MDEForgeWL: Towards cloud-based discovery and composition of model management services

Arsene Indamutsa, Juri Di Rocco, Davide Di Ruscio, Alfonso Pierantonio
Dipartimento di Ingegneria e Scienze dell'Informazione e Matematica
Università degli Studi dell'Aquila
L'Aquila, Italy
{arsene.indamutsa, juri.dirocco, davide.diruscio, alfonso.pierantonio}@univaq.it

***Abstract*—Model management services play an essential role while developing complex systems by means of model-driven engineering (MDE) practices. They carry out several model management operations (MMOs), including model transformation, validation, comparison, and merging, which are exposed as remotely consumable services. However, the adoption of MMOs on cloud-based model repositories has raised issues related to their discovery and orchestration. Notably, it is an arduous and error-prone task to carry out the composition and execution of complex workflows involving different modeling artefacts consumed by various model management services.**

This paper presents *MDEForgeWL*, a complete infrastructure to support the execution of MMO workflows that are remotely available as dedicated services. *MDEForgeWL* consists of *i)* a DSL and supporting engine for defining and executing user-defined workflows of model management services, and *ii)* a cluster infrastructure to register new services and make them available for defining workflows. A prototypical implementation of *MDEForgeWL* is presented by applying it to an illustrative example.

***Index Terms*—Model-driven engineering, Cloud-based model repository, Domain-specific language, Workflow engine, Service discovery, Service composition**

I. Introduction

MDE tackles the issues of software complexity by promoting both abstraction and automation to increase software productivity, and quality [1]. MDE relies heavily on model management operations [2], [3] when engineering large, complex, and interdisciplinary systems [4], [5]. Examples of model management operations are comparing models to identify their differences, merging models to unify certain features and versions, and automated transformations to generate target artefacts out of input ones for simulation, verification, and code generation purposes [5].

The Modeling as a Service (MaaS) initiative [6] fostered the adoption of model management operations as services over the Internet. To this end, over the last years, several repositories have been proposed by both academia and industry to enable the reuse of modeling artefacts and to enable their remote execution as services [4]. Thus, by employing the MaaS paradigm, the development of complex systems requires the composition of several atomic services, which need to be properly discovered, and orchestrated. However, currently available model persistence services do not facilitate such operations mainly because they do not expose remote APIs [7] and do not provide the means to register and discover the services to be composed.

This paper presents the MDEForgeWL language that enables efficient composition and discovery of model management services and modeling artefacts. In particular, by using the proposed DSL, the user can specify workflows to orchestrates underneath model management services. The language is based on a trigger-action paradigm where services can trigger the execution of other ones. Users can plan, organize, customize and execute an arbitrary model-driven task workflow by involving independent model management services in a specific flow to achieve their defined goals [7].

The MDEForgeWL engine is implemented using a microservice oriented architecture by exploiting the Kubernetes technology.[1] Kubernetes offers out-of-box benefits such as auto-scalability, extensibility and dynamic selection of services based on the load [7]. Moreover, Kubernetes permits to discover model management services, and their usage via remote APIs. The code repository of MDEForgeWL is available online.[2] Thus, the main contributions of this paper are the following:

– Support service and model artefacts' discovery through the MDEForgeWL platform;
– Empower the user with a DSL to define custom workflows involving model management services;
– Enable orchestration, abstraction and automation of model management services;
– Facilitate extensibility and scalability of model management services on a cloud-based model repository

The paper is organized as follows: Section II discusses the background, and Section III makes an overview of related works and comparison of existing approaches to compose model management operations. Section IV introduces the architecture of MDEForgeWL. Section V presents the MDEForgeWL language by means of an illustrative example, whereas Section VI concludes the paper.

II. Background

In this section we discuss the background of this work by focusing on aspects related to service-oriented architectures and to the development of domain-specific languages.

This work is funded by the European Union's Horizon 2020 research and innovation programme under the Marie Sklodowska-Curie – ITN grant agreement No 813884

[1]https://kubernetes.io
[2]https://github.com/Indamutsa/model-management-services.git

A. Service-oriented architecture

The current uptrend in service-oriented computing is transforming traditional software systems and infrastructures. This digital transformation involves a shift from a centralized architecture into dynamic and distributed systems that support cloud-based services [8]. This new paradigm uses cloud computing to encapsulate heterogeneous and autonomous services into a service pool that exhibits various functional and non-functional features [9]. Cloud computing is defined by the National Institute of Standards and Technology (NIST) as "*a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction.*" [9]

Migrating to the cloud facilitates affordable access to reliable and high-performance hardware and software resources and cut expenses related to system maintenance and security [9]. Moreover, such migration is a pillar in supporting features such as collaboration, remote reuse, high availability, extensibility of model artefacts, and their management services [10]. In addition, cloud computing offers a plethora of benefits such as on-demand self-service, broad network access, resource pooling, rapid elasticity, measured service, multi-tenacity and auditability and certifiability [9]. Besides, it fosters inter-organizational interaction by enabling service discovery, composition, and execution of their business logic [9]. Hence, to achieve a fully operational complex service, atomic services are combined to process data to achieve the user goal, often referred to as composition [11].

In general, a service is defined as an invokable network, independent high-abstracted and self-contained remote operation that executes low-level functionalities and might return some data [12]. In the MDE context, a model management service is a containerized model management operation (e.g., transformation) along with its engine and auxiliary operations that enable the manipulation of the input modeling artefacts and ensure the return of output data. Service discovery is referred to as the process of finding and querying from a registry services that exhibit given functional and non-functional features. In this aspect, composition stands for the operation of discovery, selecting, combining, and executing cloud-based services to achieve the user's goal [13]. To enable these features, service preconditions, effects, inputs, and outputs are encoded in a computer interpretable form such as a DSL [8].

DSLs used in service composition are designed to support the specification of composite processes, facilitate interoperability between service users and providers, and enable flexible and dynamic invocation of ad-hoc external services [14]. The resulting complex composite services from the complex invocation chain must scale with the number of composing services. Service composition offers two significant benefits to the developer and the user. For the developer, it advances service and artefacts reusability, and from the user's perspective, she has seamless access to a variety of complex services [15].

