A Domain Specific Language to Provide Middleware for Interoperability among SaaS and DaaS/DBaaS through a Metamodel Approach

Babacar Mane[1], Ana Patricia Magalhaes[2,3], Gustavo Quinteiro[1], Rita Suzana Pitangueira Maciel[1] and Daniela Barreiro Claro[1]

[1]*Formalisms and Semantic Applications Research Group (FORMAS), Federal University of Bahia – DCC – IME, Av. Adhemar de Barros, Ondina, 40170-110, Salvador, Bahia, Brazil*

[2]*Department of Exact Sciences and Earth, State University of Bahia, 2555 Silveira Martins st. Cabula, Bahia, Brazil*

[3]*Post Graduate Program in Computing and Systems, Salvador University, 251 Dr. Jose Peroba st., Bahia, Brazil*

Keywords: Metamodel, Cloud Computing, Model Driven Development, Interoperability.

Abstract: Cloud Computing (CC) is a paradigm that manages a pool of virtualized resources at infrastructure, platform, and software levels to deliver them as services over the Internet. Cloud Platforms are heterogeneous, and therefore cloud users may face interoperability and integration issues regarding consumption, provisioning, management, and supervision resources among distinct clouds. Due to the lack of standards in such a heterogeneous environment, an organization may face a lock-in situation. A middleware can minimize the effort to overcome lock-in problems. The MIDAS middleware ensures semantic interoperability between Software as a Service (SaaS) and Data as a Service (DaaS), and at the same times provides data integration between DaaS. Currently, MIDAS runtime implementations rely on Cloud Foundry, Amazon Web Services, OpenShift and, Heroku providers. To avoid ambiguity in MIDAS development and deployment an unambiguous definition of MIDAS architectural concepts must be provided. Thus, our work presents a Domain-Specific Modeling Language (DSML) comprising a metamodel of MIDAS semantic architecture and a Unified Modeling Language (UML) profile. To evaluate the DSML expressiveness, we instantiate several middleware models, and the findings demonstrate that our modeling language has an acceptable level of concepts to specify the middleware.

1 INTRODUCTION

Cloud Computing (CC) is a paradigm that manages a pool of virtualized resources at infrastructure, platform, and software levels to deliver them as services over the Internet (Shawish and Salama, 2014). These services are classified by the National Institute of Standards and US Technology (NIST) into Software as a Service (SaaS), Platform as a Service (PaaS), and Infrastructure as a Service (IaaS) respectively. CC is organized into three deployment models according to providers' access policies (private, public, and hybrid cloud). Everything in the cloud is considered as a service. For instance, a database is provided as a service (Database as a Service - DBaaS), and Data are also offered as a service (Data as a Service - DaaS). In the cloud market, providers are categorized into proprietary and open-source platforms, and each of them has their application programming interfaces (API), protocol, and data format to provide their services (Zhang et al., 2013).

In such a heterogeneous environment, organizations that migrate their applications or exchange their data from one provider to another may stay in a lock-in situation due to the lack of standard solutions. These situations can occur between the same service level of distinct clouds caused by horizontal heterogeneity (e.g., distinct DaaS or different DBaaS) or among different service levels of clouds arising from the vertical heterogeneity (e.g., SaaS and DaaS or SaaS and DBaaS) (Ranabahu and Sheth, 2010; Sanaei et al., 2013).

To solve the lock-in situation, portability or interoperability solutions have been proposed (Lewis, 2013; Maciel et al., 2016). Reuven Cohen defines a portability solution as the ability to move data or application components regardless of the provider's operating system, storage, data format, or API (Cohen, 2009). Meanwhile, an interoperability solution has been defined by the Institute of Electrical and Electronics Engineers (IEEE) as the ability to exchange and mutually use information among systems. Sys-

Mane, B., Magalhaes, A., Quinteiro, G., Maciel, R. and Claro, D.
A Domain Specific Language to Provide Middleware for Interoperability among SaaS and DaaS/DBaaS through a Metamodel Approach.
DOI: 10.5220/0010459600830094
In *Proceedings of the 23rd International Conference on Enterprise Information Systems (ICEIS 2021) - Volume 1*, pages 83-94
ISBN: 978-989-758-509-8

tems in different clouds should exchange and operate data transparently among independent platforms. Interoperability provides transparency when exchanging data between multiple clouds and also managing cloud resources with the same customized tools (Lewis, 2013). Three levels of interoperability are defined (Maciel et al., 2016): syntactic, semantic and pragmatic. In a syntactic interoperability context, different applications exchange data or messages to perform an activity with a common protocol. A semantic interoperability solution requires meaning on data exchange to disambiguate each message communication. A pragmatic interoperability solution ensures that an intended effect of a message is exchanged between applications or systems.

DaaS and DBaaS providers store and manage a high volume of heterogeneous data produced by, for instance, mobile computing, ubiquitous devices, social networks, and web-based applications. Such data are distributed geographically and available to consumers or organizations. However, to access similar data distributed from distinct DaaS/DBaaS providers, users from SaaS must connect to each provider at a time and then use a mechanism (e.g., a program) to process or aggregate data. For instance, a demographic researcher will do the same thing to get census data provided by governments from different DaaS.

To mitigate the vertical heterogeneity issue among SaaS and DaaS/DBaaS levels, an automated solution so-called MIDAS (Middleware for DaaS/DBaaS and SaaS) has been developed as an intermediate communication layer (Mane et al., 2020a; Mane et al., 2020b). MIDAS allows a cloud user to transform its request in a single real-time Structured Query Language (SQL) or Not Only SQL (NoSQL) query to retrieve data at once from distinct DaaS/DBaaS. This avoids data migration, ensures semantic interoperability, and enables DaaS and DBaaS data integration.

Currently, MIDAS runtime implementations are published on Cloud Foundry, Amazon Web Services, OpenShift and, Heroku PaaS providers. However, differents PaaS offer heterogeneous tools and services (Gonidis et al., 2013). Thus, to be able to deploy MIDAS across multiple PaaS, without lock-in to a particular vendor, model-based approaches can be exploited to develop, maintain, and reuse cloud applications.

Our work proposes a Domain-Specific Modeling Language (DSML) for MIDAS architecture with well-defined syntax and semantics. It enables the specification of the MIDAS platform-independent architecture within software components and connections. The DSML abstract syntax and static semantics are specified following the Meta-Object Facility (MOF) metamodel and the concrete syntax using a UML profile. This DSML assists software project to guide developers in a model-driven development project as a base for model transformations (Da Silva, 2015). To evaluate the DSML expressiveness, we instantiate several middleware models, and we conclude that the modeling language has an acceptable coverage level of the concepts to define a middleware.

The remainder of this paper is organized as follows: **Section 2** presents the State of Art; **Section 3** describes some related work; **Section 4** depicts our domain specific modeling language (DSML); **Section 5** presents the metamodel validation; and finally, **Section 6** presents the conclusion and future work.

