Engineering applications over Social and Open Data with Domain-Specific Languages

Ángel Mora Segura* and Juan de Lara

Modelling and Software Engineering Group
Universidad Autónoma de Madrid (Spain)
{Angel.MoraS, Juan.deLara}@uam.es

Abstract. There is a current trend among governments and organizations to make all sort of information (like budgets, demographic or economic data) public. The information released in this way is called *Open Data*. Many institutions promote the creation of innovative applications using the data they have released, e.g., in combination with social networks, but only highly skilled engineers can accomplish this task.
Our goal is to facilitate the construction of applications using open data and social networks as communication platform. For this purpose, we propose a family of domain-specific languages directed to automate the different tasks involved, like describing the structure and semantics of the heterogeneous data sets, the patterns to be sought in social network messages, the information to be extracted from static and dynamic data and the messages (over social networks) that the system needs to produce. We have built an extensible working prototype, which allows adding new open data formats and support for different social networks.

Keywords: Open data, Social networks, MDE, DSLs

1 Introduction

There is currently a trend by different kinds of entities (governmental, foundations and companies) to offer in an open way all sorts of information, like economic, demographic, legal, scientific or service data. Typically, such data are offered as *open data*, which can be freely used and distributed [10]. However, it is not enough to deploy it, but it needs to have a format and description useful for consumers, as well as some degree of interconnection amongst the different resources in which it is presented. Unfortunatelly, in practice, open data are frequently published "as is" (raw data), and with heterogeneous formats like CSV, Excel, PDF, or accessible via API queries. This heterogeneity makes difficult the systematic engineering of applications making use of them.

In the governmental context, the open data movement is directed to offer higher levels of transparency to citizens, and it is common for institutions to

* Work supported by the Ministry of Education of Spain (FPU grant FPU13/02698), the Spanish MINECO (TIN2014-52129-R) and the R&D programme of the Madrid Region (S2013/ICE-3006).

promote the creation of innovative applications that use their data (e.g., through hackatons and contests). Sadly enough, despite having been marked as highly desirable[1], the active involvement of citizens in such activities (by providing data themselves, or creating customized applications) is still scarce. One reason is that only highly qualified technical personnel is able to create and supply valuable open data sources and applications, since this involves highly complex tasks. These include the identification and monitoring of data sources, their cleansing and semantic tagging, their publication and incorporation to public repositories, the integration of data sources, their interlinking and visualization, and the implementation of an application that uses them in an effective way. Our final goal is to lower the barrier to the creation of this kind of applications.

Our proposal to tackle this problem is based on Model-Driven Engineering (MDE) [15]. In particular, we propose a Domain-Specific Language (DSL), named DataDSL, to describe the syntax and semantics of open data. It permits describing heterogeneous data (coming from different sources, formats and domains) in a unified way. The language has facilities to discover and make explicit the underlying schemas of data sets and link such schemas to different domains.

We show the integration of DataDSL within *EagleData*[2], an MDE framework for the high-level description and automated synthesis of applications over social networks [13]. Hence, DataDSL provides open data resources as one more input among the ones *EagleData* supports. *EagleData* comprises two further DSLs: PatternDSL for describing patterns to be sought on messages in social networks (like Twitter), and RuleDSL for describing rules to be triggered upon the reception of messages matching the given patterns. We demonstrate the approach with an example consisting in a service to show a user-requested bus line route in a map. The interaction with the system is performed via Twitter, and the data set is retrieved from the Spanish open data repository[3].

Paper organization. Sec. 2 motivates our approach, describing current challenges. Sec. 3 overviews our approach and introduces a running example. Sec. 4 presents DataDSL. Sec. 5 shows its integration with *EagleData*. Sec. 6 describes tool support. Sec. 7 analyses related work and Sec. 8 concludes.

2 Motivation

Over the last years, web data has become more organized and detailed, i.e., more semantic, and it is common nowadays to find technologies that aim to gather, organize, represent and query web information. Moreover, the data-based web is no longer static, but fed by a non-stop streaming of information coming from a vast load of files, databases, sensor devices and social media. Specifically, the Semantic Web[4] provides an infrastructure for defining, integrating, sharing and reusing web data according to its meaning and significance. Together with the

[1] `https://www.w3.org/2012/06/pmod/report`
[2] `http://miso.es/tools/EagleData.html`
[3] `http://datos.gob.es`
[4] `https://www.w3.org/standards/`

Semantic Web, a series of technologies have emerged – like RDF, SPARQL or OWL – enabling applications to query the data or use common vocabularies [4]. However, even though standardization has arrived to Semantic Web via *Linked Data* [2], most data made available to citizens do not follow these guidelines, being most frequently presented in an unstructured way or under private formats.