B. Domain-specific languages

DSLs pave the way for domain experts to leverage their knowledge in the development of otherwise complex functionalities using intuitive text encoded with instructions for machines to execute [16]. They are preferred due to two main reasons: firstly, they unclog a challenging bottleneck in software development: communications among stakeholders and engineers; secondly, they increase productivity among developers [17]. Due to arduous effort involved in developing a domain specific language (DSL), MDE techniques are wielded during their design and implementation [18].

In this context, MDE techniques are used to express solutions at the same level of the problem domain. Developing a DSL comprises several phases that result in a compiler capable of reading the text, parse it, and generate executable code. To realize a DSL, developers take advantage of frameworks such as JetBrains MPS and Xtext [16]. The former offers projectional editing that facilitates parsing the text and thus overcoming the limits of developing DSL editors [14]. As for Xtext[3], it requires a grammar specification and generates the full customizable infrastructure needed to build a fully-fledged domain-specific language. Xtext provides an out-of-box lexical analyzer, parser, abstract syntax tree using EMF model, type checker, compiler, and editing support for Eclipse modeling framework (EMF). Moreover, it supports the Language Server Protocol (LSP) for client-server communications [18].

III. Composition of model management tools

In this section, we make an overview of existing approaches to compose model management tools (see Sec. III-A). Different criteria are also presented to elaborate a comparative table of the analyzed approaches (see Sec. III-B).

A. Overview of related works

Languages such as BPMN, which are general purposes business process languages, tend to be complex due to the vast number of related specifications and notations. As a consequence, they sometimes lead to incorrect interpretation of its elements, and semantics [19], [20]. Moreover, although graphical specification languages such as BPMN presents a solid boundary to achieve defined operations, they tend to score down on flexibility, especially when implementing complex ideas that step out of the fixed boundaries [21]

Build tools such as Gradle require an adequate understanding of their documentation to get started. Moreover, they are not specifically conceived to run model management workflows, requiring extension mechanisms to support MDE artefacts and tools [22]. They can lead to tedious work that is abstracted by our DSL. If a task fails in Gradle, subsequent tasks that depend on the failed one are not executed [22]. We intend to implement self-healing mechanisms within our DSL that do not necessarily halt the program's execution but report on the encountered problem to facilitate troubleshooting adequately. Gradle is a very mature build tool, and we intend to use it to implement a part of our workflow engine to facilitate the task execution process.

[3]https://www.eclipse.org/Xtext/

Alvarez et al. developed MTC Flow [23], a graphical DSL intended for designing, development, and deployment of model transformation chains. Their tools are limited to the Eclipse platform and support only model transformations and validations. In addition, their implementation does not address features related to cloud-based solutions and service discovery.

Modelflow [24] is a more advanced language towards reactive model management workflows. Their implementation depends on events that can trigger a given workflow. Their execution engine can react to modification of resources, and a graph-based execution plan strategy is provided to enable alternative execution paths. However, Modelflow does not involve advanced query mechanisms, service discovery, and other features such as model persistence to remote repositories, or cloud deployment. Furthermore, Modelflow is based on the Epsilon language family, and features related to cloud-based solutions were out of scope.

MoScript [25] is based on the Eclipse platform and the supported model management operations are not cloud-based. It does not support service discovery. Although it can perform model queries within the DSL, they are limited to OCL and directly tied to inner model properties.

Wires [26] is a graphical Eclipse-based tool supporting the orchestration of ATL model transformations. However, it does not support cloud-based orchestration of model management services and their discovery as for the previously mentioned tools.

MMINT [27] is a tool assisting model management operations employing a graphical editor. It provides an interactive user interface, and the user can choose input models and feed them into a transformation, and the output can be used as input for subsequent transformations.

Moola [22] is a Groovy-based model operation orchestration language. It exhibits several features even though it does not support cloud-based solutions, such as cloud-based orchestration of services and advanced query mechanisms.

B. Comparison of model management composition approaches

In this section, we compare the most recent tools addressing the problem of composing model management operations. None of the existing approaches implements the model-as-a-service (MaaS) [28] paradigm. Moreover, as shown in Table I, existing approaches can be analyzed with respect to the features described in the following.

- *Concrete syntax*: It refers to the language used to specify the composition of the considered model management tools. The language syntax can be textual or graphical.
- *Target platform*: It concerns the platform providing the functionalities of the considered approaches. With cloud-based deployment, the tool can be used on the web or through RESTful APIs. However, most of the analyzed tools are based on the Eclipse Modeling Framework (EMF). EMF is the leading open-source modeling framework, and it is no surprise that most of the tools make use of it. Several initiatives have been migrating that infrastructure to the cloud [4] and enable features such as collaborative modeling.
- *Security support*: With this feature, we are interested in understanding how the analyzed tool manages the security layer, e.g., employing security patterns like OAuth2.0.
- *Collaborative development support*: It concerns features to share developed artefacts or even to enable the collaboration between different stakeholders.
- *Reusability*: It concerns available means to reuse already specified artefacts.
- *Scalability*: This feature is related to the architecture used during tool implementation. We evaluate if the system under analysis has some scalability support, e.g., concerning concurrently connected users, data traffic, and data storage.
- *Syntactical & semantic features*: These are features provided by the language server. Although there are several features in this context, we check if their DSLs support auto-completion, syntax/semantic highlighting, and warning and error markers.
- *Language features*: We refer to the availability of language features such as data holders, iterations, and conditional statements. These features are essential in controlling the flow of the workflow specifications.
- *Service heterogeneity*: We aim at checking if the analyzed approach can support services developed and available from different technologies.
- *Service features*: We want to check if the analyzed approach implements the MaaS paradigm. In particular, investigate if related operations are supported, such as service discovery, cloud-based orchestration, and third-party service integration.
- *Program execution*: We check if there is optional execution of the program using sequential or execution means. We also check if the user can specify the service to be executed to execute the wanted model management operation. For instance, she might prefer performing model transformation using ATL[5] rather than ETL[6] at run-time based on some service outcome.
- *Information point*: This feature concerns service and information discovery. Although one can query services based on their types to determine which one to use, the user can still discover services using the documentation with illustrative and straightforward demos.
- *Advanced features*: These are features facilitating the development of complex workflows, such as advanced query mechanisms and workflow pipelines specification.
- *Persistence support*: It concerns the technology employed to store developed specifications, which might be locally saved or pushed to a cloud-based repository.
- *Traceability*: Tracing events and problems that occurred during the execution of a composite service is an essential feature. We check the availability of debugging means such as console view and the capability to gather the logs

