool (the tables visible in Fig. 16) is directly stored in the Mongo DB and further, when a request for a web view appears, this view is generated from the “bare” Core information and modified in the interpretation mode according to the entered configuration information.

10. Future research

As was mentioned before, our approach to DL DSL is based on metamodel specialisation. In this paper two kinds of specialisation are used – simple and functional. Functional specialisation provides the possibility to define a very broad spectrum of DL DSL extensions. However, functional specialisation requires non-trivial programming from the specialisation developer. A natural question appears – whether the UMM could be extended in a way that for graphical visualisations of training progress no additional programming would be required and simple specialisation would be sufficient. Our future research direction will be based on searching for such kind of UMMs.

In the following parts of this section we will sketch out an idea for implementing this goal. It will be for defining a Run tab representing simple graphs, without any additional programming. Such graphs are important for graphically representing the increase in training result quality. In previous sections there were two such tabs, showing Loss and Accuracy graphs. We will use the concept line graph – it will be a graphical representation of two-column tables. An example of such a table is shown in Fig. 17.

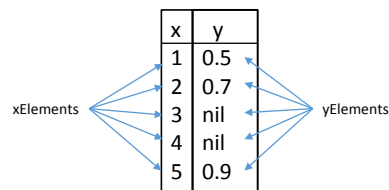


Fig. 17. Two-column representation of data.

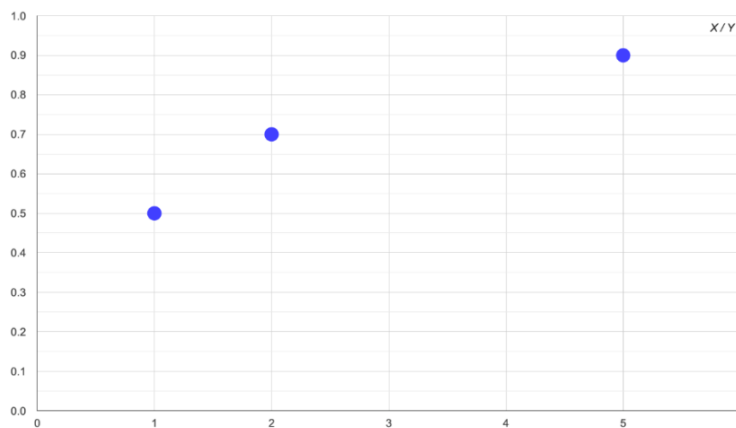


Fig. 18. A line graph of Fig. 17. data.

The graph representing this table is shown in Fig. 18. Some y-column values may be missing – they have a value of nil.

From the viewpoint of UMM+ (our take on an extended UMM), we will identify such graphs as two-column tables. Such a two-column table can be understood as two lists – one for the x-column and one for the y-column; see this in Fig. 19.

For the specialisation of UMM+ we will also use OCL navigation expressions (OMG, 2014). Now in Fig. 20 we will show an example of UMM+ specialisation where Project tab "Accuracy" is defined completely without any use of custom programs. In Fig. 19 an extension of UMM+ is visible where the previously used Custom Project Tab is split into two subclasses – Custom Functional Project Tab and Custom Graph Project Tab, where the second one is a simple specialisation. The Graph Tab has associations to two classes – *xElement* and *yElement*. Both associations are ordered at the target end. The extended UMM+ contains simple OCL expressions.

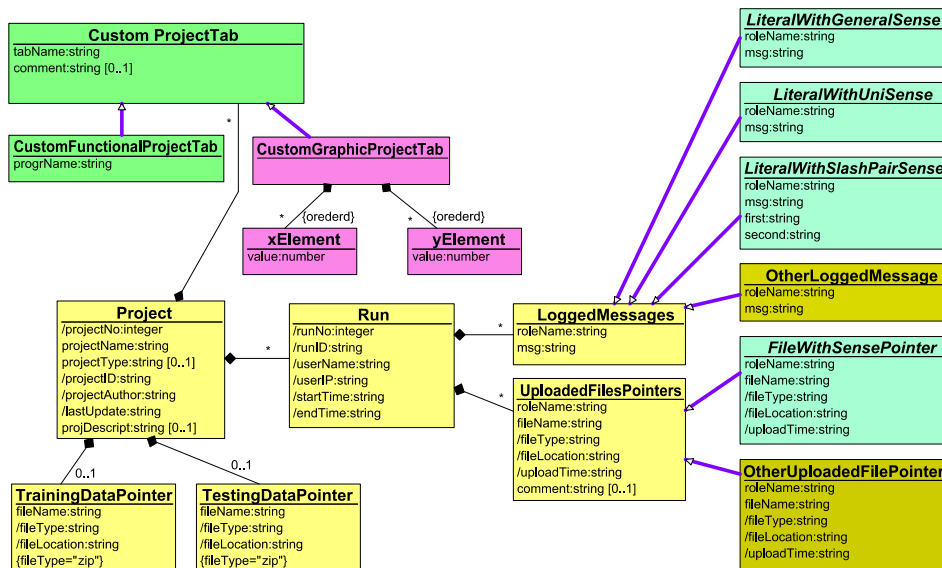


Fig. 19. An extension of the previous UMM.

Accuracy Graph is a subclass of Graph Project Tab. *xElement* is specialised to *RunNo* (Run Number) and *yElement* to *Accur* (Accuracy at the end of this run). Each of these classes has one attribute – the corresponding value. OCL expressions are used in the superclass to specify these attribute values – $\{self.RunNo.value = self.Project.RunNo\}$ and $\{self.Accur.value = self.Project.Run.Accuracy.msg\}$. We assume that UE (Universal Engine which interprets the SMM, see (Kalnins and Barzdins, 2019)) understands such OCL expressions, so nothing has to be specially programmed.

