Building MDE cloud services with DISTIL

Carlos Carrascal Manzanares, Jesús Sánchez Cuadrado, Juan de Lara
Modelling and Software Engineering Research Group
http://www.miso.es
Computer Science Department
Universidad Autónoma de Madrid (Spain)
Car.Carrascal@estudiante.uam.es, {Jesus.Sanchez.Cuadrado, Juan.deLara}@uam.es

Abstract—**Model-Driven Engineering (MDE) techniques, like transformations, queries, and code generators, were devised for local, single-CPU architectures. However, the increasing complexity of the systems to be built and their high demands in terms of computation, memory and storage, requires more scalable and flexible MDE techniques, likely using services and the cloud. Nonetheless, the cost of developing MDE solutions on the cloud is high without proper automation mechanisms.**

In order to alleviate this situation, we present DISTIL, a domain-specific language to describe MDE services, which is able to generate (NoSQL-based) respositories for the artefacts of interest, and skeletons for (single or composite) services, ready to be deployed in Heroku. We illustrate the approach through the construction of a repository and a set of cloud-based services for bentō reusable transformation components.

Index Terms—**Model-Driven Engineering, Domain-Specific Languages, Service-Oriented Programming, REST services, Cloud Computing, Code Generation**

I. INTRODUCTION

The development of a Model-Driven Engineering (MDE) solution involves dealing with artifacts of different kinds and their relationships. The most common approach is to store artifacts using local folders and projects, for instance, using the infrastructure of the Eclipse plataform. In this approach, computations also occur in the local machine. This has several shortcomings, such as limited reuse opportunities, scalability problems and reduced flexibility.

Instead, advanced solutions, aiming to tackle the challenges of scalability and flexibility raised by today's complex systems [14], require scalable database storage, with retrieval strategies of different nature, ranging from simple, tag-based searches to complex queries [21], and the scalable execution of model management operations such as model transformations, or code generation.

In this setting, cloud computing is an appealing approach to build advanced solutions, with scalability up-front [8]. While there are many service providers, the entry level to develop a cloud-based service is high, specially for research prototypes. Thus, proposals to facilitate the development of cloud-based MDE tools are needed to increase its adoption, both within the MDE community but notably by other software engineering communities.

In this work we present our first results towards addressing this issue. Our approach is based on a domain-specific language intended to facilitate the specification of MDE services. The language, named DISTIL (for MDE service SpecificaTIon

Language), permits the specification of the structure of the repository for storing the MDE artifacts, the basic services required for them (like upload, download or search), as well as more advanced user-defined services, their (parallel or sequential) composition, and their triggering conditions.

DISTIL is accompanied with a set of code generators able to synthesize the repository structure and persitence services, to generate support for basic (REST) services for the defined artifacts and skeletons for the user-defined services, and an HTML web client interface for them. DISTIL uses by default MongoDB [5] for the persistence, and the Spark Java framework[1] as the support for the REST services. The generated elements are included in a project ready to be deployed in the Heroku cloud platform [11]. This way, using DISTIL, the developer is freed from the technical details of the many technologies involved, and typically he only needs to provide the functionality (in Java) for the user-defined services. For this purpose, the DISTIL development environment (an Eclipse plugin) has a smooth integration with the Java IDE. We illustrate the approach by defining a set of cloud services for reusable model transformation components developed with bentō [17].

The rest of the paper is organized as follows. Section II introduces a running example that will be used in Section III to explain the main concepts of the DISTIL language. Section IV describes its tool support, Section V compares with related research, and Section VI finishes with the conclusions and plans for future work.

II. RUNNING EXAMPLE

We first start by proposing a motivating scenario, which we then solve using DISTIL.