[4]https://www.eclipse.org/emfcloud/
[5]https://www.eclipse.org/atl/
[6]https://www.eclipse.org/epsilon/doc/etl/

TABLE I
COMPARISON OF SERVICE COMPOSITION TOOLS FOR MODEL MANAGEMENT OPERATIONS

Feature	**MDEForgeWL**	**Moola**	**MTC Flow**	**Modelflow**	**MoScript**	**Wires**	**MMINT**
Concrete Syntax							
Graphical			✓			✓	✓
Textual	✓	✓		✓	✓		
Target platform							
Cloud-based (Web integration)	✓						
Local infrastructures (Eclipse,...)	✓	✓	✓	✓	✓	✓	✓
Security Support							
In-built security patterns	✓						
Security pattern	✓						
Collaborative development support							
Artefact sharing capabilities	✓						
Sharing configuration	✓						
Reusability							
Code reuse	✓	✓			✓		
artefacts' reuse	✓	✓	✓	✓	✓	✓	
Scalability support							
Number of users	✓						
Data traffic	✓						
Data storage	✓						
Language features							
Data holder	✓	✓		✓	✓		
Condition	✓	✓			✓	✓	
Iteration	✓	✓				✓	✓
Syntactical & semantic features							
Auto-completion	✓						
Syntax highlighting	✓	✓	✓	✓	✓	✓	✓
Warning & Error markers	✓						
Service heterogeneity							
Model management	✓	✓	✓	✓	✓	✓	✓
Non MMSs	✓						
Service features							
Service-oriented (MaaS, SaaS, ...)	✓						
Service discovery	✓						
Service composition	✓	✓	✓	✓	✓	✓	✓
Cloud-based orchestration	✓						
Third party service integration	✓						
Program execution							
Sequential	✓	✓	✓	✓	✓	✓	✓
Parallel	✓	✓					
Alternative service execution	✓		✓				
Knowledge base							
Documentation	✓						
Query mechanisms	✓				✓		
Advanced features							
Advanced query mechanisms	✓						
Workflow pipelines' specification	✓						
Persistence support							
Cloud-based model repository	✓						
Local file system		✓	✓	✓	✓	✓	✓
Traceability							
Debugging means	✓						
Service call logs	✓						

of the service calls.

Table I shows the result of the analysis we performed on the existing tools. In the next section we present the proposed MDEForgeWL approach, aiming at supporting all the previously presented features.

IV. THE PROPOSED MDEFORGEWL PLATFORM

Figure 1 shows an overview of the proposed MDEForgeWL architecture, which has been designed to support the definition and execution of scalable workflows consisting of cloud-based compositions of model management services. The architecture relies upon and extends MDEForge [4] by adding discovery mechanisms to identify available services, which can be involved in the workflows being executed. The architecture is organized into four tiers: the client tier, the execution engine, the cluster of model management services, and the persistence layer. In the following, the four tiers of the proposed architecture are singularly described.

A. The MDEForgeWL Frontend

It provides users with a cloud-based modeling environment to specify and execute workflows of model management services. To this end, the MDEForgeWL language is defined as detailed in Section V. The language is implemented using the Xtext [18] framework. Thus, the language can be deployed as an Eclipse plugin. Moreover, the Xtext framework provides an efficient web integration that supports JavaScript text editors in web applications [18]. Thus, in our implementation, we used the Ace text editor[7] and the LSP services offered by

[7]https://ace.c9.io/

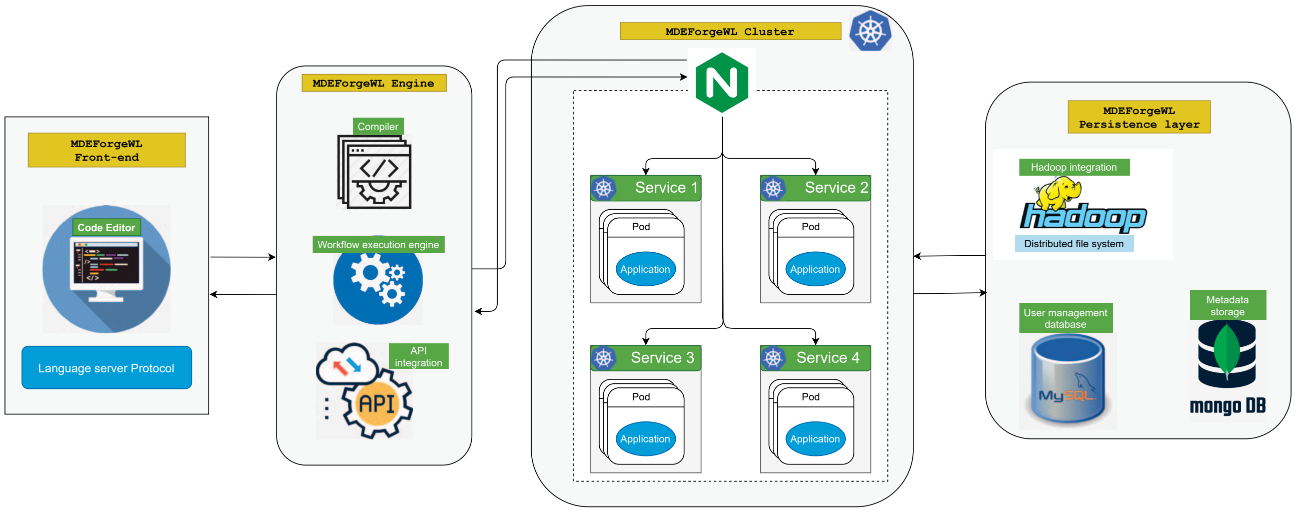

Fig. 1. Overview of the MDEForgeWL architecture

Xtext. Our language server defines a JSON-RPC protocol that enables implementation, integration, and independent distribution of language support features within the code editor within our ecosystem. Currently, the developed client environment supports content-assistance functionalities (code completion), automatic validation (syntactical, cross-reference and concrete syntax validation that ensures grammar constraints are respected), syntax highlighting, occurrence marking, hover information and formatting [29]. The front-end is implemented using the Vue.js framework,[8] and we provide users with a console to facilitate transparent monitoring of user workflow composition and debugging activities.