As an example, the Lod Cloud project[5] holds a series of data sets published by contributors, organisations and others. It provides a trustful measurement of how accessible are these sets, using a rating system known as the *5-star* model[6]. This defines a 5-step incremental path to publish data from *available online* (1 star), *structured* (2 stars), with *non-proprietary formats* (3 stars), with *links* (4 stars) and based on *Linked Data* technologies (5 stars).

As Fig. 1 shows, many of the published data sets do not obtain more than 3 stars, meaning that the majority uses non-proprietary formats. Hence, it might be stated that the Semantic Web orthodoxy is still not sufficiently established, probably because companies and distributors find it easier to publish their data in common formats like CSV or JSON, rather than the ones that Linked Data suggests (namely RDF).

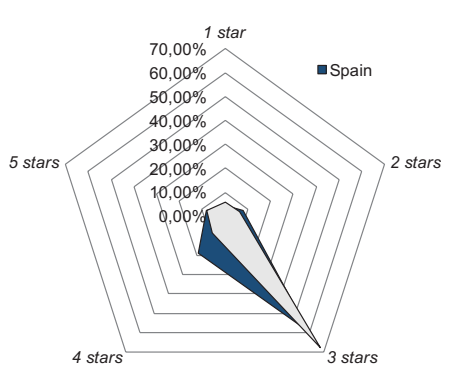

Fig. 1: Status of Lod data sets.

An analysis of the Spanish open data portal evidences the wide variety of formats used, revealing up to 55 different ones, like CSV (16%), HTML (10,6%), JSON (10,4%), XML (9,7%), ASCII (9,3%), XLS (6,8%), and PDF (6,8%). These formats often lack a semantic description of their content, which is one of the main challenges of this work. Hence, in practice, *Linked Data* formats are not the first choice for open data, probably because most of these data sources are aimed to be read both by developers and non-expert users, and, despite being a very complete standard, human readability is not amongst *Linked Data* features.

Therefore, since expecting a scenario in which public administration achieves acceptable levels in the 5-star model is highly idealistic, it is necessary to have a sound way of giving semantics to open data sets, independently of the chosen format for their representation.

Our aim is not only to make sense of open data, but also enabling their active use in social applications. For this purpose, we will integrate open data descriptions with *EagleData*. This is a tool for building simple applications over social networks, like Twitter. This way, the content of open data sources will be accessible through social networks.

[5] `http://lod-cloud.net/`
[6] `http://5stardata.info/`

3 Overview and Running Example

Our challenge is to build a framework that lightens the developement of applications combining open data, social network data, and faciliating human interaction with them. Fig. 2 shows the typical process, whose steps are automated by *EagleData* or by the social network of choice (*Twitter* in this case).

The process starts with a certain organism releasing some open data. The developer is to become familiar with these data, performing data *cleansing* if needed [11] (label 1). Ideally, an explicit description of the data structe and its semantics would be produced. Our DataDSL language permits describing such structure and the tooling helps discovering the underlying data schema and linking it with existing data descriptions (see Secs. 4.1 and 4.2). Then, a platform-specific data structure has to be created to store the items (label 2), and the open data sets are parsed and instances of those structures created. If data from several sources are handled, they need to be integrated. Our tooling automates this process as well (see Sec. 4.3).

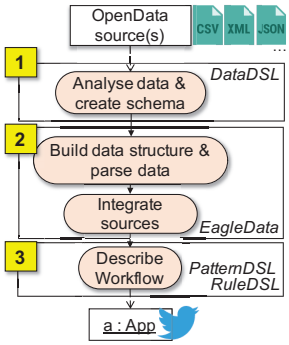

Fig. 2: Creating social apps over open data.

Then, an app using these data is created (label 3). *EaglaData* relies on social networks, and provides two DSLs (see Sec. 5). PatternDSL permits describing patterns to be sought in social network messages. RuleDSL enables the description of actions to be performed upon the reception of social messages, like querying data (e.g., received in previous messages, or in open data sources), and sending messages. The final app uses Twitter as front-end to interact with the open data. Many users are already familiar with social networks, so their use presents and advantage with respect to learning a new *app*.