Assume we would like to build a repository for reusable, generic transformation components built with bentō [17]. In bentō, a basic reusable transformation component includes one or more *concepts* and a transformation template (a regular ATL transformation). A concept is similar to a meta-model, but it describes the minimal structural requirements that a concrete meta-model must fulfil for the reusable transformation to become applicable to it. When the component is used, the concept needs to be bound to a concrete meta-model, and then
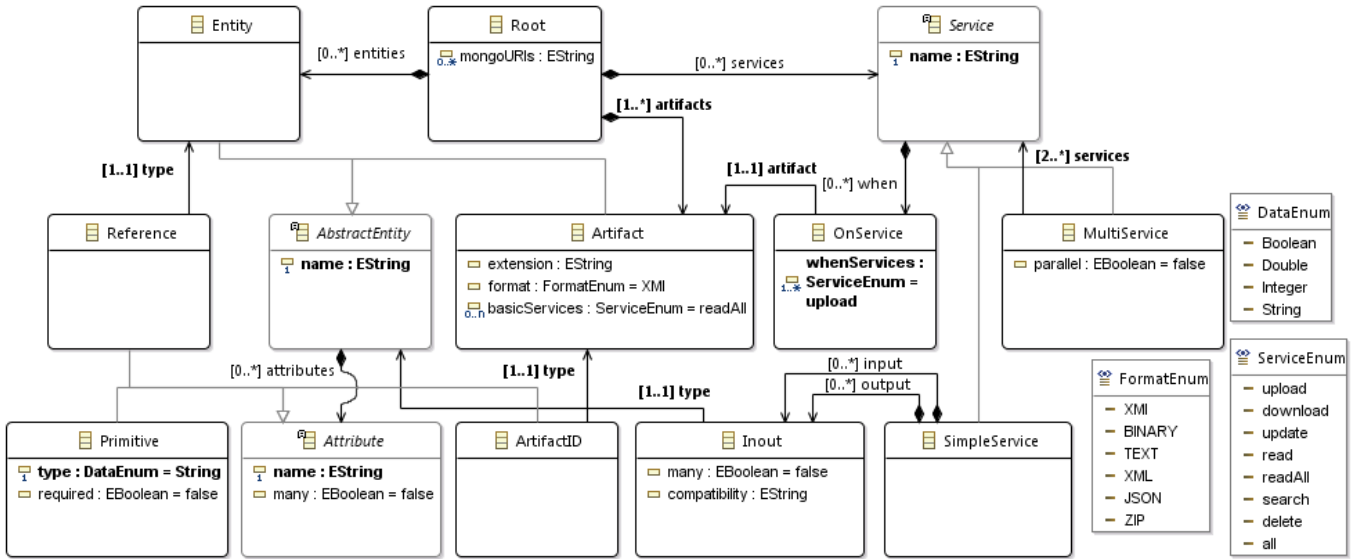
---

[1]http://sparkjava.com/

Fig. 1: DISTIL meta-model excerpt.

the bentō engine rewrites the transformation template, so that it gets adapted for the particular meta-model.

In practice, a component developer is interested in uploading components, and checking their correctness, for example performing a static type-checking of the transformation [18] or obtaining some metrics about the transformation and the found errors. A reuser might be interested in retrieving components using two main search strategies: (1) finding components via tag search, and (2) retrieving components whose concept(s) matches a given concrete meta-model. This process could be integrated in the IDE or accessible via a web interface. Thus, the repository should be accessible via a REST API to enable different user interfaces.

In previous works [19] we described the packaging mechanism of bentō components. It includes a *manifest* file describing the artefacts of the component, one or more .ecore files implementing the component concepts and an ATL transformation template. These elements are packaged into a *zip* file to ease distribution.

III. DISTIL: AN MDE SERVICE SPECIFICATION LANGUAGE

The DISTIL language permits describing *both* the Artifacts and the Services of interest. An excerpt of its meta-model is shown in Fig. 1.

By default, DISTIL uses MongoDB as the underlying database system, and hence the Root class contains the URI of the database (but providing a value is not mandatory). A DISTIL specification is made of two parts. On the one hand, a description of the structure of the artifacts of interest (using classes Artifact and Entity) and their associated basic services. On the other hand, a declaration of the inputs and outputs of extra user-defined services (subclasses of Service).