B. The MDEForgeWL Engine

Our engine comprises a compiler, an execution engine, an API integration component, and the language server-side that supports the language editor. Our compiler, which is a subprocess, consumes the text from the code editor. Next, the lexer lexically analyzes the text, and the extracted tokens conform to the building blocks of our language, such as keywords and statements. The parser takes the list of incoming tokens and generates the abstract syntax tree (AST). The AST generated by Xtext is an EMF model, and the model is traversed using the EMF API. Once the AST is available from the incoming DSL text, a code generation process is triggered. Technically, code generation in Xtext traverses the AST and translates the tree into executable code that conforms to the language of your choice, in our case, JavaScript. Our language is statically typed, and the data structures we intend to support are arrays and nested objects.

At last, the compiler returns a valid executable code that can be run by the execution engine. The execution engine runs the provided executable code and uses the API integration component to leverage services provided by the MDEForgeWL cluster. For instance, when a user-defined workflow requires the definition of some available model management services, the engine triggers the orchestration of the involved services at the cluster. They get executed asynchronously (parallel) or in sequence based on user preferences. Another sub-process, the language server, is also running in the engine behind the scene to provide server-side functionalities to the client code editor. It is important to remark that each service (e.g., a service exposing model transformation functionalities) can have several engines (e.g., ATL, and ETL), and the user can choose which one should be used. With our discovery mechanism, the user can find out which engines are available. The system selects the right engine based on different criteria determined by the container orchestrator and API gateway. Moreover, the workflow engine is entangled with logging and monitoring mechanisms that keep track of the execution of workflows. For example, we keep track and visualize API calls using services such as Prometheus[9], Grafana[10] and Zipkin[11]. We have also implemented within the workflow engine distributed logging mechanisms to monitor the progress of the workflow executions and facilitate troubleshooting.

C. The MDEForgeWL Cluster

This DSL can be used as a plugin in Eclipse platform, but our endeavors aim at migrating model-driven development infrastructures from locally environment to the cloud. In this aspect, we can ensure our modeling infrastructures are more scalable and extensible than in traditional fashion of modeling. The cluster is built using Kubernetes, an open-source container-orchestration platform [30]. We use it to automate deployment, scale, and manage our containerized model management services into logical units that facilitate their discovery. The Kubernetes cluster offers several features: service discovery and load balancing, self-healing, horizontal

[8]https://vuejs.org/

[9]https://prometheus.io
[10]https://grafana.com
[11]https://zipkin.io

scaling, automatic bin packing, storage orchestration, secret and configuration management, and batch execution. In addition, the Kubernetes cluster is designed to be extensible and loose-coupled to facilitate feature update without hardcore changes to mainstream code-base and architecture [30]. We rely on the Kubernetes ingress controller to accept and load balance the traffic to the microservices. It also manages egress traffic, representing communications from internal to external services out of the cluster. In addition, the ingress controller monitors running pods within the cluster and automatically updates load-balancing rules regarding removed or added services.

Adoption of containerization technology to build cloud-native microservices, accelerates the development process. Containers are inherently portable and are built to ensure effective isolation and efficiency of resources [30]. Self-healing mechanisms enable containers to be advertised only when they are ready to serve, and can be killed, restarted, replaced or rescheduled to conform to health check as defined by the user. Containers are scaled based on CPU usage to balance application workload. This is enabled by assigning a single DNS name for a set of pods, which is referred to as a service, thus all communication are made through the service and the service load-balance the workload among the bootstrapped pods [30]. Since the MDEForgeWL cluster is deployed using Google Kubernetes Engine[12], the system administrator is allowed to set any resource limits, e.g., on storage, CPU, and memory usage. Kubernetes auto-scales resources based on available maximum and minimum ones or replicas set by the administrator. It has built-in vertical, horizontal, cluster autoscalers. Based on current usage, desired target, and user demands, autoscalers scale up or down the number of running pods or replicas, perform dynamic management of CPU/memory utilization of machines inside the cluster and increase or decrease the number of nodes where pods are running [31].

Self-contained, fully-fledged model management services are organized in a distributed microservice architecture that ensures their resilience, security, loose-coupling, flexibility, fault-tolerance, extensibility, and scalability. These microservices are referred to as a *resource server*. Moreover, other services such as automated clustering of model artifacts, search engine integration, and model metrics calculator are integrated at this level. To access our cluster, we use the Nginx ingress controller to interact with underneath microservices.

Our orchestration and discovery approach uses current trending containerization and orchestration technologies that automate the manual work related to service discovery activities. Existing discovery approaches mainly rely on WSDL documents [32]. In particular, typically, clients are expected to read and process WSDL files to determine the services exposed by the server of interest. To call the services listed in the analyzed WSDL file, the user uses SOAP over transfer protocols like HTTP [33].

The proposed microservice architecture also includes a service registry, a service API gateway, authorization & authentication server (it implements the OAuth2.0 protocol[13]) and a resource server as shown in Fig 2. When a user accesses the web browser via a service endpoint published by the Nginx ingress controller to request the resource server, it goes through the service API-gateway. The service API-gateway cross-checks the credentials to validate the user authentication. If the user is not authenticated, the service API-gateway redirects her to the authorization & authentication server. The server asks the user to authenticate and issues an access token which enables her to access the resource server. The resource server ensures the access token is valid from the authorization server, and then it is set to execute the request. All resource servers implement a client discovery feature to publish their service. The service registry server keeps an open connection to discover and register all self-published services from the resource servers. The service API-gateway fetches all available services from the service registry and acts as a proxy server to the resource server. Briefly, registering new services with MDEForgeWL is done by implementing a client discovery that publish the implemented service. Our engine by the use of service registry server, will discover and register it in our registry. Once the service is registered, it can be used by our API gateway as MDEForgeWL services. Extending services in this manner does not require modifications of the grammar of the DSL.