Running example. We would like to create an application to visualise the route of a given bus line, so that users can see the closest stop to their current position. The city of Santander (Spain) has made available the static data of the stop positions and bus lines at the open data Spanish portal, in CSV format (see an excerpt in Fig. 3). In the file, the bus line names have been abbreviated to *A* and *B*. In the data set, there is a row per stop and bus line, so that if both lines share a stop (e.g., *Camarreal 109*), there are two rows representing that stop: one for line A, and another for line B.

Taking advantage of the widespread use of Twitter, and that it especially favours interaction via mobile devices, our application expects a user tweet or

```
56; Camarreal 109;          A; Calle; Camarreal; 109; 43.44625; -3.87072;
61; Campogiro 23;           A; Calle; Campogiro; 23;  43.45044; -3.85070;
63; Campogiro 5;            A; Calle; Campogiro; 5;   43.45017; -3.85127;
64; Cajo 17;                B; Calle; Cajo;      17;  43.45405; -3.83708;
80; Parque Doctor Morales;  A; Calle; Cajo;      1;   43.45496; -3.83419;
88; Camarreal 109;          B; Calle; Camarreal; 109; 43.44625; -3.87072;
```

Fig. 3: Excerpt from an Open Data file in CSV format

direct message indicating the line to visualize, search the bus line number in the data set, and build a Google Maps URL that opens a map with the complete bus line route. The resulting URL is sent back to the user as a direct message.

4 DataDSL: a DSL to describe heterogeneous open data

As we have seen, the main issue with open data is the heterogeneity of formats and the frequent lack of explicit semantics. Hence, in order to treat all these data uniformly, a means to describe their underlying schema is needed. Ideally, these means should allow processing the sources automatically, so that such data can be queried and combined, also with data coming from dynamic sources, like social networks. The solution needs to be extensible, because we would like developers to be able to increase it with support for other new formats.

Several established languages exist in different technological spaces to describe the structure of data. For example, in MDE, one resorts to meta-modelling languages like MOF or Ecore [14]; in grammarware, EBNF grammars are used; while in ontology engineers use languages like OWL [4]. However, none of these languages offer native support for the following requirements: (i) being format independent, but allow to add format dependent options describing the pecularities of certain formats (e.g., the separator or the indication of a header in CSV files), (ii) extensibility mechanisms to add new formats and a set of options for these. While one could somehow encode this information in the mentioned languages (e.g., as annotations in Ecore meta-models), it is more convenient to develop a DSL offering this support in a native way.

4.1 Language syntax and semantics

DataDSL is a textual DSL that allows describing the semantics of data in heterogeneous formats. On the one hand it allows building format-independent, reusable descriptions called *fragments*. On the other, it supports format-specific descriptions for concrete data sources, called *data descriptions*. These may reuse fragments, and are typically enriched with options of the particular data format.

Listing 1 shows a DataDSL model with some fragments. Fragments are organized in packages (lines 1–21 and 22–40), which can be annotated (lines 1, 22 and 23) with their application domain. A fragment contains fields, which may declare a cardinality interval, and whose type may be primitive (e.g., source in fragment Line in line 6), an enumeration or another fragment. Fields can be declared key (e.g., id in line 5) to convey that the value is unique. Key fields are used in formats with no native support for references, to uniquely identify the referenced elements. For example, if fragments Line and Stop are reused to describe the content of a CSV file, then the stops reference (line 10) would be serialized as a list of stopIDs. Fragments can be reused to build other fragments, through an extension mechanism similar to inheritance, where a fragment can extend zero or more fragments.

DataDSL supports a large number of primitive data types, like Boolean, Lat (for latitude), Long (for longitude) and String. When using data types, especially within data descriptions, it is possible to indicate the specific serialization of their allowed values. In some specific file, a Boolean may be serialized as true and false, while in others as t and f, or 1 and 0. Hence, fields with primitive type may declare lists of options in parenthesis. Lines 18 and 19 declare the minimum and maximum value for the latitude and longitude. To allow reuse, a refined data type (i.e., a data type with options) can be declared explicitly and given a name. For example, line 3 declares Id as a refinement of Int, where the allowed values have 3 digits as a maximum. As a special case, Strings can be refined by means of options, but also patterns can be specified. For example, PostalCode in lines 26–29 is a refinement of String made of two parts. The first (the city) is made of two digits, and the second (the district) has three.

```
1  @transport
2  package BusLine{
3    datatype Id : Int (minLen = 1, maxLen = 3)
4    fragment Line{
5      key Id id
6      String source
7      String destination
8      GeoPoint sourcePoint (null = "")
9      GeoPoint destinationPoint (null = "")
10     Stop[*] stops
11   }
12   fragment Stop{
13     key Int stopID
14     String name
15     GeoPoint stopPoint
16   }
17   fragment GeoPoint{
18     Lat latit (min = −90.0, max = 90.0)
19     Long longit (min = −180.0, max = 180.0)
20   }
21 }
22 @city
23 @geo
24 package Address{
25   enumeration AddrKind{"Calle","Avenida","Paseo"}
26   datatype PostalCode : String {
27     Digits[2] city
28     Digits[3] district
29   }
30   fragment Address{
31     AddressPoint addressPoint
32     String others
33     PostalCode postalcode
34   }
35   fragment AddressPoint{
36     AddrKind kind
37     String name
38     Int number
39   }
40 }
```