An Artifact is an element to be stored in the repository. Its definition triggers the creation of the corresponding database structure, and hence a MongoDB *collection* is created for every artifact. An Artifact is described by an extension, a format, a set of basic services to be generated, and a set of attributes. Attributes may be of primitive type (class Primitive), references to Entities (class Reference) or to other Artifacts (class ArtifactID). In MongoDB, each stored artifact (each *document*) receives a String identifier. Therefore, while in the DISTIL specification ArtifactID models a reference to another artifact, in the generated code, the navigation is done through such string identifiers. Entities are similar to Artifacts, but are used to factor out commonalities of Artifacts, and hence they do not generate a separate collection in the database. The required attribute in Primitive attributes is used to indicate whether such attribute will be required in the *request*s to the generated upload and update services.

For each Artifact DISTIL provides complete support for basic services: upload, download, update, read, readAll, search and delete (see the ServiceEnum enumerate type). However, new user-defined services can be described through subclasses of Service. On the one hand, SimpleServices receive Artifacts or Entities as input and output. While services can be called on demand, it is also possible to specify when they should automatically be executed, for example when some basic service is invoked for an Artifact. On the other hand, MultiServices are used to compose (in sequence or parallel) several (composed or simple) services. The language includes validations checking the compatibility of the outputs/inputs of services. For inputs and outputs it is possible to specify required values for some of their attributes. As we will see in next section, DISTIL creates both REST APIs and HTML front-ends for the specified services (both basic and user-defined).

Listing 1 shows the DISTIL specification for the running example. The listing declares an Artifact for storing bentō components in lines 1–8. The Artifact declares in line 2

the extension (.bentoz), format (zip), and requires all basic services. There are several built-in formats, and for some of them automated injectors and extractors can be made available (e.g., for zip we automatically generate packing and unpacking facilities). Then, in lines 4–7 defines the structure of a component, which has the input and output concepts, the ATL transformations it encapsulates, and the tags. We use the id prefix to indicate a reference to another artifact (e.g., in lines 4-7). The primitive attributes (tags in this case) are used to enable the search service. Notably, for string attributes our implementation of this service is able to look for synomyms, as it uses Wordnet [15], a lexical database for the English language.

The listing also declares artefacts for storing the ATL transformation (ATLTrafo, lines 10–15) and the meta-models/concepts (MetaModel, lines 17–22). It also declares an Artifact for models (Model, lines 24–28), because it is used by other user-defined services.

```
1  Artifact Bento [
2    extension .bentoz format ZIP services : all
3  ]{
4    id many inputConcepts : MetaModel
5    id many outputConcepts : MetaModel
6    id many atl : ATLTrafo
7    many tags : String
8  }
9
10 Artifact ATLTrafo [
11   extension .atl format TEXT
12   services : upload, delete, download, search
13 ]{
14   nameATL : String
15 }
16
17 Artifact MetaModel [
18   extension .ecore format XMI
19   services : upload, delete, download, search
20 ]{
21   uri : String
22 }
23
24 Artifact Model [
25   format XMI services : all
26 ]{
27   nameModel : String
28   id type : MetaModel
29 }
30
31 MultiService Analyse {
32   when : Bento [upload, update]
33   services : TypeCheck, Metrics
34 }
35
36 Service TypeCheck {
37   output: Model [with type.uri="http://prob.ecore"]
38 }
39
40 Service Metrics {
41   input: Model [with type.uri="http://prob.ecore"]
42   output: Model [with type.uri="http://mtr.ecore"]
43 }
44
45 Service SemanticSearch {
46   input : MetaModel
47   output : many Bento
48 }
```

Listing 1: DISTIL specification for the running example

The listing declares a multi-service Analyse (lines 31–34), which runs in sequence the simple services TypeCheck and Metrics when a bentō component is uploaded or updated. Both services receive the bentō component as input. TypeCheck (line 37) is a simple service, whose goal is to statically analyse the transformation [18] and outputs a model reporting the problems found. Metrics (lines 40-43) is a service calculating some transformation and error metrics, which also takes as input the output of the TypeCheck service.