D. The MDEForgeWL Persistence Layer

The persistence layer of the proposed system is divided into three categories. The first category is used to store structured data using SQL databases such as user management services or other sensitive data. The second part stores in NoSQL databases unstructured data (such as logs or data mined by data mining services from the MDEForgeWL cluster). The last part persists artefacts such as models, metamodels, and transformations. MDEForge, our cloud-based model repository, consists of model management services that allow persistence and management of typical modeling artifacts and tools. Services are accessed used through RESTful Web APIs [4]. Our repository is built to handle big data with features such as high velocity, volume, and variety and perform analytics and predictions on store data. A Hadoop cluster is the most pragmatic way to manage big data, break down big problems into smaller elements and enable effective analysis and predictions on the stored data. As in the case of Kubernetes, the Hadoop cluster is self-healing and supports dynamic addition and removal of servers from the cluster [34].

V. The MDEForgeWL language

The ultimate goal of this work is to have a cloud-based domain specific language to specify workflows of model management services. Although this DSL can be used in standalone mode, it is also the backbone of the Low-Code Development Environment [7] based on drag and drop capabilities. As shown in Fig. 3, the environment aims at enabling citizens developers to plan, organize and execute their workflows of model management services. Within, the platform, the user

[12]https://cloud.google.com/kubernetes-engine

[13]https://oauth.net/2/

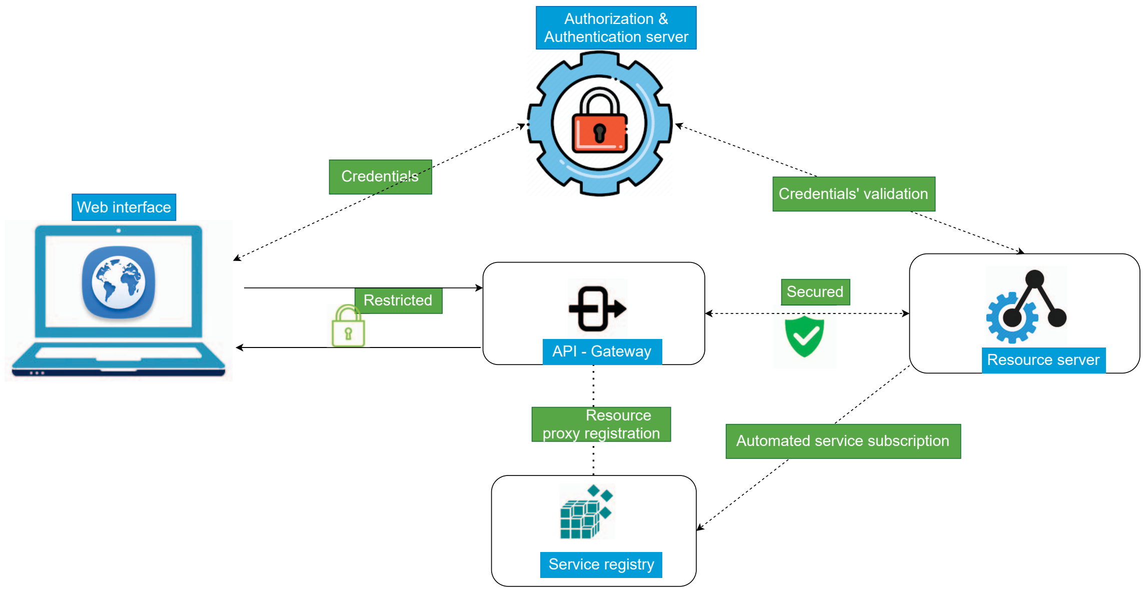

Fig. 2. Detailed view of the MDEForgeWL cluster

can opt to use custom scripting to complement his defined workflow in the graphical interface, or develop an entirely new workflow by using the custom scripting option provided by the domain specific language we are presenting. These features concerning the custom scripting are under development. The incoming model from the graphical interface in a format of xmi/json is transformed to this DSL and execute by our workflow engine.

The domain-specific language we are proposing, should be able to perform necessary low-level functionalities needed in a program but also achieve complex functionalities with reduced efforts for the user. To achieve this objective, the language needs to be declarative: you specify what you want and we deliver the results according to the given specification. As previously mentioned, MDEForgeWL has been developed in Xtext, and a fragment of the corresponding metamodel is shown in Fig.4. Each specification has `WorkflowProgramModel` as root model element. Each workflow consists of several elements, e.g., statements, workflow blocks, methods, or functions. A statement contains features such as variables or other callable statements including method invocations. Within statements, we can have expressions, conditional and loop statements that assist the control flow during the execution of the specification composition. A workflow block is made up of steps that contain statements. The statement can also be a service, i.e., one of the model management services managed by the cluster. Furthermore, a statement can be a query to identify artifacts of interest.

Within our language, services are an abstraction of model management operations (e.g., model transformations, validations, model query, and model comparison operations). These are the functionalities that are containerized and deployed individually in the cluster. Executing model management operations like model transformations via traditional techniques could be complex, time-consuming, and error-prone, and it usually requires specific frameworks that need to be installed locally. With our approach, this can be achieved by calling specific services with corresponding arguments as shown below.

```
// We perform the transformation, the etl script is retrieved by
    ↪ id
call service _transfoModel(sourceModel, sourceMetamodel,
    ↪ targetMetamodel, id: 4)
```

Listing 1. Service call example

The argument can be a variable or the identifier of the artifact stored in the repository. As shown in Listing 2, you can use advanced query mechanisms to search and find a metamodel based on several criteria, including parameters and properties set at the repository level. This feature is convenient because it permits users to specify query predicates to find the artifacts satisfying given properties. In particular, as shown in line 2 of Listing 2 2, the language permits to declare variables, which are evaluated and used later in the script. For instance, the variable `sourceModel` can be queried using its `id`, `type`, and `extension` type. This is a query that returns a single result if successful. The returned results carry other information as well, such as the execution status. The user can check if an executed query succeeded before carrying on with subsequent commands.

```
1 //Create variables : This part is improved by advanced query
    ↪ mechanisms. You can query the type of models u want based
    ↪  on your defined criterias
2 var sourceModel = query artefact(id: 1, type: model, ext: xmi)
3 var sourceMetamodel = query artefact(
4           type: metamodel, ext: ecore, hasModel: sourceModel)
5
6 var targetMetamodel = query artefact(
7         user: "john", period: (03,2020 - 2021), hasAttribute: "
            ↪ person",
8         size: <500kb) -> retrieve( startsWith: "catalogue",
9         contains: "class book").first
```