2 BACKGROUND

In this section, we present a brief overview of models and modeling language concepts.

2.1 Domain Modeling Languages

Model-Driven Development (MDD) is a model-based approach emerging as one of the leading software engineering approaches employing models as a primary artifact to maintain the development process on track and be able to implement an increasingly reliable software design (Christensen and Ellingsen, 2016).

Models are abstract representations of a system, which comprise both the structure and the behavior(Mellor, 2004). In MDD, models are the fundamental artifacts, and they produce other models as well as the application source code. Models must be specified according to a modeling language with well-defined syntax and semantic. A model is a set of elements describing some aspects of a system with an abstraction level, and it is a real authentic representation simplified and contextualized (Mellor, 2004). In MDD, models are more than documentation; they facilitate and improve communication among technical and non-technical stakeholders by sharing the same vision and knowledge. During the systems development process, the model concept provides a product that is easier to understand through its graphical representations.

In MDD, modeling languages are employed to specify models graphically, textually, or both, and they are always based on specific domain requirements in use. These languages are classified into two categories: one is specified to a domain (a knowledge area), called DSML (Domain-Specific Modeling Language) and, the other is a general-purpose, called GPL (General Purpose Languages), such as the

Unified Modeling Language (UML) (Da Silva, 2015). GPL is widely used in different application fields and provides a higher number of generic constructs instead of DSMLs that use a few constructs or concepts tailored to the needs of a specific application domain and make models more expressive. For each application domain, it is often necessary to create a new DSML; thus, this task can generate a new language implementing, maintaining, and learning cost. A General-Purpose Language (e.g., UML) can be customized for specific domains based on the specification of profiles and does not change the original semantic elements. A profile definition comprises the specification of stereotypes, meta-classes, and labeled values.

Modeling languages formally write models with a well defined syntax and semantic, and are composed of four main elements: abstract syntax, concrete syntax, static semantics, and dynamic semantics (Da Silva, 2015).

- Abstract Syntax: this conceptualizes, and synthesizes the application domain knowledge by identifying concept names close to the application domain and by establishing particular relations among them. Language constructors are defined in the abstract syntax;
- Concrete Syntax: This is either textual or graphical. The graphical concrete syntax is often defined by the structure of the abstract syntax and a set of graphical representations for classes and associations in the abstract syntax (Herrera et al., 2016). It defines how abstract syntax features are presented to users (e.g., UML profile);
- Static Semantics: Constraints and rules define relations among concept names to ensure static semantics. The Object Constraint Language (OCL) is a declarative constraint language that is the most used in this case [1];
- Dynamic semantics: This defines how instantiated models are executed.

In MDD, the abstract syntax and static semantics of a DSML is usually expressed in terms of metamodels. According to OMG[2] a metamodel is a model of models. Models built along the MDD-based software development process are metamodel instances, and should always be in accordance with their metamodel. Models must satisfy restrictions defined by the metamodel and should be syntactically correct concerning the metamodel. Through a metamodel we can document our models, reuse, debug and execute them without its source code.

[1]https://www.omg.org/spec/OCL/

[2]https://www.omg.org//

3 RELATED WORK

Diversities of the cloud PaaS environment (a set of application framework, runtime environment, and programming languages) do not permit multi-cloud applications development and deployment from one cloud to another. Solutions to mitigate lock-in issues in the cloud can face a challenge when deploying or implementing them in another cloud. A way to solve this issue is to employ model-based approach features to simplify solutions design, management, and migration across multiple clouds. In this section, we highlight some works from researchers who have attempted to solve the same problem.

Researchers in (Merle et al., 2015) propose a metamodel for Open Cloud Computing Interface (OCCI) to ensure interoperability at IaaS, PaaS and SaaS levels. OCCI core concepts are specified in natural language and interpreted in various ways during their implementation. To avoid an imprecise, ambiguous, and incomplete OCCI specification, the static semantic of the OCCI metamodel is defined in the Object Constraint Language (OCL). This proposes a precise type classification system, an extensible datatype system, both extension and configuration concepts. Authors in (Zalila et al., 2019; Zalila et al., 2017) then extends the previous work of (Merle et al., 2015) providing an OCCIware approach to allow design, validate, generate, implement, deploy, execute, and supervise everything as a service with OCCI.

Cloud Modelling Framework (CloudMF) (Ferry et al., 2018) is another approach relying on a model-driven concept to simplify the complexity of maintaining and evolving complex applications potentially deployed across multiple cloud platforms. CloudMF captures cloud applications provisioning and deployment in platform-independent models and automates their operations and adaptations through a models at run-time environment. The Cloud Application Modelling Language (CAML) (Bergmayr et al., 2014) is a UML internal language developed during the ARTIST EU project to represent cloud-based deployment topologies, and to capture cloud environment-specific information from different perspectives with a set of UML profiles. Following the PIM/PSM concept in the MDA framework, CAML manages to separate cloud-provider into independent and specific deployment models. CAML does not provide a runtime environment to perform the multi-cloud application deployment. There are some configuration management tools such as Cloudify[3], Puppet[4], Chef[5], and

[3]https://cloudify.co/

[4]https://puppet.com/

[5]https://www.chef.io/products/chef-infra/

Ansible[6] providing Domain Specific Modeling Language (DSML) to facilitate the specification of an application's configurations, services, their dependencies, deployment, monitoring, adaptation, and execution plans, without employing a language-dependent of platform.

In contrast to the metamodel architecture of MIDAS, the approaches mentioned above are conceived DSML automatically or semi-automatically from design-time to the run-time to allow multi-cloud applications to deploy from a particular cloud to another. Our work provides a DSML to specify the MIDAS independent platform that can generate code in any specific platform.

4 OUR DOMAIN SPECIFIC LANGUAGE

To address the MIDAS deployment or development issue across multiple clouds, we rely on the MDD approach to simplify MIDAS' design, implementation, and migration from one cloud to another. This section presents MIDAS DSML for the specification of MIDAS middleware models independent of the platform. The abstract syntax and static semantic of our DSML are specified in the metamodel presented in (Section 4.2), and the concrete syntax is represented as a UML profile in (Section 4.3). This language makes it possible to define models independent of the platform using UML diagrams. First, we provide an introduction to the MIDAS architecture.

4.1 MIDAS Architecture

To mitigate vertical heterogeneity issues among SaaS and DaaS/DBaaS, an automated solution so-called MIDAS (Middleware for DaaS/DBaaS and SaaS) has been developed as an intermediate communication layer to allow the retrieval of data from DaaS or DBaaS through a single real-time Structured Query Language (SQL) or Not Only SQL (NoSQL) (Mane et al., 2020a; Mane et al., 2020b). This solution avoids data migration, and ensures semantic interoperability among SaaS and DaaS/DBaaS and, data integration among DaaS and/or DBaaS.