Listing 1: Some DataDSL fragments for the running example.

The purpose of fragments is to describe knowledge of a domain in a reusable, format-independent way. Instead, data descriptions describe a particular data source, in a particular format. As an example, Listing 2 shows a description for the example CSV of Fig. 3. Such description reuses different fragments and definitions in Listing 1.

```
1  import BusLine.*
2  import Address.*
3  description "CSV" SantanderCityBus{
4    Id lineId
5    AddressPoint address
6    Stop stop
7  } (
8    language={"es−ES"},
9    separator={","},
10   order={stop.stopID, stop.name, lineId, address.addressPoint, stop.stopPoint}
11 )
```

Listing 2: Describing the CSV in Fig. 3 reusing fragments.

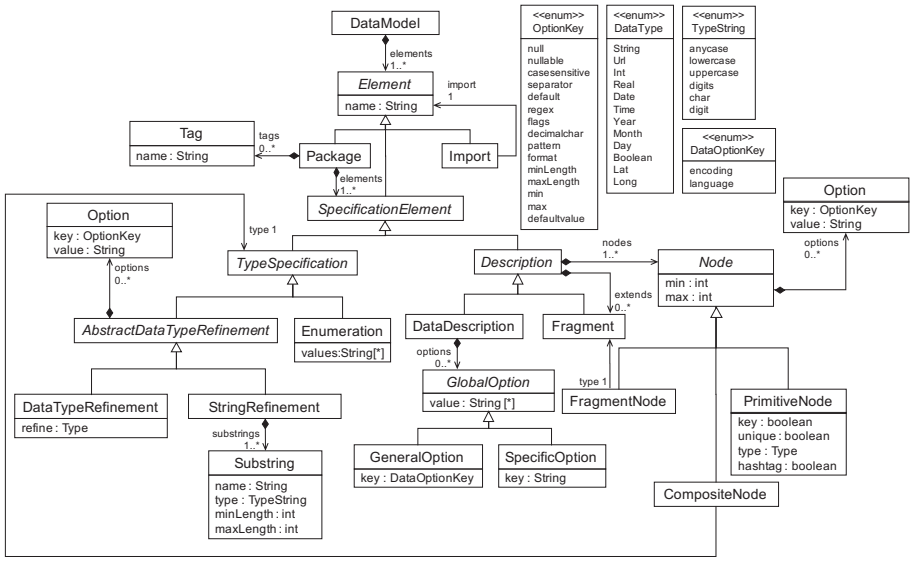

Fig. 4: Simplified meta-model excerpt of DataDSL.

A data description declares its format, which is CSV in the listing (line 3). DataDSL is an *extensible DSL*, where different format handlers (e.g., for CSV, JSON, XML) can be included. As we will see later, each handler is in charge of parsing the data sources of the given format into a common representation, and to provide support for a number of configuration options. Just like fields, data descriptions may include options, specified in parenthesis after their definition (lines 8–10 in the listing). Options for a description may be general (applicable to any format and provided by default by DataDSL) or specific for the given format. The listing declares that the data within the CSV file is in Spanish (line 8) using the general option language. In addition, the CSV format handler contributes with three options: separator (the separator character of fields in the file, ";" in the example), header (whether there is a header row, false by default), and order (to indicate the specific order in which the components of the description will appear in the file). None of the options is mandatory. In particular, if no order is specified, the order in which the fields are listed is taken. If order is specified, and some field of the description is not listed, it is ignored.

An excerpt of DataDSL's meta-model is shown in Fig. 4. It can be observed that both Frament**s** and DataDescription**s** are defined through Node**s** (i.e., fields). These can be of primitive type (PrimitiveNode), fragment type (FragmentNode) or composite. In the latter case, they may have an enumeration or refined type. Node**s** may have Option**s**, some of which are shown in the OptionKey enum. DataDescription**s** can have either general or specific options. The former are predefined in DataDSL, while interpreting the later is done through the specific format handler. Finally, the language distinguishes between DataTypeRefinement**s** and StringRefinement**s**, because the later can specify a structure for the string (class Substring).