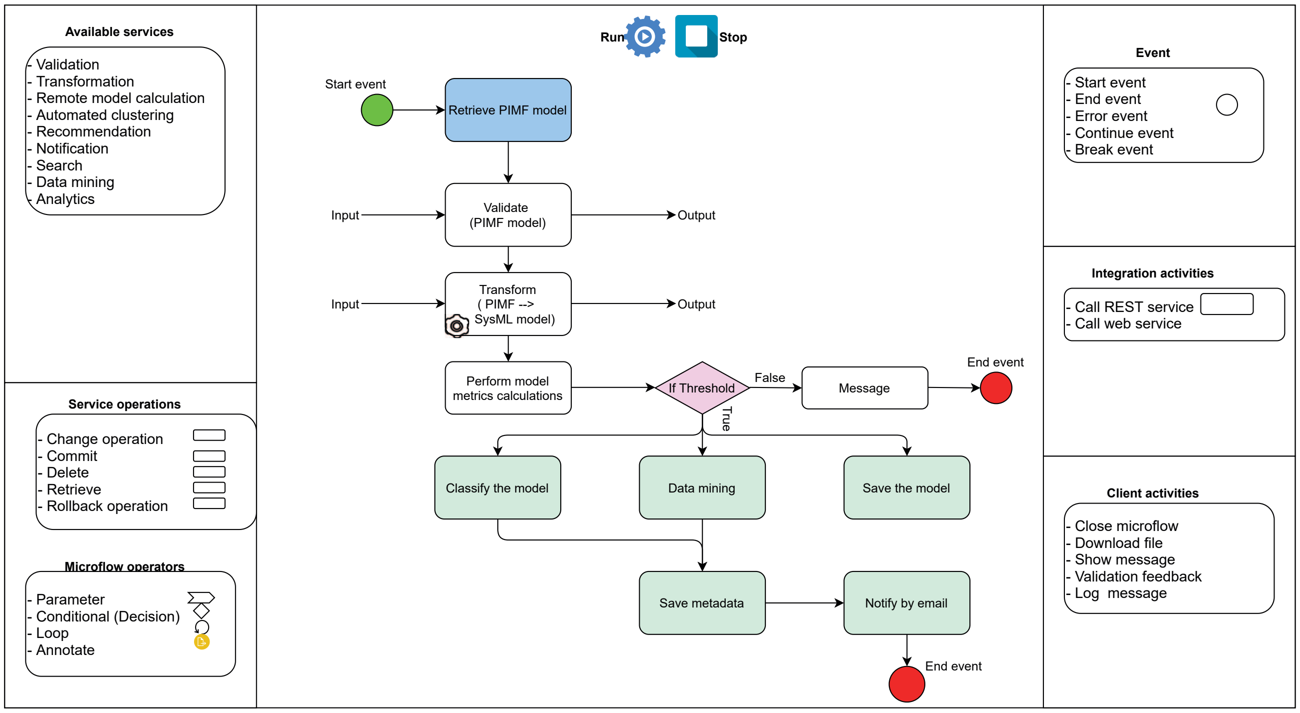

Fig. 3. Proposed LCDP that rely on our domain-specific language [7]

```
10
11 var model1 = query artefact( name: "catalogue.xmi", conformsTo: "
       ↪ catalogues.ecore",
12         sharedUsername: ["john"], sharedUserNumber: < 3 )
13 var emlscript = query artefact(id: "23")
14 var eclscript = query artefact(id: "44")
15 var eolscript = query artefact(id: "12")
16
17 Workflow workflow type:sequence{
18    step "Validate"{
19       // Let's validate our model with the retrieved ecl script
20       global var eventValid = call service _validateModel(
21               sourceModel, sourceMetamodel, evlscript)
22       }
23    }
24    step "Compare Transform Merge Persist"{
25       // We will proceed if the validation passed
26       if(eventValid){
27         // We perform the transformation, the etl script is
              ↪ retrieved by id
28         var targetModel = call service _transfoModel(
29       sourceModel, sourceMetamodel, targetMetamodel, id: 4)
30          //If there is a matched trace, we can merge some model
               ↪ aspects
31          var matchedTrace = call service _compareModel(
32       model1, targetModel, eclscript)
33      if(matchedTrace){
34         // We merge the models, and persist the merged model and
              ↪ target model
35             var mergedModel = call service _mergeModels(
36                model1, model2, eclscript, emlscript)
37             call persistArtifact(targetModel, mergedModel)
38      }
39       }
40    }
41 Post{  // We can notify the user of the outcome of the workflow
42     call notify(email: "johndoe@email.net", message: "message")
43 }
44
45 Execute workflow()
```

Listing 2. An illustrative MDEForgeWL specification

The query at line 3 concerns a metamodel with the exact model to which the previous query retrieved the model. The query at lines 6-9 is more complex and is used to search for an artifact with a certain user and persisted on the repository in the specified period. We query the model's content to find if it has a certain attribute and its size is larger than 500kb. Let us suppose the query returns more than one result that meets the criteria specified. We can pipe the returned results and specify additional criteria such as the artifact name starting with a given text or the artifact containing a given text literally. On the returned collection, we can retrieve the first result. It is possible to retrieve models according to their name, the metamodel they conform to, the shared username, and the number of users that is shared with (see line 11). Such properties and parameters are set on the repository level, and our DSL is aware of information regarding the retrieved artifacts. In addition, the user can query the services and choose which one to use based on its functionality, such as a transformation to be executed.