MIDAS is developed in two phases following the interoperability types classification: syntactic and semantic. The MIDAS syntactic interoperability is composed of three modules (**Request, Data, and Result**), a Dataset Information Storage (DIS), and an external *Crawler* to update DIS. To provide a semantic solution in MIDAS, a (*semantic*) module is appended to the syntactic interoperability. *Figure* 1 shows the syntactic and semantic architecture of MIDAS.

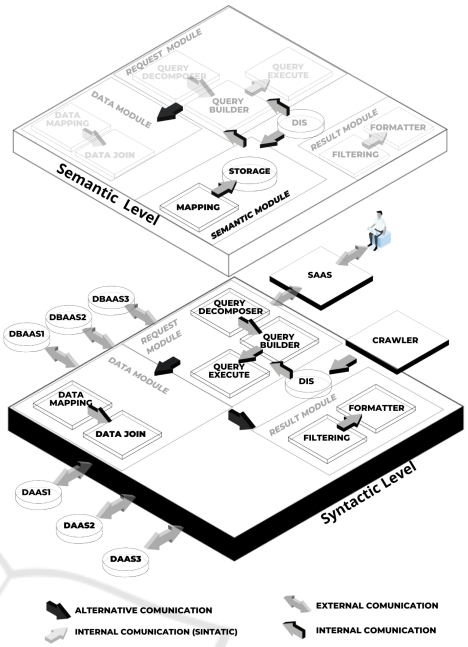

Figure 1: MIDAS Architecture.

The **Request Module** *Query Decomposer* component receives users queries from SaaS providers, breaks them into terms, and sends results to the *Query Builder* component to build DaaS APIs. Before setting DaaS APIs, the *Query Builder* component always check terms consistency accessing **Dataset Information Storage** (DIS) storing DaaS APIs information. If some terms are different, it accesses the *Semantic Module Storage* database to find terms which are semantically similar. After that, DaaS APIs are send to be performed by the **Request Module** *Query Execute* component and the results are forwarded to the **Result Module** *Filtering* component. If user requests contain a *Join* SQL or a *lookup* NoSQL clause allowing users to get data from more than one DaaS or DBaaS, the **Data Module** *Data Join* component aggregates the results of DaaS APIs executed, and send them to the *Filtering* component.

The **Data Module Data Mapping** component receives DBaaS query terms from the *Query Builder* component decomposed and datasets credentials stored in DIS to build DBaaS APIs and execute them at the corresponding DBaaS. If *join* SQL or *lookup* NoSQL clause are employed in user queries, data are aggregated by the *Data Join* component and sent to the *Filtering* component to be filtered. Otherwise, data are sent directly to the *Filtering* component.

[6]https://www.ansible.com/

The *Filtering* component can receive data from the components *Data Join*, *Query Execute* or **Data Mapping** to be filtered. This component is responsible for removing some irrelevant attributes (e.g., id) before sending the data to the *Formatter* component to be formatted into comma-separated-values (csv), Extensible Markup Language (xml) or JavaScript Object Notation (json) format. **Result Module** manages indistinctly users data from DBaaS or DaaS.

The *Mapping* component of **Semantic Module** shares DIS database with *Query Builder* component to manage APIs attributes evolution. Each time the DIS database is updated by the MIDAS *Crawler*, the *Mapping* component identifies the semantic similarity between old and new DaaS attributes evolution and it stores the results in the *Storage* component in a tree data structure. The *Crawler* is out of MIDAS' architecture, and it is developed to maintain and update semi-automatically DIS database. A temporary data structure is created to store users' terms query change in order to allow the *Formatter* component to send data to the user with the parameters defined in the original query.

Currently, MIDAS runtime implementations are published on Cloud Foundry, Amazon Web Services, OpenShift and, Heroku PaaS providers. However, different PaaS offer heterogeneous tools and services (Gonidis et al., 2013). To be able to deploy MIDAS across multiple PaaS, without lock-in to a particular vendor, model-based approaches can be exploited to develop, maintain, and reuse cloud applications.

4.2 MIDAS Architecture Metamodel

This section presents our metamodel to represent the semantic interoperability domain. It is defined in the M2 layer of the OMG model, and is organized into one package as shown in **Figure** 2. The metamodel is specified following the Meta-Object Facility (MOF) meta-metalanguage (OMG layer M3), and will be described in detail in the following subsection.

The MIDAS semantic architecture metamodel is implemented on the Eclipse Modeling Framework (EMF), and it provides a metamodeling technology called Ecore to encode the metamodel structure. It is inspired by the OMG Model-Driven Architecture (MDA) [7], and it represents the Platform-Independent Model (PIM). It is agnostic to any development paradigm and technology, and it enables developers to implement their software by choosing their preferred programming languages and frameworks.

[7]https://www.omg.org/mda/

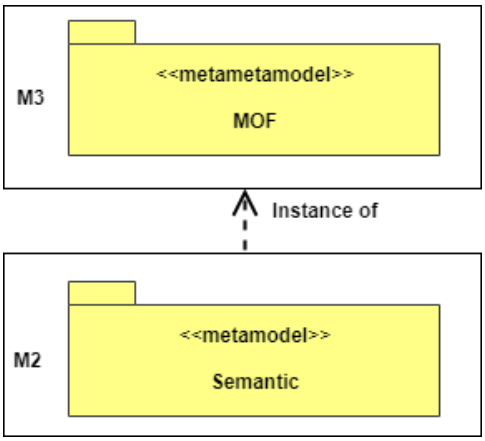

Figure 2: MIDAS architecture metamodel representation at different levels of abstraction.

4.2.1 Metamodel of the MIDAS Semantic Architecture

Our metamodel of the MIDAS is depicted in **Figure** 3. It comprises the concepts (as classes) and their corresponding properties according to the domain of a semantic interoperability middleware.

In the context of semantic interoperability, a Middleware (**Middleware** class in **Figure** 3) has a name (*middlewareName* attribute) and can be specialized into two types: **APIData**, and **Module**.

The **APIData** exhibits the DaaS/DBaaS provider information, and it is associated to several sets of attributes (e.g., search_path, query_param, filters_param, newFields_param in class **Attribute**) identified by the attribute (*domain*). A *module* represents MIDAS functionalities. The four main types of *module* are *Request*, *Data*, *Result*, and *Semantic*. Each type expresses a MIDAS functionality:

- *Request Module*: Identifies by the propriety *requestName*, and it transforms the user query into API links;
- *Data Module*: Manages data from user queries, and it has *dataName* as its attribute, to identify it;
- *Result Module*: It has a name (*resultName* attribute), and it formats data accordingly to users requests;
- *Semantic Module*: This is identified by the propriety *semanticName*, and it manages DaaS attributes updating in *APIData* to identify semantic similarity between previous and new attributes so as to execute transparently user queries.