Please note that we declare constraints on the properties of the allowed input and outputs of these services. This way, we declare that the output artifact of TypeCheck has type.uri equal to "http://prob.ecore". In this case, this indicates the meta-model the model is an instance of. Finally, SemanticSearch is a service that does not define a trigger, and hence has to be invoked on demand. Its input is a meta-model and its output is a collection of bentō components whose concepts are structurally close to such meta-model. As we will see in next section, the DISTIL description of user-defined services enables the generation of supporting Java code, which the designer needs to fill with the appropriate behaviour (calculation of metrics, type checking, etc).

IV. ARCHITECTURE AND TOOL SUPPORT

A. Architecture

We have built a code generator that, given a DISTIL specification, synthesizes the needed repository structure, the full implementation of the basic services, the skeleton of user-defined services, and prepares the generated code in a project ready to be deployed on Heroku. The latter includes the needed dependencies, a *pom.xml* file for compiling the code using Maven (https://maven.apache.org/), and some configuration files for the execution in Heroku. The generated code uses the Spark framework as the support for the generated REST services, and MongoDB for the storage. However, the generated code facilitates changing the database engine.
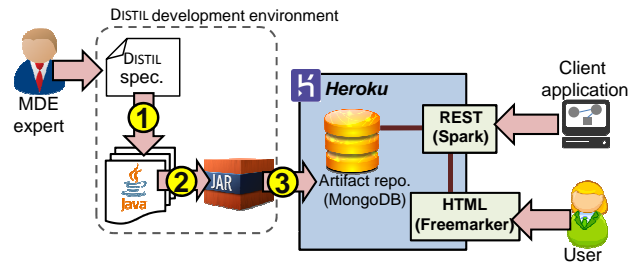

Fig. 2: Architecture of the solution and process for creating cloud-based MDE services with DISTIL

Fig. 2 depicts the process to create cloud-based MDE services with DISTIL, as well as the solution general architecture. In a first step, the MDE expert interacts with the DISTIL IDE. As we have seen, he should build a DISTIL specification. The IDE generates Java code (label 2), and the engineer has typically to fill some functionality holes for the user-defined services. The details of the IDE and its ability to integrate

DISTIL programs with Java will be explained in the next subsection. Then, the IDE (label 3) can package the different artifacts to be uploaded into Heroku. By default, the generated services contain the REST services for the basic and user defined services, an HTML frontend (for which Freemarker is used) and a MongoDB database. This way, the generated services can be used by humans, or be integrated into client applications. The REST services manage the different artifacts using the JSON format.

The generated code from a DISTIL specification extends a manually created service kernel. The scheme of the generated code is shown in Fig. 3. For each Artifact (e.g., ATLTrafo), the generator synthesizes a Java package (atltrafoServices), which contain both necessary classes to create the REST API (package atltrafoServices.basic and class ATLTrafoJson), and an automatically generated HTML web client (package atl-trafoServices.htmlCover and class CustomATLTrafoHtml), and for the persistency (class ATLTrafo.java). Then, one class for each user-defined service is also generated (in package service, not shown in the figure). This generation scheme facilitates the extension of the generated code with new features. For example, it is easy to add features to the HTML client by modifying the methods in class CustomATLTrafoHML. To give an idea of the size of our framework, the kernel has over 1200 Lines of Code (LOC) in 32 Java classes. The automatically generated Java code for the example amounts to 2100 LOC in 47 Java classes, and the code generator has over 3100 LOC in 22 *xtend* files.
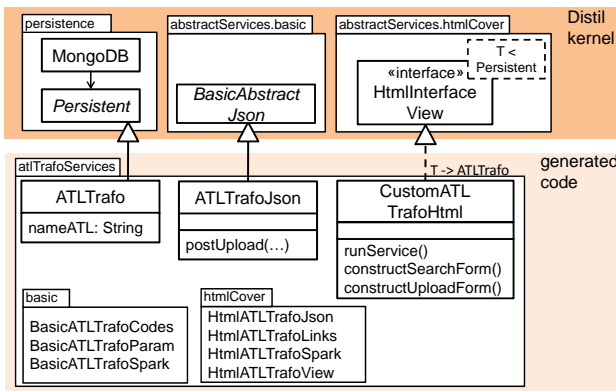

Fig. 3: Structure of the generated code

Typically, given a DISTIL program, the engineer would need to fill some functionality "holes" in the generated classes. Next subsection describes our tool support to make this process smooth.