4.2 Inferring semantics from data sets

Because we intend to minimize the developer's workload, each format handler can contribute an algorithm for inferring the structure of a particular data set, and make it explicit as a DataDSL model. Then, such inferred schema can be refined by the user, or refactored for the reuse of available fragments. It is to notice that both XML and JSON permit the availability of external descriptions of the data, like *DTD (Document Type Definition)* or *JSON Schema*[7]. With them, it is easy to convert every element into its proper representation in DataDSL. Unfortunately, neither XML nor JSON force the use of these schemas, and hence it is common to find data sets with no description of their structure. In that case, the process is left to heuristics and user intervention, just like with CSV. Generically, the schema inferring algorithms work in four phases:

1. **Check file readability.** In a first step, conformance to the given format is checked. Thus, we approach an early pre-processing of the input file by removing unexpected characters and literals and by warning the user if the open data source does not meet the format's serialization schema.
2. **Data source structure detection.** Having a valid file to process, we create the structure of the DataDescription to generate. By analysing its components, we infer a correspondence to an instance of our DataDSL meta-model. In this step, we merely identify each component, without assigning them a Type.
3. **Data type inference.** By default, we set primitive types as String if they are not explicitly specified (as it happens with DTD). Then, we look over each primitive value from the input file (e.g., we check CSV column values) for finding out whether we can assign the corresponding Node a more specific primitive type. If, for instance, we discover a column that is entirely filled by integers, then we can set that node type to Int. Additional heuristics to refine the primitive type can also be applied. For example, for CSV, we check the column name against predefined sets of keywords, to suggest whether a column is a key, or represent the Latitude and Longitude of some place if two columns are in the same time in the range of a latitude and longitude values. The compatible types are presented in order from more specific (e.g., Lat) to more general (e.g., Real). To give the algorithm some control about mixed values we provide a list of probabilistic rates based on how much values match with the type discovered. Nodes may have options, and so we suggest values for them. For example, for PrimitiveNode we suggest the minimum and maximum values for numbers and the null value for strings (if allowed).

Running example. With *EagleData* we provide a wizard to import files and discover the value types. In the running example the system automatically produces the description in Listing 3 (where field names have been renamed). Enumerations (like the in in line 1) are generated if the number of different strings values is below a certain configurable threshold. For the example, the algorithm suggests

[7] `http://json-schema.org/`

two enums (AddressKind and Line). Once inferred, the user may change the description, and *EagleData* checks if the corresponding source is still conformant to the description.

```
1 enumeration AddressKind {"Calle", "Avenida"}
2 enumeration Line {"A", "B"}
3 description CSV SantanderCityBus{
4   Int id (min = 56, max = 88)
5   String idStop (null = " ")
6   Line line
7   AddressKind addressKind
8   String addressName (null = " ")
9   Int addressNum (min = 1, max = 109)
10  Lat latPoint (minDouble = 43.44625, maxDouble = 43.45496)
11  Long longPoint (minDouble = −3.83419, maxDouble = −3.87072)
12 }("delimiter" = ";")
```

Listing 3: Automatically inferred data source description.

4.3 Handling the heterogeneity of formats

As we will see in Sec. 6, *EagleData* provides a series of (Eclipse) extension points to add support for more formats. In order to enable a uniform access to heterogeneous data, we use a common meta-model, shown in Fig. 5. The idea is to provide mappings for different static data sources to this common model. This meta-model has the structure of a Table that can be used to represent tabular data [6]. Cells of the table can be either ContentCell or TableCell which means the cell contains a subtable instead of a string value. This way, *EagleData* assumes that every data provided by format extensions will consist of a group of Tables.

	CSV	**XML**	**JSON**
Table	Resource	Document	JSONObject
Row	Record	Node	JSONArray
ContentCell.content	Cell	Content	JSONObject.value
TableCell.table	n/a	Node.array	JSONArray.value

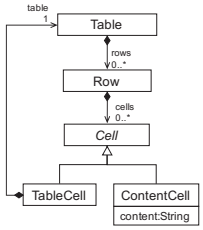

Fig. 5: Mapping data modelling technologies to EagleData's data model.