The **Request Module** is composed by a *QueryDecomposer*, a *QueryBuilder*, and a *QueryExecute*, represented as class. The *QueryDecomposer* decomposes user query in terms, and it has three properties

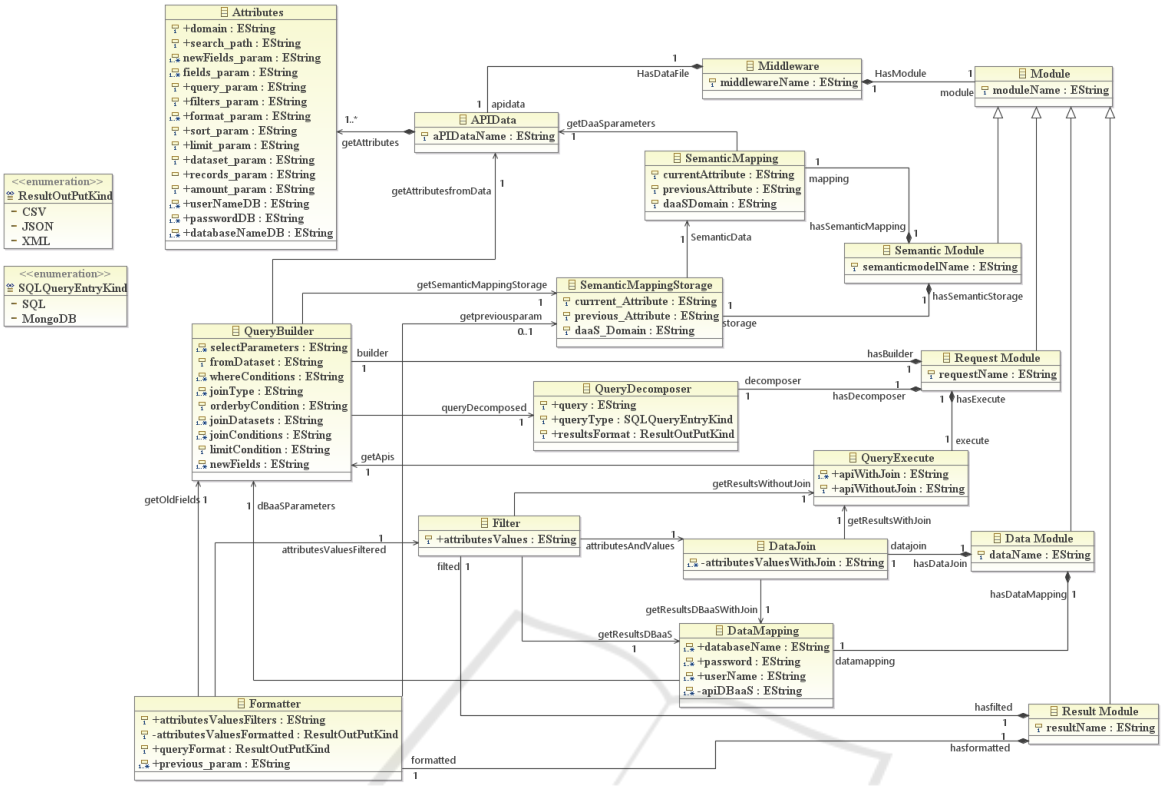

Figure 3: Metamodel of the MIDAS semantic architecture.

a *query*, a *queryType*, and a *resultsFormat*. The *query* represents user query, the queryType defines its type (e.g., Structured Query Language, or MongoDB), and the *resultsFormat* exhibits the data formatted type provided by the user (e.g., csv, json, or xml). Users can choose the data type format they wish to receive. The *queryType* value defines how to decompose a user query in terms. The *QueryBuilder* verifies the user query terms compatibility with DaaS/DBaaS providers information represented by **APIData** before building an API link. This has eight properties *selectParameters*, *fromDataset*, *whereConditions*, *joinType*, *orderbyCondition*, *joinDatasets*, *joinConditions*, and *limitCondition* representing user query terms. The class *QueryExecute* has two properties *apiWithJoin*, and *apiWithoutJoin* to represent two types of APIs links: an API link from a user query formulated without *Join* clause (*apiWithoutJoin*), and a set of APIs links gained from a user query with *Join* clause (*apiWithJoin*). An ≪enumeration≫ class named *SQLQueryEntryKind* is created to define the two types of a user query: Structured Query Language (SQL) and MongoDB.

The **Data Module** comprises a *DataJoin*, and a *DataMapping*, represented as classes. The class *DataMapping* has four attributes *databaseName*, *password*, *userName*, and *apiDBaaS*. It is responsible for building DBaaS API and executing them to get data from DBaaS providers. The properties *password*, and *userName* represent user credentials, and databaseName the database name. The attribute *apiDBaaS* represents DBaaS APIs formatted by the *DataMapping*. The *DataJoin* class aggregates data obtained from API Links (*apiWithJoin* attribute, and *apiDBaaS*).

The **Result Module** consists of a (*Filter*, and a *Formatter*) represented as class. The *Filter* has one propriety *attributesValues*, which represents data collected from DaaS/DBaaS providers. Some unnecessary data parameters are removed for user information. The *Formatter* has four attributes the *queryFormat*, the *attributesValuesFormatted*, the *attributesValuesFilters*, and the *previous_param*. The *attributesValuesFilters* represents data filtered by the *Filter*, and the *queryFormat* exhibits three types of data format (e.g., csv, json or xml). *Previous_param* represents user query terms identified in the *QueryBuilder* as incompatible with DaaS/DBaaS represented by **APIData**. To return the query result conform request by the user with the same parameters, these terms are used to format data filtered. The attribute *attributesValuesFormatted* represents the data format-

ted in one of the format types represented by the attribute *queryFormat*. An ≪enumeration≫ class named *ResultOutPutKind* is created to define three types of data format: CSV, JSON, and XML.

SemanticMapping manages the attributes of DaaS evolution represented in *APIData* to avoid mismatching with user query terms. *SemanticMappingStorage* organizes these attributes per DaaS. The *SemanticMapping* has three properties *currentAttribute*, *previousAttribute*, and *daaSDomain*. The *currentAttribute*, and *previousAttribute* represents semantically similar attributes, and the *daaSDomain* attribute exhibits DaaS domain. The *SemanticMappingStorage* has three attributes *current_Attribute*, *previous_Attribute*, and *daaS_Domain* that organize for each DaaS domain, semantically similar attributes to maintain user requests transparent. This new organization allows the *QueryBuilder* class to format API links even when there is a mismatch among user' query terms and attributes represented by the **APIData** class. To do this, the *QueryBuilder* class has one more property (*newFields*) that represents the semantic similar attributes with user query terms. The *Formatter* class has more than one propriety *previous_param* to assign user terms incompatible with attributes represented in the **APIData** class. This property can format the user's data with the previous parameters defined in the query.