B. The DISTIL *development environment*

We have built an Eclipse plugin aiming at providing a tight and seamless integration of DISTIL with the generated Java code. The tool is freely available at http://miso.es/tools/distil.html.

One of the tool design goals is to facilitate the completion of the generated code for the user-defined services using Java. A screenshot of the tool is shown in Fig. 4. The language editor

(label 1 in the figure) has been developed using Xtext [3], and enables the following features:

1) Hyperlinking the Artifact and Service names with the generated Java classes. The Java classes contain "holes" where the user can provide extra functionality for the services, which are marked as "TO-DOs".
2) The DISTIL editor gives warnings if any of such "TO-DOs" for any declared Artifact or Service remains. One such warning is shown in the Figure (label 2), and we provide a *quickfix* with an hyperlink to the corresponding Java file (label 3).

In the package explorer view of the Figure (label 4), we can see that the DISTIL IDE has created services for every Artifact and Service defined, following the scheme explained in previous section.

Fig. 5 shows how by hovering over the Artifact name, the engineer is offered the possibility to navigate to the Java classes which might need to be extended by hand. The code generator respects the manually written code in the Java files, so that it does not get overwritten.
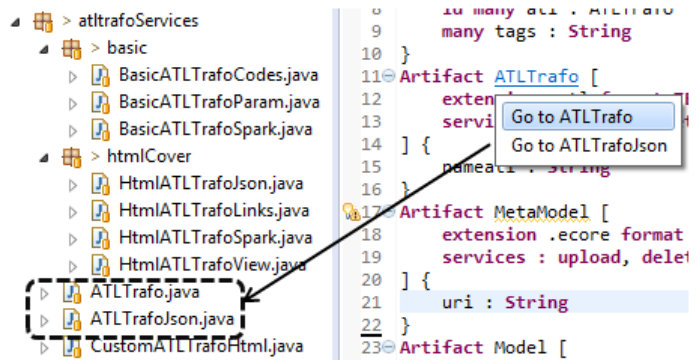

Fig. 5: Hyperlinking DISTIL specifications and generated Java code

Fig. 6 shows the search facility of the generated web interface, running after it has been deployed in Heroku. It can be noted that the interface permits search using synonyms. The running example, is available at https://miso-distil-bento.herokuapp.com/.

V. RELATED WORK

One of the earliest work dealing with the servitization of model operation is ModelBus [20]. This was a tool integration technology which built upon web wervices, following a SOA approach. ModelBus featured a model repository and facilitates the orchestration of modeling services. However, it lacked dedicated languages to describe both the repository structure or the service integration.

