5

Tools

Kevin Feeney[1], Christian Dirschl[2], Katja Eck[2], Dimitris Kontokostas[3], Gavin Mendel-Gleason[1], Helmut Nagy[4], Christian Mader[4] and Andreas Koller[4]

[1]Trinity College Dublin, Ireland
[2]Wolters Kluwer Germany, Germany
[3]University of Leipzig, Germany
[4]Semantic Web Company, Austria

5.1 Model Catalogue

5.1.1 Introduction

Careful management of metadata is essential for the effective reuse of data and the correctness of any software designed for processing the data. Metadata may capture best practice in a domain and as such the reuse of metadata can proliferate best practice. Traditionally, most metadata is usually captured implicitly, and embedded in the software or system that use the data. Motivation for developing a tool for managing and curating metadata includes data and software interoperability, documenting metadata as models and building a platform for automatically generating software systems from models.

Without easily available metadata, determining the compatibility of datasets and software systems becomes challenging. The problem is compounded where there are multiple datasets and software systems, as the metadata for each needs to be documented manually, ex post facto, in order to determine compatibility. There are several challenges to collecting, managing and documenting metadata. Metadata may have only been documented conceptually, and there may be ambiguities in the metadata that require further clarification.

The Model Catalogue is an online tool that supports the capture and documentation of metadata as generic and reusable models. The tool facilitates collaboration between metadata creators and potential users. The system defines a core language for describing metadata, which enables sharing, documentation and reuse of metadata. The tool uses standards-based concepts for registration, versioning and a standard four level architecture with an API for interoperability with external tools.

Metadata is data that describes data by capturing essential relationships, classifications and atomic data elements. Metadata is important for data reuse and underpins the software that stores, processes and analyses any dataset. This information captures the meaning and guides interpretation of the data. However, in typical usage, metadata is captured implicitly and embedded within the software system that uses the data. This is problematic because metadata is essential to reuse the data outside the original context.

Metadata is also essential to interoperability. The compatibility of two datasets may only be determined by examination of the datasets' metadata. This can be an arduous task where the metadata is embedded with the software and the difficulty can be compounded where there are several datasets involved. Software interoperability, where independent systems can share messages and data, relies on compatible metadata for the involved systems.

When metadata is separated from the use of the data, two important efficiencies are made possible. Firstly, the metadata can be explored independently from the data, so the compatibility of datasets and interoperability of software can be determined without analysis or re-collection of the data. Metadata can also be captured in a generic form, as metadata models and software systems for managing data can be generated from sufficiently detailed metadata models. Other benefits to a metadata-oriented approach include reasoning and discussion about the underlying model with experts, reuse of metadata and creating a map of data across the systems in an organisation.

Metadata can also be used to encode the established best practice in a domain. The definitions of how data should be structured, what data should be captured and the intended use for the data can be captured in metadata. For example, metadata can specify the resolution and level of granularity for data capture. When metadata is embedded directly with the use of the data, reuse of the best practice can be a challenge. Where metadata is encoded as generic models and the models are documented independently of any system, the essential information about a domain becomes more readily available for reuse. Software developers can use these metadata models to encode the best practice of a domain. Reusing metadata models helps proliferate established best practice.

Generic models of metadata can encode the best practice of a domain. This can be taken further by allowing potential data users to reuse subsets of a generic data model. This means that only the relevant data elements and classifications from a model can be reused and repurposed in a new context. However, reusing, changing or merging models for a new context is challenging unless elements have exactly the same meaning in both contexts. Subtle differences between models' elements can make the reuse of models and data problematic, especially when changes in versions are not documented or tracked. Each element of metadata must be documented in order to determine the compatibility of that data element for reuse in a new context.

5.1.2 Model Catalogue

The Model Catalogue is a toolkit for creating, sharing, and updating data models. The system uses a layered architecture, described below, which allows for a number of possible Graphical User Interfaces. The data models are descriptions or specifications of data artefacts, objects, or implementations.

A data model may describe or specify:

- a dataset or database holding data of interest
- a request for data from a collection of databases or datasets
- a standard for developers to work to
- a form used for data entry
- a message carrying data from one system to another
- a report offered or required
- a workflow or pathway in which data are collected and used.

A data model will be simpler than the artefact it describes. It need not consider every aspect of the artefact or implementation, only the data items of interest and the relationships between them.

A data model will be more comparable. It is easier to compare data models, written in a single modelling language, than to compare artefacts implemented using a range of different technologies.

A data model will be more re-usable. It is safer to produce a new artefact by copying parts of a model than by copying parts of the existing implementation. There are additional advantages – cost, consistency – if the new artefact can be generated automatically.