4.3 UML Profile of the Metamodel of MIDAS

This section presents the concrete syntax of our language, specified as a UML profile. A UML profile is an extension mechanism defining domain-specific languages using UML concepts. The definition of the UML profile named **pmidas** contains the stereotypes specification, metaclass and labeled values. Stereotypes are labels that can be applied to various UML elements, and extend metaclass representing UML concepts to which the stereotypes can be applied. Finally, the labeled values assign values to the extended UML concepts. Terminologies, syntax, and notations are introduced to provide additional semantics for existing concepts. Furthermore, new meta-attributes (target values), new meta-associations, and new meta-enumeration are defined. This extension mechanism does not allow modification of the underlying metamodel.

The stereotypes names in the UML profile **pmidas** are practically maintained in relation to MIDAS metamodel concepts. For instance, the *QueryDecomposer* concept of MIDAS metamodel corresponds to the ≪*QueryDecomposer stereotype*≫ of the *pmidas* profile applied to class ≪*metaclass*≫. The **pmidas** profile is defined in the *pmidas* package, illustrated in Table 1.

Table 1 shows the corresponding stereotypes created, and the metaclass in which the stereotype can be applied during development.

Table 1: Stereotypes and their instantiated metaclass.

Stereotypes	Metaclass
QueryDecomposer, QueryBuilder, QueryExecute	Class
APIData, Attributes, DataMapping, DataJoin	Class
SemanticMapping, SemanticMappingStorage	Class
Formatter, Filter	Class
queryDecomposed, getApis, dBaaSParameters	Association
getResultsDBaaSWithJoin, getResultsDBaaS	Association
getOldFields, getResultsWithoutJoin	Association
AttributesAndValues, AttributesValuesFiltered	Association
getAttributesfromData, getResultsWithJoin	Association
semanticData, getDaaSParameters	Association
getSemanticMappingStorage	Association
SQLQueryEntryKind, ResultOutPutKind	Enumeration

Our metamodel which represents the semantic interoperability of MIDAS architecture will be evaluated qualitatively instantiating its models.

5 VALIDATION

The DSML proposed in this work comprises a metamodel, and a UML profile. Unlike programs, metamodels are not executable artifacts, thus validation cannot be performed using test cases (Magalhães et al., 2015). The expressiveness of a modeling language is usually assessed instantiating models according to its metamodel. The DSML is said to be expressive when it represents user requirements in a natural way to software developers. The concept expressiveness measures whether the expressivity concepts levels are enough to capture the main aspects of MIDAS middleware.

This section presents the assessment of the MIDAS DSML expressiveness. The validation goal is summarized using the GQM (Van Solingen et al., 2002) template as follows:

- **Analyze** the MIDAS DSML;
- **For the purpose of** evaluating expressiveness;
- **With respect to** coverage of the DSML constructors;
- **From the perspective of** software developers;
- **In the context of** the existing metamodel of MIDAS.

The questions underlying the validation are:

- **Q1:** Are the DSML constructors sufficient to specify aspects of semantic interoperability in the middleware?

- **Q2:** Are the selected UML diagrams sufficient to represent the middleware?

We use several measures as dependent variables: the number of constructors used, the need for new constructors, the number of existing changes in the constructors, and the UML diagrams used. The constructor may be represented by classes, associations among classes or attributes. The validation process lasted one month and was performed by a researcher who has more than three years of experience on the MDD approach, and a knowledge of middleware concepts. It consists of the use of MIDAS DSML to define current middlewares.

Three scenarios with three types of user query are defined to validate our models: a user query in a SQL statement, a user query in MongoDB, and a user query in a SQL statement with a *join* clause. We describe each of them below:

1: A User Query with SQL Statement.

- User Request: **SELECT** id, name, address, city **FROM** nyc-wifi-hotspot-locations **WHERE** city = ”New York” **ORDER BY** id **LIMIT** 1

This user's request obtains from the DaaS provider (*nyc-wifi-hotspot-locations*), the *id*, the *name*, and the *address* of every wifi hotspot located in New York. The number of records needed is limited to one, and is classified by values of the parameter *id*.

This user request retrieves from DaaS provider (*nyc-wifi-hotspot-locations*), the *id*, the *name*, and the *address* of every wifi hotspot located in New York. The number of records is limited to one, and it is classified by values of the parameter *id*. As the DaaS provider *nyc-wifi-hotspot-locations* parameter *name* is updated to *names*, the user's query term *name* must be changed to *names* in MIDAS.

Figure 4 shows the instance represented in a class diagram stereotyped using our profile. To facilitate readability in this model we named class with the same name as the stereotype, but this is not necessary. So, three properties of *≪QueryDecomposer stereotype≫* represent *query* = ”**SELECT** id, name, address, city **FROM** nyc-wifi-hotspot-locations **WHERE** city = ”New York” **ORDER BY** id **LIMIT** 1”, *queryType* = ”SQL”, and *resultsFormat* =”JSON”).

The user's query is decomposed into terms and they are assigned to five properties defined in the *≪QueryBuilder stereotype≫* (*selectParameters*=”id, name, address, city”, *whereConditions*=”city=”New York””, *orderbyCondition*=”id”, *limitCondition*=1, and *fromDataset*=”nyc-wifi-hotspot-locations”). The compatibility of these terms are verified in the *nyc-wifi-hotspot-locations* DaaS parameters represented in the *≪APIData stereotype≫*. It was found that the term *name* is evolved to the term *names* in the *nyc-wifi-hotspot-locations* DaaS, and it is managed by the *≪SemanticMapping stereotype≫* with their three properties (*currentAttribute*=”names”, *previousAttribute*=”
name”, and *daaSDomain*=”nyc-wifi-hotspot-locations”), and it organizes by the *≪SemanticMappingStorage≫* through their three attributes (*current_Attribute*=”names”, *previous_Attribute*=
”name”, and *daaS_Domain*=”nyc-wifi-hotspot-locations”). The ”nyc-wifi-hotspot-locations” DaaS API is created with the value of the attribute *current_Attribute* represented by the *≪QueryBuilder stereotype≫* property (*new_Fields*=”names”). The *≪QueryExecute stereotype≫* represent the API Link with its propriety (`apiWithoutJoin = "https://data.nyc-wifi-hotspot-locations.com/ browser?dataset=data.nyc-wifi-hotspot-locations&fields=id,names,address,city"`). The data obtained from the API Link are represented by *≪Filter stereotype≫* attribute (`attributesValues="["id":"1359", "name":" Metropolitan Museum of Art","address":"1000 Fifth Avenue", "city":"New York"]"`). The ”id” parameter is removed for a better understanding of the data by the user. *≪Formatter stereotype≫* formats the data in the type represented by (*queryFormat*=”JSON”), and it exchanges the attribute ”names” represented by *≪QueryBuilder stereotype≫* propriety (*new_Fields*=”names”) with the attribute ”name” represented by the propriety (*previous_param=”name”*). The user data formatted is represented by the attribute (*attributesValuesFormatted*=””name”:”
Metropolitan Museum of Art”,”address”:”1000 Fifth Avenue”,”
city”:”New York””).