With the advent of cloud technology, in [4] the authors introduced the notion of Modeling as a Service (MaaS) as a way to provide MDE services from the cloud, analysing the applications of such idea. Approaches to realize this idea have emerged over the years. Some of them focus on the extensible definition of repositories for MDE artifacts.
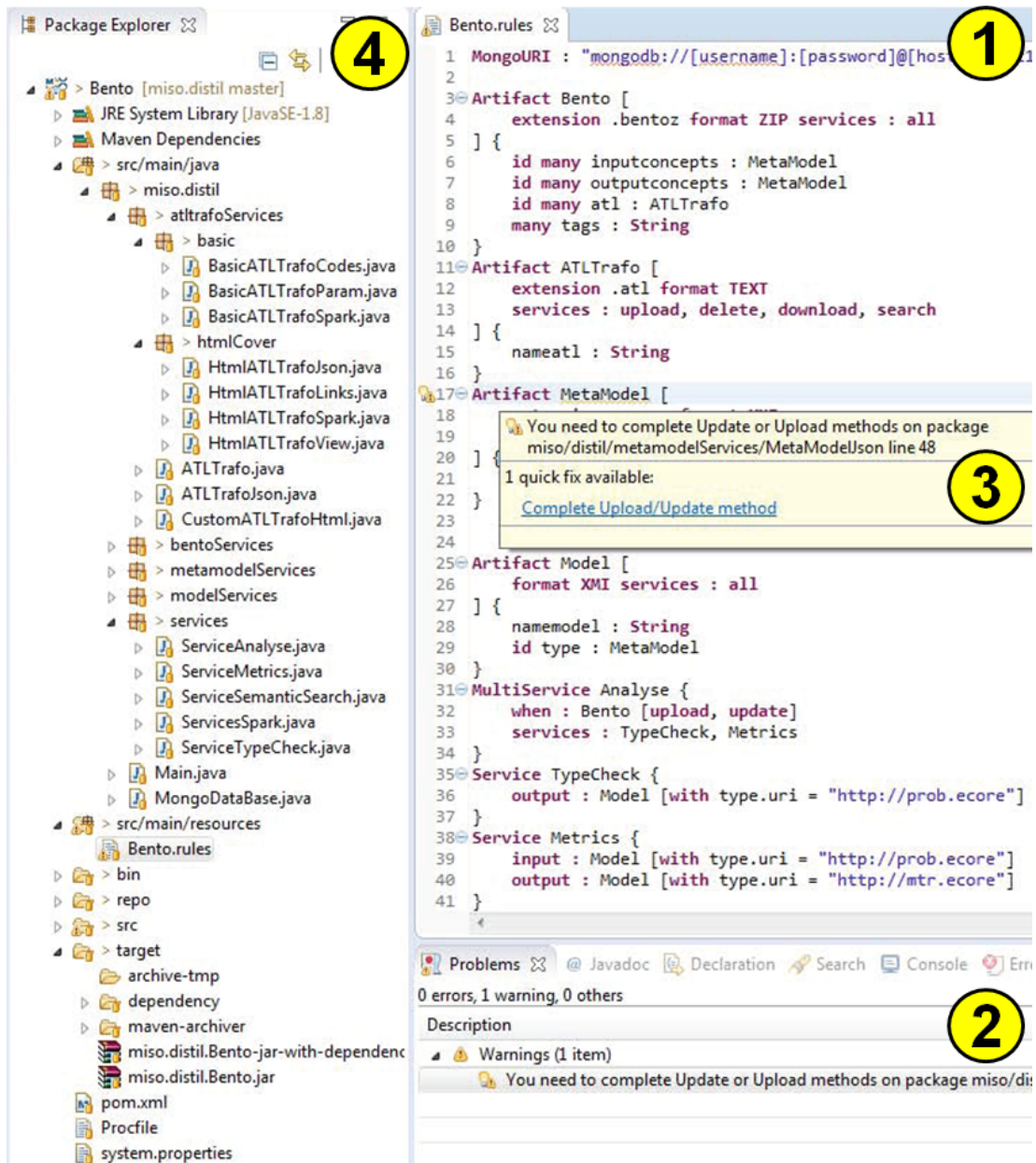
Fig. 4: The DISTIL development environment.

In this category, ReMoDD [10] is based on general-purpose content management systems, like Drupal. On the other hand, others like MORSE [12], have an explicit meta-model to adapt to the specific artifact to be stored. MORSE is able to store models and meta-models and provides version control. Another related approach is MDEForge [2], which is perhaps the closest to our approach (http://mdeforge.org). MDEForge is an extensible, generic respository to store MDE artifacts, and provide services for them. MDEForge is extensible, as it allows adding new kinds of artifacts and services. However, while the previous approaches provide at most a meta-model to customize the storage part, we provide a DSL and rich tool support for both storage and service definition, including code generation and deployment.