The information about service executions is displayed in a dedicated console. Control flow statements such as conditions and loops are also supported. However, these functionalities are deemed low-level, hence their use is discouraged since the user can still achieve the same results by piping results for further processing of the query. It is important to remark that the language is still undergoing development. Still, we currently support basic data types such as booleans, numbers, and strings, conditional statements, loop constructs, and func-

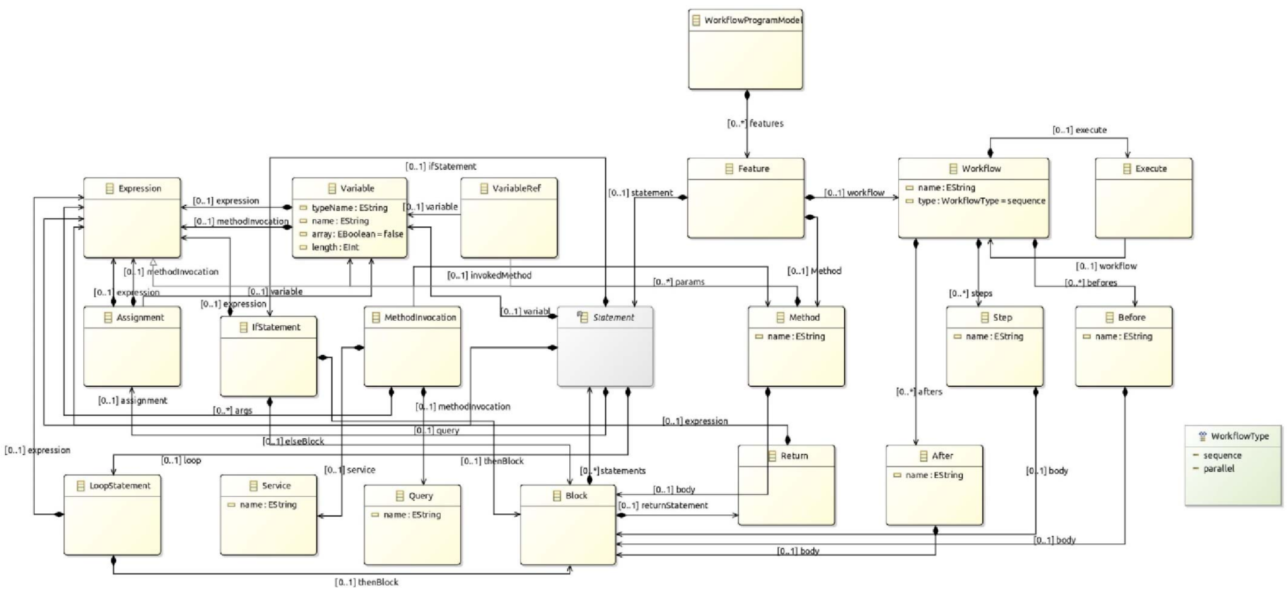

Fig. 4. Fragment of the MDEForgeWL metamodel

tions. We intend to support in the future also data structures such as objects and arrays.

MDEForgeWL specifications can have a single workflow code block. You can specify the execution pattern, such as sequential or parallel (see the workflow definition at line 17). The user can declare dependencies within steps and among different steps. Step blocks enable pipeline and batch execution of code. For instance, the user might have a step where she wants to validate the model (see lines 18-23) before performing a model transformation or perform some model testing before the subsequent operations (see the step defined at lines 24-39).

In the second step of the explanatory workflow, we make use of conditional statements (see line 33) to match the traces from the model comparison operation before we can merge the models (see line 35). The resulted model from the model merging operation is persisted using one line of code (see line 37). The user can specify the models to be persisted by delimiting them with a comma. Underneath, the engine saves models and ensures their relationships with other artifacts such as the metamodels they conforms to. The user can specify `pre` and `post` code blocks for workflows (e.g., see line 41). In this instance we chose to perform an non-model service regarding notifying the user about the results. Notifying the user using emails or other notification services is not the only way we use to reflect the progress status of the considered workflow; we can also use the console view to reflect exception logs captured by the platform. In addition, we intend to embed visualization capabilities to reflect logs about the program execution progress and eventual results in real-time to the user.

When the user cancels the workflow execution, the program preemptively forestalls the subsequent executions of the workflow and returns the current status. Although it is out of the scope of the present work, we plan to integrate abilities to pause and resume the execution of the workflows by persisting the current state in the context object and pass it to the interpreter to resume the paused execution. The actual execution of the specified workflow has to be triggered by means of the `Execute` statement as in line 45.

VI. Conclusion and Future work

This paper presented the MDEForgeWL platform to support the discovery and composition of model management operations, which are consumable as remote services. The objective of the proposed approach is to foster the usage of model management service orchestrations and enable modelers to access a variety of composite services satisfying defined requirements. A prototypical implementation of the approach is available, and its architecture has been presented. Further than being cloud-based, the distinguishing characteristics of the system include its capability of discovering and composing model management services according to declaratively specified workflows. As future work, we plan to finalize the development of the presented approach, and we intend to perform a user evaluation to assess the usability of the proposed modeling pipelines. In addition, we want to add advanced query mechanisms with an underlying model search engine and support data structures such as arrays and objects.

References

[1] B. Selic, “What will it take? A view on adoption of model-based methods in practice,” *Software and Systems Modeling*, vol. 11, no. 4, pp. 513–526, 2012.

[2] B. Sanchez, D. S. Kolovos, and R. Paige, “Modelflow: Towards reactive model management workflows,” in *Proceedings of the 17th ACM SIGPLAN International Workshop on Domain-Specific Modeling*, ser. DSM 2019. New York, NY, USA: Association for Computing Machinery, 2019, p. 30–39. [Online]. Available: https://doi.org/10.1145/3358501.3361238

[3] A. García-Domínguez, D. S. Kolovos, L. M. Rose, R. F. Paige, and I. Medina-Bulo, "Eunit: A unit testing framework for model management tasks," in *Model Driven Engineering Languages and Systems*, J. Whittle, T. Clark, and T. Kühne, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 395–409.

[4] F. Basciani, J. Di Rocco, D. Di Ruscio, A. Di Salle, L. Iovino, and A. Pierantonio, "MDEForge: An extensible Web-based modeling platform," *CEUR Workshop Proceedings*, vol. 1242, no. September, pp. 66–75, 2014.

[5] L. Berardinelli, A. Mazak, O. Alt, and M. Wimmer, "Model-driven systems engineering: Principles and application in the cpps domain," *Multi-Disciplinary Engineering for Cyber-Physical Production Systems: Data Models and Software Solutions for Handling Complex Engineering Projects*, pp. 261–299, 05 2017.

[6] H. Brunelière, J. Cabot, and F. Jouault, "Combining Model-Driven Engineering and Cloud Computing," in *MDA4ServiceCloud'10 Workshop co-located with ECMFA*, Jun. 2010.