The user query is a request to get data from one DaaS, and does not define a *join* clause in statement. For this reason, *≪DataJoin stereotype≫* and *≪DataMapping stereotype≫* are not described in this section. *≪DataJoin stereotype≫* aggregate data from DaaS/DBaaS providers, and the *≪DataMapping stereotype≫* get data from DBaaS providers.

2: A User Query with MongoDB Statement.

This user request is a conversion of the user request previously defined in SQL format to a MongoDB query. In this case, the user should get the same result obtained by the SQL query. The DaaS provider *nyc-wifi-hotspot-locations* is updated, and the parameter

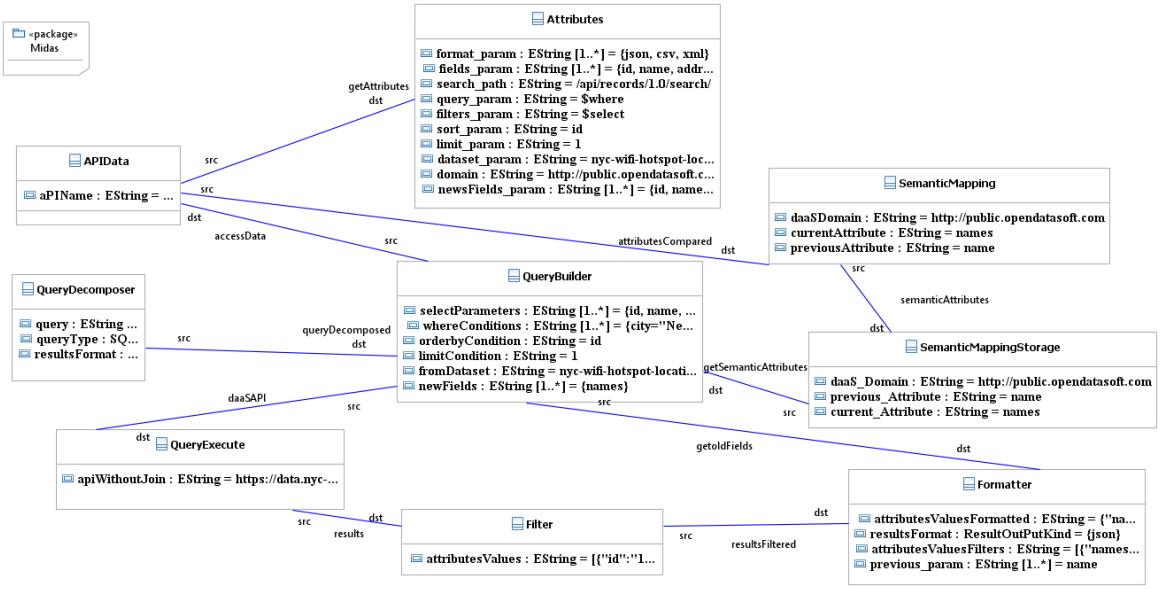

Figure 4: Executing user SQL query.

name is change to *names*.

- User Request: db.nyc-wifi-hotspot-locations.find(”city”: ”New York”, ’id’: 1, ’name’: 1, ’address’: 1, ’city’: 1).limit(1).sort(’id’: ’);

We employed models in **Figure** 4 to test the user MongoDB query. This model shows the instance represented in a class diagram stereotyped using our profile, and the name of classes are the same as the stereotype to facilitate readability. The execution process is the same as performed by the user SQL query. The classes stereotype and attributes used to execute user SQL query is the same as employed by the user MongoDB query. It is worth mentioning that in the *≪QueryBuilder stereotype≫* the properties used to represent the user SQL query terms are the same as those employed for the user MongoDB query terms. This is possible to identify the equivalence of clause statements between the two types of query.

3: A User Query with SQL Statement Employing *join* Clause.

This user SQL query retrieves user *name, email*, and *ages* that have the same *cpf* from the three DaaS (DaaS1, DaaS2, and DaaS3). We observe that the term *ages* is evolved to *age* in the DaaS3 provider.

- User Request: **SELECT** DaaS1.name, DaaS2.email, DaaS3.ages **FROM** DaaS1 **INNER JOIN** DaaS2 **ON** DaaS2.cpf = DaaS1.cpf **INNER JOIN** DaaS3 **ON** DaaS3.cpf = DaaS1.cpf;

The user’s query is decomposed into terms and assigned to five properties defined in the *≪QueryBuilder stereotype≫* (*selectParameters*=”DaaS1.name, DaaS2.email, DaaS3.ages”, *joinType*=”INNER JOIN, INNER JOIN”, *joinDatasets*=”DaaS2,DaaS3”, *joinConditions*=” DaaS2.cpf = DaaS1.cpf, DaaS3.cpf = DaaS1.cpf ”, and *fromDataset*=” DaaS1”).

The compatibility of these terms are verified in DaaS1, DaaS2 and DaaS3 parameters represented in the *≪APIData stereotype≫*. It was found that the term *DaaS3.ages* is evolved to the term *DaaS3.age* in the DaaS3. Therefore, this term is managed by the *≪SemanticMapping stereotype≫* with their three properties (*currentAttribute*=”age”, *previousAttribute*=”ages”, and *daaSDomain*=” data.fipe-parallelum.rhcloud.com”), and organized by the *≪SemanticMappingStorage≫* through three attributes (*current_Attribute*=”age”, *previous_Attribute*= ”ages”, and *daaS_Domain*=”data.fipe-parallelum.rhcloud.com”). The attribute *newFields* defined in the *≪QueryBuilder stereotype≫* represents the new term ”*age*”. The three APIs links are represented by the propriety *aPILinkDaaS* in the *≪QueryExecute stereotype≫* Table 2:

In the *≪DataJoin stereotype≫*, results from the three API links are represented by the attributes *resultDaaS*. These data are aggregated in the *≪Filter stereotype≫*, and represented by the propriety *resultsDaaS*= ””_id: Objectid(”4578”), ’name’:’João’, ’email’:’dao@ufba.br’, ’age’:’20’,”_id: Objectid(”4589”)’name’:’Maria’,

Table 2: Three APIs links.

https://data.fipe-parallelum.rhcloud.com/browser?dataset=DaaS3&fields=age,cpf
https://data.cityofnewyork.us/browser?dataset=DaaS2&fields=email,cpf
https://data.public.opendatasoft.com/browser?dataset=DaaS1&fields=name,cpf

'email':'lia@ufba.br','age':
'20'". The *_id* parameter is removed from the data to make the information understandable for the user. The result is represented by the attribute *resultsFiltered*='name':'João', 'email':'dao@ufba.br','age':'20',"name':'Maria', 'email':'lia@ufba.br','age':'20' in the ≪*ResultFormatter stereotype*≫. The previous term of DaaS3, *ages* is recovered and replace the term *age*. The user query final result is represented by the ≪*ResultFormatter stereotype*≫ propriety *resultsFormatted*='name':'João', 'email':'dao@ ufba.br','ages':'20','name':'Maria', 'email':'lia@ufba.br','age':'20' in a JSON format specified by the attribute *resultFormat*="JSON". **Figure** 5 shows the number of constructors (classes, associations, and attributes) employed to perform the user query. Compare to **Figure** 4, we have a new class named ≪*DataJoin stereotype*≫ to perform terms belonging to the user query *Join*. None of the three user queries employed the ≪*DataMapping stereotype*≫ to not specify a DBaaS provider in their requests. Each attribute defined in the ≪*QueryBuilder stereotype*≫ is related to the name of a clause stated in the user SQL query.