We provide three basic extensions for the resolution of static data injection: CSV, XML and JSON since they are the top choices when it comes to representing open data. The table shows how those three technologies can be mapped to our common meta-model. Our design is extensible, as it facilitates the integration of new technologies. We will elaborate on the extensibility mechanisms in Section 5.

5 Integration with *EagleData*

Once we have assured that our system is able to work with the structure of a certain open data set, the developer needs to have a means to implement the

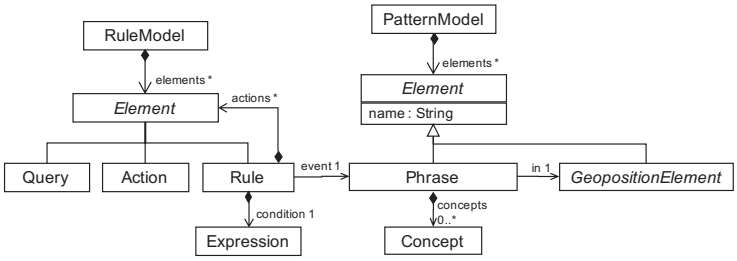

Fig. 6: Simplified meta-model excerpt of PatternDSL and RuleDSL.

actual behaviour of an open data application. Our previous work [13] introduced a framework called *EagleData* for describing simple applications over Twitter by detecting patterns on messages (with a DSL called PatternDSL) and for performing rule-based actions (with the RulesDSL language), like sending a *message*. In order to benefit from the high use of social networks nowadays, we consider very interesting to be able to integrate open data sets with data coming from social networks. For the example, we use Twitter, but similar to the different open data formats, *EagleData* provides an extension mechanism for adding support for additional social networks.

The execution scheme of an *EagleData* application consists in a loop that observes tweets mentioning a given account. A number of patterns (described using PatternDSL) are sought in those tweets, and the matching of any such pattern may trigger associated rules (defined using RuleDSL). The rules may perform actions, like generating tweets or private messages. These generated messages may contain information found both in the social network messages or in the open data sets. Because most of the information that users post in social networks does not follow a prearranged structure, *EagleData* provides facilities for detecting written language patterns. These patterns are the only information that the final user needs for using an *EagleData* application, although this does not necessarily imply the use of keywords or pre-defined hashtags, but following a flexible conceptual structure the application provider shall ensure that the user knows in advance. Other information, like geoposition, is intrinsically available with the tweets themselves as metadata.

Fig. 6 shows a very simplified meta-model. The concept *Phrase* represents the largest information unit a social network message can possibly hold. When a message matches a defined pattern, its associated rules are triggered. These rules follow an event-condition-action structure, which means that rules are evaluated whenever a mentioning tweet is received (event), and executed whenever the message matches a given pattern (the *condition*, referenced with the keyword on). For execution, rules may include a series of actions (starting with the keyword do). Currently, we support element-query and message-reply rules. In the action block, static data coming from open data sets can be combined with dynamic social network: both are uniformly available as variables in the domain. See [13] for further details.

Running example. Listing 4 shows an example of a simple pattern definition in line 1, which serves as the input user interface to our running example, expecting the bus line the user needs to know about. Any other character in the tweet is ignored. If multiple occurrences of a single concept are found, only the first one is considered. The pattern is named MessageLine, and is made of a string "Line" followed by the line id. This id is declared elsewhere as being either A or B. Patterns can also make use of synonyms for words (we use the Wordnet data base [8]) and perform approximate match (e.g., matching an expression where some vowels are missing). Whenever our running application receives a tweet matching the pattern from a user, the query addresses takes place (lines 5 – 6), which returns a list of the addresses of the queried bus line stops. The query result can be reused for the reply action, started by compose. The language supports string interpolation (line 8), which permits inserting the elements of lists using separators, or perform replacements. The message is sent back to the originator of the tweet.

```
1 phrase MessageLine ("Line", id)
2
3 on MessageLine
4 do
5   addresses:
6     select latPoint, longPoint where MessageLine.id = SantanderCityBus.id
7   compose
8     "https://www.google.com/maps/dir/@{addresses.latPoint,addresses.longPoint}[separator='/']"
```

Listing 4: Rule composing a message with a map of the stops for a given line.

6 Architecture and Tool Support

In this section we introduce an extended version of the prototype created for [13], now called *EagleData*. The tool is based on Eclipse. A schema of the *EagleData* architecture is shown in Fig. 7. *EagleData* is made of a Core component, which provides support for the schema discovery and data injection into the common internal representation. The Core component provides two extension points to support for additional data format handlers. The UI component provides Eclipse views and wizards for performing the queries using the three languages.