5.1.2.1 Architecture

The Model Catalogue has been built in a traditional layered architecture, facilitating access through both manual and programmatic means. This structure is shown in Figure 5.1. At the base layer is a relational database: in the current implementation, we use PostgreSQL as a stable, tried-and-tested open source solution. To ensure consistency of the underlying data, we insist

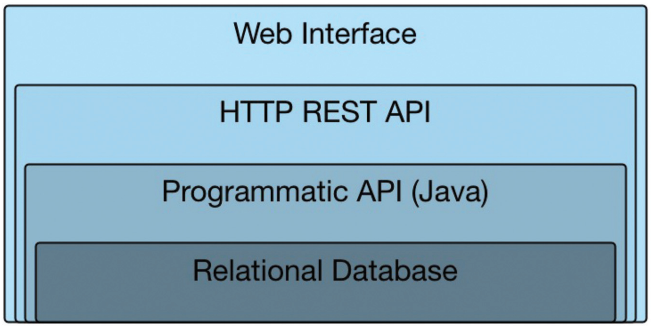


Figure 5.1 The layered architecture of the Model Catalogue.

that all data access and manipulation is through a programmatic API: this is currently implemented in Java and is used by the higher levels of the stack, but can also be used by external tools built with Java.

At the next level up is a Web-based REST API. This can be used to programmatically access remote deployments of the catalogue, and is language-independent: it can be used by any sophisticated toolset to interact with a publicly available catalogue. The final layer is the human-readable Web interface. This provides an attractive view of the data to facilitate a range of use cases, accessible on modern Web browsers using standard interface patterns for security and interaction. The Model Catalogue provides a generic API so that any Java-based tool, including Eclipse, can be integrated programmatically.

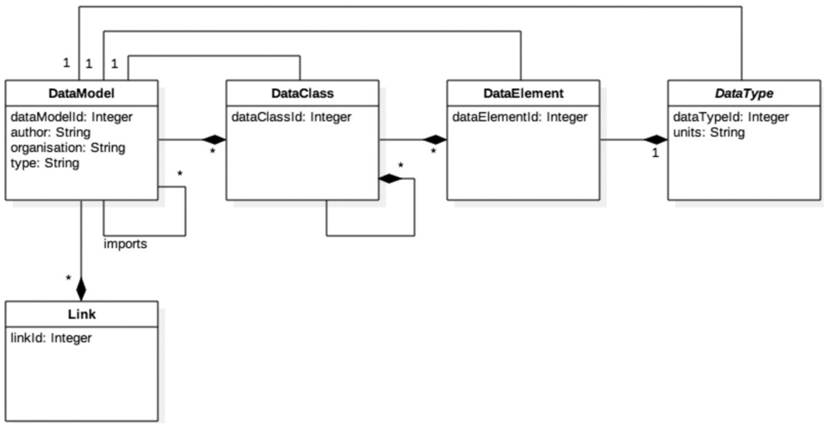


Figure 5.2 Core concepts – data model components – within the Model Catalogue.

At the core level, the catalogue contents are structured in a simple hierarchy, a subset of which is shown in Figure 5.2. At the top level is a Data Model, which may be versioned and published. A model contains a number of Data Classes, which provide categorisation or structuring. At the lowest level are Data Elements, which describe individual data points. Each Data Element has a Data Type, which may be either: a Primitive, such as String or Integer; an Enumerated Type, where allowed values may be defined in the context of this model or taken from a larger terminology; or a Reference Type, denoting a pointer to data from another Data Class. Figure 5.3 shows this hierarchy.

Components of data models may be linked: a link between two elements can represent that two data items are equivalent, that one is derived from another, or that one is different to another. For example, one element in a dataset might be linked to a definition within a data-standard to assert that the guidelines have been followed in the collection of that data item. Another item might be linked to that same data standard definition with a "different to" annotation to assert that although these data points might look the same, there is a subtle difference that may be explained in the item's description.

The Model Catalogue Web interface currently supports the fundamental use case requirements: browsing, searching and editing/updating models. These are described in the next sections.

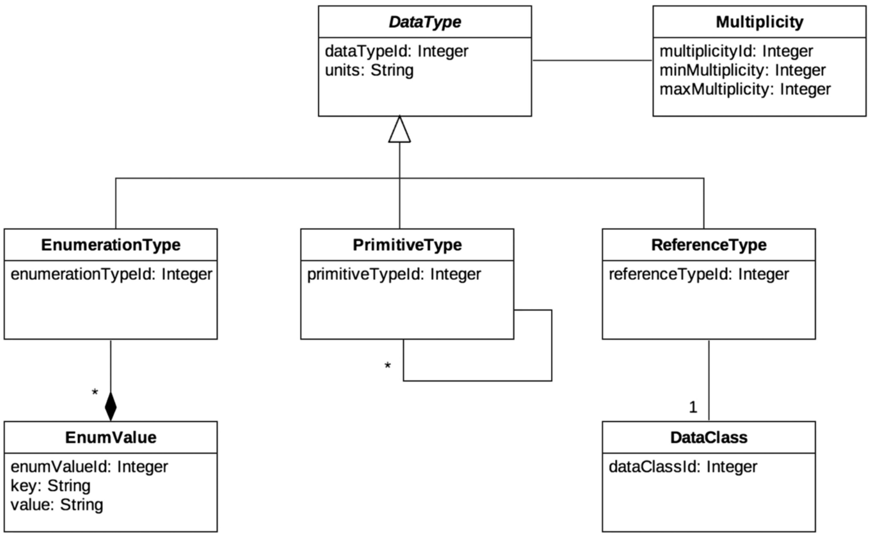


Figure 5.3 A model showing the datatypes represented in the Model Catalogue.