Related to question **Q1**, the results obtained executing the three types of user queries defined above allow us to append and separate some MIDAS DSML constructors. Classes as the *QueryExecute* is appended to our metamodel, the *DataJoin* are separated from the *DataMapping* to only aggregate data from DaaS/DBaaS providers, and the class defined in the **Result Module** is broken down into two classes: the *Filter* and the *Formatter*.

The *QueryExecute* class is created to manage DaaS' API links built by the *QueryBuilder* class. The *DataJoin* class is constructed to handle data from DaaS/DBaaS' API links built according to a user query with *Join* or *lookup* clause. The *DataMapping* is responsible for building DBaaS APIs and getting data from DBaaS providers. The *Filter* class removes some unnecessary data parameters for easier to user understanding, and the *Formatter* class is responsible to format the user data into JSON, XML or CSV type.

Besides the creation of classes *SemanticMappingStorage*, and *SemanticMapping stereotype* to handle the semantic interoperability concept in our metamodel, three properties *previous_param="name"*, *newFields*, and *newFields_param* are defined respectively in the *Formatter*, *QueryBuilder*, and *Attributes*.

News associations are created among the *QueryExecute* and the two classes: *DataJoin* and *Filter*. We also have an association between *QueryExecute* and *QueryBuilder* class. The division of the class defined in the **Result Module** established a new association between the *Filter* and the *Formatter* class. Two news properties *previous_param="name"*, and *newFields* defined respectively in *Formatter*, and *QueryBuilder* create association relations among *SemanticMappingStorage* and the two classes *Formatter* and *QueryBuilder*. The attribute *newFields_param* defined in the *Attributes* class establishes an association relation between the *Attributes* class and the *SemanticMapping*.

Related to question **Q2**, the UML class diagram adopted by our profile was sufficient to specify instantiation of the syntactic, and semantic aspects of the user query. Therefore, we consider that the DSML was stable to be used.

We conclude that the MIDAS DSML constructors are sufficient to specify the middleware and stable enough to be employed. The examples used covered the concepts needed within DSML. For now, they no longer need to be changed or added within DSML. This may happen in the future to change or accommodate some concepts that the examples did not contemplate. The expressiveness of the language is evaluated according to the level of comprehension of the language constructors. This means we assess whether the defined metamodel presented the necessary concepts to instantiate the models. We know that our validation results are limited, and the application of the DSML in other scenarios is necessary to improve the approach.

6 THREATS TO VALIDITY

Some threats of validity were identified in this process validation. This section describes threats to internal validity, external validity, and construction validity.

Internal Validity. Threats to internal validity are related to the possibility of uncontrolled factors influencing our results. The participant's experience can be decisive in the metamodel validation. The

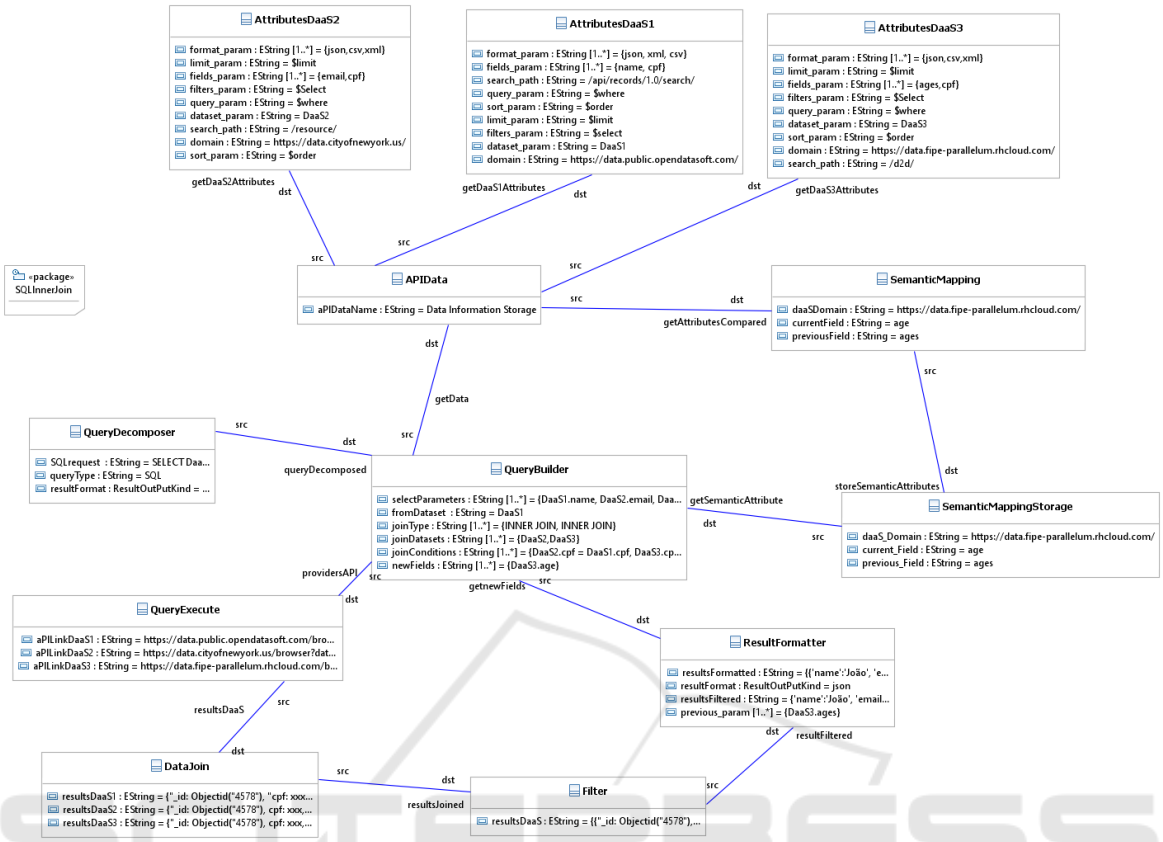

Figure 5: Executing user's SQL query with *join* clause.

expert who is responsible for validating the DSML expressiveness has more than three years experience with the MDD approach, and knowledge in middleware and cloud computing concepts. Therefore, the threats related to the participant's level of experience are quite few.