Other approaches focus on providing specific MDE services. For example, AToMPM [6] offers a complete cloud-based modelling environment, while Hypersonic [1] is a cloud-based tool to perform model analysis. However, these approaches are specific for a certain task (modelling, analysis), while we provide a DSL to specify MDE cloud services.

Service oriented programming languages, like Jolie [16] provide primitives to define and orchestrate services. Instead, we opted by a high-level DSL, and a code generation approach. However, in the future we aim to extend DISTIL with more sophisticated service orchestration primitives.

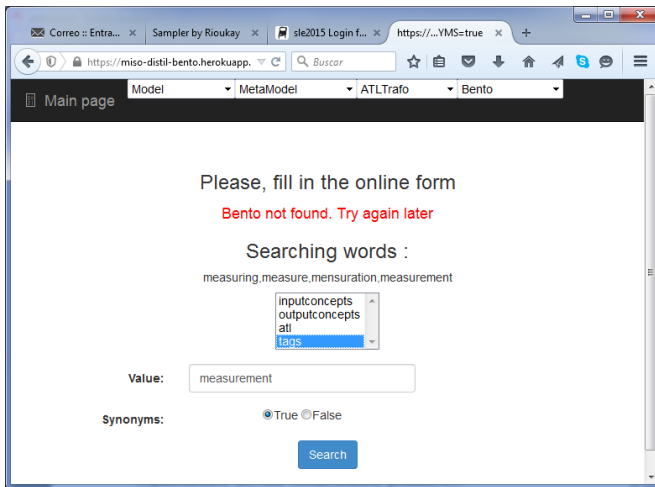In [7], an approach called EMF-REST to expose an EMF

Fig. 6: The generated web interface for the defined Heroku cloud services.

model as a REST API is presented. This allows basic model management operations to be performed remotely. Our notion of artifact is coarser grained, and thus we do not automatically generate access facilities to individual elements. Instead, services generated by EMF-REST could be manually integrated into DISTIL generated code.

Our approach to describe services is somehow similar to Apache Ant (http://ant.apache.org/). In [13], Ant was extended with model management tasks for e.g., model validation, transformation or merging. However, these are not executed in the cloud, but locally.

Hence, altogether, the contribution of this work is a DSL to describe both the structure of the MDE repository and the associated services, supported by a generative framework.

VI. CONCLUSIONS AND FUTURE WORK

This paper has presented DISTIL, a domain-specific language for specifying MDE services, to be deployed in the cloud. The language permits generating No-SQL based persitence services for the artifacts of interest, as well as skeletons for user-defined services. The tool is tightly integrated with the Java IDE, to facilitate the addition of functionality to the service skeletons, and generates ready-to-deploy projects for Heroku.

In the future, we would like to provide support for other cloud systems, in addition to Heroku, and enable the definition of policies for scaling, like those of CloudMF [9]. We are also improving tool support, extending DISTIL with more sophisticated support for service definition and composition (like exception handling, compensation actions, transactions) and using it for more advanced case studies.

REFERENCES

[1] V. Acretoaie and H. Störrle. Hypersonic - model analysis as a service. In *Joint Proceedings of MODELS 2014 Poster Session and the ACM Student Research Competition*, volume 1258 of *CEUR Workshop Proceedings*, pages 1–5. CEUR-WS.org, 2014.

[2] F. Basciani, J. D. Rocco, D. D. Ruscio, A. D. Salle, L. Iovino, and A. Pierantonio. MDEForge: an extensible web-based modeling platform. In *CloudMDE, satellite event of MoDELS*, volume 1242 of *CEUR Workshop Proceedings*, pages 66–75. CEUR-WS.org, 2014.

[3] L. Bettini. *Implementing Domain-Specific Languages with Xtext and Xtend*. Packt Publishing, 2013. See also https://eclipse.org/Xtext/.

[4] H. Brunelière, J. Cabot, and F. Jouault. Combining Model-Driven Engineering and Cloud Computing. In *MDA4ServiceCloud, satellite event of ECMFA*, June 2010.