[7] A. Indamutsa, D. Di Ruscio, and A. Pierantonio, "A low-code development environment to orchestrate model management services," in *Advances in Production Management Systems. Artificial Intelligence for Sustainable and Resilient Production Systems*. Cham: Springer International Publishing, 2021, pp. 342–350.

[8] S. Pang, Q. Gao, T. Liu, H. He, G. Xu, and K. Liang, "A Behavior Based Trustworthy Service Composition Discovery Approach in Cloud Environment," *IEEE Access*, vol. 7, pp. 56 492–56 503, 2019.

[9] A. Jula, E. Sundararajan, and Z. Othman, "Cloud computing service composition: A systematic literature review," *Expert Systems with Applications*, vol. 41, no. 8, pp. 3809–3824, 2014. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0957417413009925

[10] J. Di Rocco, D. Di Ruscio, L. Iovino, and A. Pierantonio, "Collaborative repositories in model-driven engineering," *IEEE Software*, vol. 32, no. 3, pp. 28–34, 2015.

[11] A. L. Lemos, F. Daniel, and B. Benatallah, "Web service composition: A survey of techniques and tools," *ACM Computing Surveys*, vol. 48, no. 3, 2015.

[12] P. Rodriguez-Mier, C. Pedrinaci, M. Lama, and M. Mucientes, "An integrated semantic web service discovery and composition framework," *IEEE Transactions on Services Computing*, vol. 9, no. 4, pp. 537–550, 2016.

[13] D. Martin, M. Paolucci, S. McIlraith, M. Burstein, D. McDermott, D. McGuinness, B. Parsia, T. Payne, M. Sabou, M. Solanki, N. Srinivasan, and K. Sycara, "Bringing semantics to web services: The owls approach," in *Semantic Web Services and Web Process Composition*, J. Cardoso and A. Sheth, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 26–42.

[14] G. Cordasco, M. D'Auria, A. Negro, V. Scarano, and C. Spagnuolo, "Toward a domain-specific language for scientific workflow-based applications on multicloud system," *Concurrency Computation*, no. February, 2020.

[15] G. Stürmer, J. Mangler, and E. Schikuta, "A domain specific language and workflow execution engine to enable dynamic workflows," *Proceedings - 2009 IEEE International Symposium on Parallel and Distributed Processing with Applications, ISPA 2009*, pp. 653–658, 2009.

[16] M. Völter, S. Benz, C. Dietrich, B. Engelmann, M. Helander, L. C. L. Kats, E. Visser, and G. Wachsmuth, *DSL Engineering - Designing, Implementing and Using Domain-Specific Languages*. dslbook.org, 2013. [Online]. Available: http://www.dslbook.org

[17] M. Fowler, *Domain-specific languages*. Pearson Education, 2010.

[18] L. Bettini, *Implementing domain-specific languages with Xtext and Xtend*. Packt Publishing Ltd, 2016.

[19] M. Cortes-Cornax, S. Dupuy-Chessa, and D. Rieu, "Choreographies in bpmn 2.0: new challenges and open questions," in *Proceedings of the 4th Central-European Workshop on Services and their Composition, ZEUS*, vol. 847. Citeseer, 2012, pp. 50–57.

[20] M. Von Rosing, S. A. White, F. Cummins, and H. De Man, "Business process model and notation-BPMN," *The Complete Business Process Handbook: Body of Knowledge from Process Modeling to BPM*, vol. 1, no. January, pp. 429–453, 2014.

[21] M. Hjorth, "Strengths and weaknesses of a visual programming language in a learning context with children," 2017. [Online]. Available: http://www.diva-portal.org/smash/record.jsf?pid=diva2%3A1111152&dswid=-8735

[22] A. G.-b. M. O. Orchestration, "A Groovy-based Model Operation Orchestration Language," vol. 2017.

[23] C. Alvarez and R. Casallas, "MTC Flow: A tool to design, develop and deploy model transformation chains," *ACadeMics Tooling with Eclipse, ACME 2013 - A Joint ECMFA/ECSA/ECOOP Workshop*, 2013.

[24] B. Sanchez, D. S. Kolovos, and R. Paige, "Modelflow: Towards reactive model management workflows," *DSM 2019 - Proceedings of the 17th ACM SIGPLAN International Workshop on Domain-Specific Modeling, co-located with SPLASH 2019*, pp. 30–39, 2019.

[25] W. Kling, F. Jouault, D. Wagelaar, M. Brambilla, and J. Cabot, "MoScript: A DSL for querying and manipulating model repositories," *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 6940 LNCS, pp. 180–200, 2012.

[26] J. E. Rivera, D. Ruiz-Gonzalez, F. Lopez-Romero, J. Bautista, and A. Vallecillo, "Orchestrating ATL Model Transformations," *Proc. of MtATL 2009*, no. June, pp. 34–46, 2009.

[27] A. Di Sandro, R. Salay, M. Famelis, S. Kokaly, and M. Chechik, "MMINT: A graphical tool for interactive model management," *CEUR Workshop Proceedings*, vol. 1554, pp. 16–19, 2015.

[28] M. Brambilla, J. Cabot, and M. Wimmer, *Model-Driven Software Engineering in Practice*, 2012.

[29] M. Eysholdt and H. Behrens, "Xtext - Implement your language faster than the quick and dirty way tutorial summary," *Proceedings of the ACM International Conference Companion on Object Oriented Programming Systems Languages and Applications Companion, SPLASH '10*, pp. 307–309, 2010.

[30] J. Arundel and J. Domingus, *Cloud Native DevOps with Kubernetes*, 2019.

[31] kubernetes.io. (2021) Kubernetes documentation. [Online]. Available: https://kubernetes.io/docs/home/

[32] W3C. (2007) Web services description language (wsdl) version 2.0 part 1: Core language. [Online]. Available: https://www.w3.org/TR/wsdl20/

[33] S. Mallick, R. Pandey, S. Neupane, S. Mishra, and D. S. Kushwaha, "Simplifying Web service discovery & validating service composition," *Proceedings - 2011 IEEE World Congress on Services, SERVICES 2011*, pp. 288–294, 2011.

[34] H. V. Karambelkar, *Scaling Big Data with Hadoop and Solr Second Edition*, 2015.