Thus, we have sketched one idea for future research. This idea can certainly be extended in various ways. One is the most natural – to support various more advanced graphical representation styles. First, we may want to define the colour of the graph in the specialisation – red, blue etc. Further, we could have several graphs for representing advances in the training process – like we have the Accuracy and Loss graph. In fact,

there are two accuracies showing training advances – training Accuracy (discussed here) and validation or testing result Accuracy.

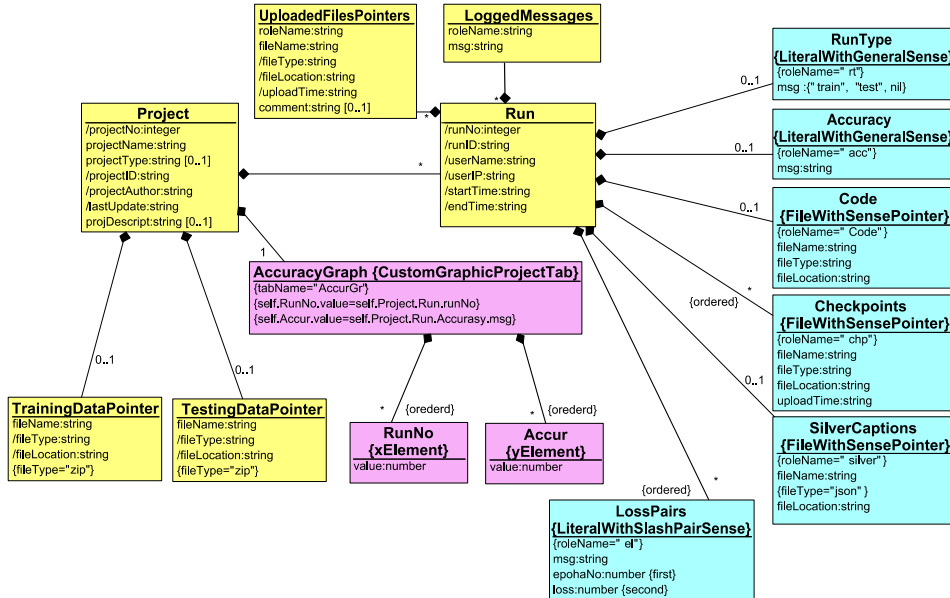


Fig. 20. An example specialisation of the UMM+.

We may want to show several graphs in the same visual presentation, differentiated by graph colour. All these options seem realistic to define via UMM specialisation without any programming, so there is a lot to do.

11. Conclusions

The paper provides both new theoretical and practical results for the DL lifecycle data management area. The main theoretical result is a significant extension of the metamodel specialisation approach for DSL development. But the practical aspects are related to the ERDF project 1.1.1.1/18/A/045, where an easily usable system for DL lifecycle data management framework covering all phases of DL must be developed.

A special orientation there is towards tasks for news agencies, including the Latvian News Agency LETA (it is a partner in the project). That is namely why the task of image captioning is chosen as a specialisation example in the paper.

This paper, as mentioned in the Introduction, is an extended version of the Baltic DB&IS 2020 paper (Celms et al., 2020). The main additions, besides some local extensions, expand on: related works, especially Weights&Biases (WEB, g) which is the closest project to our approach; the extension mechanism, as well as introducing a Project tab; the DSL tool user point of view, description of the use of the library within your Python code. Towards the end of the paper there is also a novel section "Future Research", talking about some of the ideas for where this project could go in the future.

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.

References

- 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.
- 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. (2020). 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.
- 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.
- OMG (2014). Object Constraint Language (OCL), version 2.4, OMG document formal/14-02-03, <https://www.omg.org/spec/OCL/2.4/PDF>.
- Sprogis, A., Barzdins, J. (2012). Specification, configuration and implementation of DSL tools. *Frontiers in Artificial Intelligence and Applications*, vol, 249, pp. 330-343.
- WEB (a). *Flyte - Cloud Native Machine Learning and Data Processing Platform*. <https://flyte.org>
- WEB (b). *Dagster - System for building modern data applications*. <https://github.com/dagster-io/dagster>
- WEB (c). *Metaflow - Framework for real-life data science*. <https://metaflow.org>
- WEB (d). *DVC - Open-source Version Control System for Machine Learning Projects*. <https://dvc.org>
- WEB (e). *Observatory - Solution for tracking machine learning models*. <https://github.com/wmeints/observatory>
- WEB (f). *lab - MLearning Lab*. <https://github.com/beringresearch/lab>
- WEB (g). *Weights&Biases*. <https://www.wandb.com>
- WEB (h). *comet*. <https://www.comet.ml>
- WEB (i). *mlflow - An open source platform for the machine learning lifecycle*. <https://mlflow.org>
- WEB (j). *FGLab - ML Dashboard*. <https://kaixhin.github.io/FGLab>
- WEB (k). *Sacred*. <https://github.com/IDSIA/sacred>
- WEB (l). *guild.ai - The ML Engineering Platform*. <https://guild.ai>
- WEB (m). *Sacredboard - Web dashboard for the Sacred machine learning experiment management tool*. <https://github.com/chovanecm/sacredboard>

Received November 30, 2020, accepted November 30, 2020