The framework supports the inference of DataDSL descriptions from data sets. Fig. 8 shows the wizard providing the assistance for refining the induced types. Datatypes and enumerations are discovered based on the values saved in the concrete data sources. The tool is able to generate template queries and rules based on the underlying open data set structure. Hence, once the DataDSL model is ready (label 2), we can invoke the generator of patterns and rules, with which a user is enabled to obtain an application in just a few minutes. The tool counts with a set of views to check the fragments and datatypes saved on the open repository, and the tweets that match with the patterns (label 3). The tool is available at `http://miso.es/tools/EagleData.html`. The web page contains videos showing the tool in action to define simple applications over open data sets.

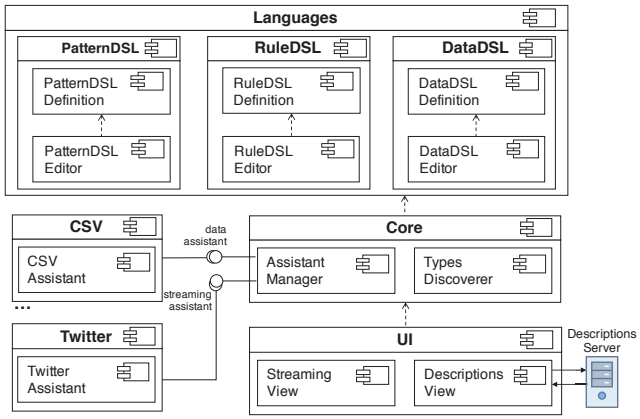

Fig. 7: Architecture of the proposal.

7 Related work

Some data pre-processing tasks in our framework have similarities with Extract-Transform-Load (ETL) approaches, especially from data warehouses [7]. Many commercial and open source tools exists nowadays supporting ETL processes, like Pentaho's PDI/Kettle[8], and data/API integration technologies like MuleSoft[9] also offer ETL support. Solutions such as CKAN[10] seems to be more complete, offering stream management for publishing and discovering datasets.

However, the purpose of those systems is different from ours. The goal of DataDSL is to describe the structure of existing data sets, both in format independent (through fragments) and format-dependent ways (through data sources). Then, both types of descriptions can be linked, and so the information in heteregeneous data sets can be related in such way. This approach is normally not followed in ETL, whose goal is moving data from some source data sets into a relational or multi-dimensional data base. Instead, DataDSL serves to describe and relate existing data sets.

DataDSL is a meta-modelling language especially tailored to describe open data. There exist well established meta-modelling languages, like the Meta-Object Facility (MOF) [9], and widely used frameworks like the Eclipse Modelling Framework (EMF) [14]. We refrained to use such approaches and opted instead for a customized language with native support for elements like format specification; fragments and data sources; and the possibility to add options, to be interpreted by the plugins supporting the specific formats. While this might be emulated in MOF/EMF e.g., by using annotations, native support for all these features yields a more natural syntax. Previously to build DataDSL, we also experimented with multi-level modelling technologies, like MetaDepth [12]. While it provided enhanced flexibility (by e.g., enabling the placement of more

[8] `http://goo.gl/G41Vaa`
[9] `http://mulesoft.com`
[10] `http://ckan.org/`

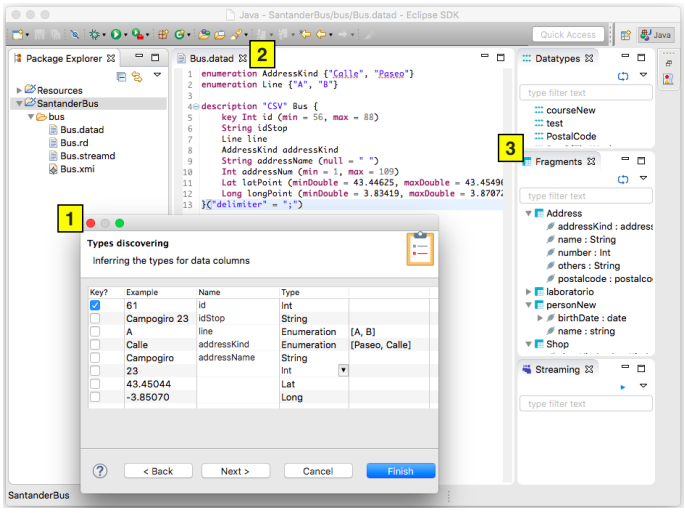

Fig. 8: Type refinement wizard and DataDSL editor

general fragments at higher meta-levels), it also lacked the useful mentioned primitives that DataDSL has. Moreover, it was not integrated with a framework to build social network applications. The need (raised by open data) to describe textual, tabular data is evidenced by the recent W3C recommendation [6]. While we aim at interperate with such recommendation, the goal is also to be *extensible* to arbitrary formats, including XML and even APIs.