External Validity. Threats to external validity are related to the possibility of generalizing the results to other participants than the expert. MDD is a recent approach relevant in some areas of development in the industry, and academia (Whittle et al., 2013). However, as it is a recent approach and therefore requires a specialized profile, it is difficult to find a participant with experience in MDD approach, middleware, and cloud computing concepts. It is necessary to confirm the results obtained using a participant with experience in the MDD approach, middleware but not in MIDAS, and in cloud computing concept.

Construction Validity. The construction threats to the validity are related to the measures employed by the expert, whether they really represent what is intended to be evaluated. Assessing the DSML expressiveness is not a trivial task, as it involves participants performing tasks. As a consequence, the evaluation depends on the experience of each participant in MDD approach, middleware and cloud computing concepts.

Despite all these threats, efforts were made to avoid bias in this validation process.

7 CONCLUSION AND FUTURE WORK

In this paper, to enable implementing, running, and deploying MIDAS on a large scale despite the diversity of some cloud development and runtime environment characteristics, we propose a MIDAS Domain-Specific Modeling Language (DSML) comprised of the metamodel of MIDAS semantic architecture and the Unified Modeling Language (UML) profile. The metamodel enables the instantiation of platforms independent models, and provides a portability solution for MIDAS to be develop and run on distinct cloud providers without creating a lock-in situation. Through the metamodel of MIDAS, we can document our models, reuse, debug and perform our artifacts without ending implementation. Models are key ar-

tifacts to keep the development process on track and to implement increasingly reliable software design.

To evaluate the DSML expressiveness, we executed three types of query, and our results show that modeling language has an acceptable coverage level of concepts to define the middleware. In future work, we plan to perform a controlled experiment to analyze the consistency, correctness, completeness, and development time to obtain MIDAS source code from the metamodel of MIDAS architecture.

REFERENCES

Bergmayr, A., Troya Castilla, J., Neubauer, P., Wimmer, M., and Kappel, G. (2014). Uml-based cloud application modeling with libraries, profiles, and templates. In *CloudMDE 2014: 2nd International Workshop on Model-Driven Engineering on and for the Cloud co-located with the 17th International Conference on Model Driven Engineering Languages and Systems (MoDELS 2014)(2014), p 56-65*. CEUR-WS.

Christensen, B. and Ellingsen, G. (2016). Evaluating model-driven development for large-scale ehrs through the openehr approach. *International journal of medical informatics*, 89:43–54.

Cohen, R. (2009). Examining cloud compatibility, portability and interoperability. http://www.elasticvapor.com/2009/02/examining-cloud-compatibility.html. Online; Accessed: 2016-12-13.

Da Silva, A. R. (2015). Model-driven engineering: A survey supported by the unified conceptual model. *Computer Languages, Systems & Structures*, 43:139–155.

Ferry, N., Chauvel, F., Song, H., Rossini, A., Lushpenko, M., and Solberg, A. (2018). Cloudmf: model-driven management of multi-cloud applications. *ACM Transactions on Internet Technology (TOIT)*, 18(2):1–24.

Gonidis, F., Simons, A. J., Paraskakis, I., and Kourtesis, D. (2013). Cloud application portability: an initial view. In *Proceedings of the 6th Balkan Conference in Informatics*, pages 275–282.

Herrera, A. S.-B., Willink, E. D., and Paige, R. F. (2016). A domain specific transformation language to bridge concrete and abstract syntax. In *International Conference on Theory and Practice of Model Transformations*, pages 3–18. Springer.

Lewis, G. A. (2013). Role of standards in cloud-computing interoperability. In *System Sciences (HICSS), 2013 46th Hawaii International Conference on*, pages 1652–1661. IEEE.

Maciel, R. S. P., David, J. M. N., Claro, D. B., and Braga, R. (2016). Full interoperability: Challenges and opportunities for future information systems. *I GranDSI-BR*, page 107.

Magalhães, A. P., Maciel, R. S. P., and Andrade, A. M. S. (2015). Towards a metamodel design methodology: Experiences from a model transformation metamodel design. In *SEKE*, pages 625–630.

Mane, B., Rocha, W. d. S., Lima, E., and Claro, D. B. (2020a). Semantic similarity attributes for data cloud: A case study in midas. In *Proceedings of the Brazilian Symposium on Multimedia and the Web*, pages 89–96.

Mane, B., Rocha, W. d. S., Ribeiro, E. L. F., Jesus, L. E. N. d., Motta, I. C., Lima, E., and Claro, D. B. (2020b). Enhancing semantic interoperability on midas with similar daas parameters. In *XVI Brazilian Symposium on Information Systems*, pages 1–8.

Mellor, S. J. (2004). *MDA distilled: principles of model-driven architecture*. Addison-Wesley Professional.

Merle, P., Barais, O., Parpaillon, J., Plouzeau, N., and Tata, S. (2015). A precise metamodel for open cloud computing interface. In *2015 IEEE 8th International Conference on Cloud Computing*, pages 852–859. IEEE.

Ranabahu, A. and Sheth, A. (2010). Semantics centric solutions for application and data portability in cloud computing. In *Cloud Computing Technology and Science (CloudCom), 2010 IEEE Second International Conference on*, pages 234–241. IEEE.

Sanaei, Z., Abolfazli, S., Gani, A., and Buyya, R. (2013). Heterogeneity in mobile cloud computing: taxonomy and open challenges. *IEEE Communications Surveys & Tutorials*, 16(1):369–392.

Shawish, A. and Salama, M. (2014). Cloud computing: paradigms and technologies. In *Inter-cooperative Collective Intelligence: Techniques and Applications*, pages 39–67. Springer.

Van Solingen, R., Basili, V., Caldiera, G., and Rombach, H. D. (2002). Goal question metric (gqm) approach. *Encyclopedia of software engineering*.

Whittle, J., Hutchinson, J., and Rouncefield, M. (2013). The state of practice in model-driven engineering. *IEEE software*, 31(3):79–85.

Zalila, F., Challita, S., and Merle, P. (2017). A model-driven tool chain for occi. In *OTM Confederated International Conferences" On the Move to Meaningful Internet Systems"*, pages 389–409. Springer.

Zalila, F., Challita, S., and Merle, P. (2019). Model-driven cloud resource management with occiware. *Future Generation Computer Systems*, 99:260–277.

Zhang, Z., Wu, C., and Cheung, D. W. (2013). A survey on cloud interoperability: taxonomies, standards, and practice. *ACM SIGMETRICS Performance Evaluation Review*, 40(4):13–22.