[5] K. Chodorow. *MongoDB: The Definitive Guide*. O'Reilly Media, 2013. See also https://www.mongodb.org/.

[6] J. Corley and E. Syriani. A cloud architecture for an extensible multi-paradigm modeling environment. In *Joint Proceedings of MODELS 2014 Poster Session and the ACM Student Research Competition*, volume 1258 of *CEUR Workshop Proceedings*, pages 6–10, 2014.

[7] H. Ed-Douibi, J. L. C. Izquierdo, A. Gómez, M. Tisi, and J. Cabot. EMF-REST: Generation of RESTful APIs from Models. *arXiv preprint arXiv:1504.03498*, 2015.

[8] T. Erl, R. Puttini, and Z. Mahmood. *Cloud Computing: Concepts, Technology & Architecture*. Prentice Hall, 2013.

[9] N. Ferry, H. Song, A. Rossini, F. Chauvel, and A. Solberg. CloudMF: Applying MDE to Tame the Complexity of Managing Multi-Cloud Applications. In R. Bilof, editor, *UCC 2014: 7th IEEE/ACM International Conference on Utility and Cloud Computing*, pages 269–277. IEEE Comp. Soc., 2014.

[10] R. France, J. Bieman, and B. H. C. Cheng. Repository for model driven development (ReMoDD). In *Proceedings of the 2006 International Conference on Models in Software Engineering*, MoDELS'06, pages 311–317, Berlin, Heidelberg, 2006. Springer-Verlag.

[11] A. Hanjura. *Heroku Cloud Application Development*. Packt Publishing, 2014. See also http://heroku.com.

[12] T. Holmes, U. Zdun, and S. Dustdar. Automating the management and versioning of service models at runtime to support service monitoring. In *16th IEEE International Enterprise Distributed Object Computing Conference, EDOC 2012*, pages 211–218. IEEE Computer Society, 2012.

[13] D. Kolovos, R. Paige, and F. Polack. A framework for composing modular and interoperable model management tasks. In *In Model-Driven Tool and Process Integration Workshop*, pages 79–90, 2008.

[14] D. S. Kolovos, L. M. Rose, N. Matragkas, R. F. Paige, E. Guerra, J. S. Cuadrado, J. De Lara, I. Ráth, D. Varró, M. Tisi, and J. Cabot. A research roadmap towards achieving scalability in model driven engineering. In *BigMDE, satellite event of STAF*, pages 2:1–2:10, New York, NY, USA, 2013. ACM.

[15] G. A. Miller. Wordnet: A lexical database for english. *Communications of the ACM*, 38(11):39–41, 1995.

[16] F. Montesi, C. Guidi, and G. Zavattaro. Service-oriented programming with jolie. In *Web Services Foundations*, pages 81–107. Springer, 2014.

[17] J. Sánchez Cuadrado, E. Guerra, and J. de Lara. A component model for model transformations. *IEEE Trans. on Soft. Eng.*, 40(11):1042–1060, 2014.

[18] J. Sánchez Cuadrado, E. Guerra, and J. de Lara. Uncovering errors in ATL model transformations using static analysis and constraint solving. In *25th IEEE International Symposium on Software Reliability Engineering, ISSRE 2014*, pages 34–44. IEEE, 2014.

[19] J. Sánchez Cuadrado, E. Guerra, and J. de Lara. Reusable model transformation components with bentō. In *Proceedings of ICMT'15*, LNCS, 2015. Springer.

[20] P. Sriplakich, X. Blanc, and M. Gervais. Collaborative software engineering on large-scale models: requirements and experience in modelbus. In *Proceedings of the 2008 ACM Symposium on Applied Computing (SAC)*, pages 674–681. ACM, 2008.

[21] G. Szárnyas, B. Izsó, I. Ráth, D. Harmath, G. Bergmann, and D. Varró. Incquery-d: A distributed incremental model query framework in the cloud. In *MODELS*, volume 8767 of *LNCS*, pages 653–669. Springer, 2014.