Many works have tackled the problem of inferring schemas from data in different formats. For example, in [5] the authors propose an approach for discovering schemas (meta-models) from JSON data, and in [1] the authors propose an algorithm to obtain a concise DTD from XML data. We took ideas from [5] for our JSON support. As our framework is open, new formats can be added, and therefore algorithms for schema induction can be incorporated into EagleData.

BPM4People[11] extends BPMN enabling to model complex data flow, coming from a social domain, and [3] extends WebML to incorporate social primitives, permitting cross-platform operations for several social networks. While these are more general languages, our DSL approach aims to be much lighter, and specifically target to make accesible open data through social networks.

Hence, altogether, our work is novel with respect to existing works, combining a DSL for data description of heterogeneous open data sets, with DSLs for the description of applications over social networks, to automate the generation of open data applications.

8 Conclusions and future work

In this paper we have presented DataDSL, a DSL to describe the structure of heterogeneous *open data* sets. The DSL is integrated within *EagleData*, an MDE

[11] `http://bpm4people.webratio.com/`

framework to create applications over social networks. We have presented a working prototype and a running example consisting in a service to obtain a map-based representation of a bus line, with interaction via Twitter.

While we have used DataDSL to describe static open data, we will tackle its integration with open APIs. Hence, with DataDSL the underlying data schema can be described in the style of [5], and options can be used to describe the available operations of the API. We are currently working in providing *EagleData* with visualization outputs, and also automating the process of mapping data source descriptions to fragments. Our goal is to be able to define programs and services on fragments, that would become reusable to particular data sources when the fragments are reused. To simplify even more the development of open data applications, we will build higher-level DSLs specially targeted for end-users, whose execution will rely on our *EagleData* framework and its three DSLs.

References

1. G. J. Bex, F. Neven, T. Schwentick, and K. Tuyls. Inference of concise DTDs from XML data. In *VLDB*, pages 115–126. ACM, 2006.
2. C. Bizer, T. Heath, and T. Berners-Lee. Linked data - the story so far. *International Journal on Semantic Web and Information Systems (IJSWIS)*, 5(3):1–22, 2009.
3. M. Brambilla and A. Mauri. Model-driven development of social network enabled applications with webml and social primitives. In *ICWE Worksh.*, volume 7703 of *LNCS*, pages 41–55. Springer, 2012.
4. P. Hitzler, M. Krotzsch, and S. Rudolph. *Foundations of Semantic Web technologies*. CRC Press, 2010.
5. J. L. C. Izquierdo and J. Cabot. Discovering implicit schemas in JSON data. In *ICWE*, volume 7977 of *LNCS*, pages 68–83. Springer, 2013.
6. G. K. Jeni Tennison and I. Herman. Model for Tabular Data and Metadata on the Web. `https://www.w3.org/TR/tabular-data-model/`, 2015.
7. R. Kimball. *The Data Warehouse ETL Toolkit: Practical Techniques for Building Dimensional Data Warehouses*. John Wiley, 1996.
8. G. A. Miller. Wordnet: A lexical database for english. *CACM*, 38(11):39–41, 1995.
9. OMG. MOF 2.5. `http://www.omg.org/spec/MOF/2.5/`, 2015.
10. Open Data Commons. `http://opendatacommons.org/`.
11. E. Rahm and H. H. Do. Data cleaning: Problems and current approaches. *IEEE Data Eng. Bull.*, 23(4):3–13, 2000.
12. A. M. Segura, J. de Lara, and J. S. Cuadrado. ODaaS: Towards the model-driven engineering of open data applications as data services. In *EDOCW'14*, pages 335–339. IEEE Computer Society, 2014.
13. A. M. Segura, J. de Lara, and J. S. Cuadrado. Rapid development of interactive applications based on online social networks. In *WISE*, volume 8787 of *LNCS*, pages 505–520. Springer, 2014.
14. D. Steinberg, F. Budinsky, M. Paternostro, and E. Merks. *EMF: Eclipse Modeling Framework, 2^{nd} Edition*. Addison-Wesley, 2008.
15. M. Völter and T. Stahl. *Model-driven software development*. Wiley, 2006.