5.1.2.2 Searching and browsing the catalogue

There may be many models in the catalogue with the same name. For this reason, the contents of the catalogue can be searched or browsed using system metadata:

- model name
- editors – the catalogue users with write access
- status – draft or finalised
- catalogue version
- creation date
- last edit date
- imports from – the list of models which use the datatypes defined within this model

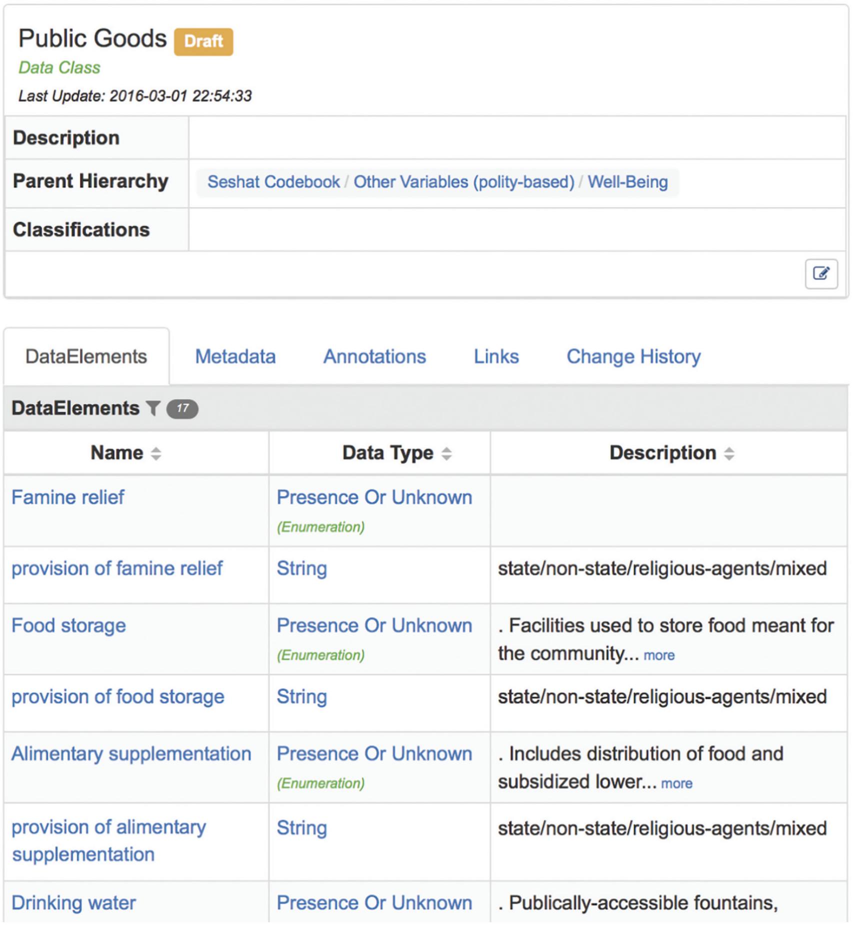
or by user-supplied metadata:

- owner(s) – responsibility
- author(s) – credit
- organisation(s) – authority
- external version name/label
- external release name/label

Other searching and browsing requirements will be addressed using annotations and classifications. Users with write access to a model can add annotations against templates provided for that model type. Users with read access can classify models and model components to which they have read access.

The two-panel view of the catalogue, as shown in Figure 5.4, provides a familiar interface, with the structured model contents in the left-hand pane, and the currently viewed data model component in the larger right-hand pane. A view of a data model displays metadata about that model and a list of all child data classes; similarly, the view of a data class shows metadata about that class, along with all contained data elements. The view of a data element, as illustrated in Figure 5.5, shows detailed information about the datatype, including any enumerated values, along with the description, and metadata about its place within the model, its current publication status, and when it was last updated.

The Web interface currently offers a basic keyword search across different component types within the catalogue. This helps potential users of collected datasets find data items that may be useful to their work. Figure 5.6 shows this keyword search in action for the Seshat Code Book model.


Public Goods Draft
Data Class
Last Update: 2016-03-01 22:54:33

Description	
Parent Hierarchy	Seshat Codebook / Other Variables (polity-based) / Well-Being
Classifications	

DataElements 17

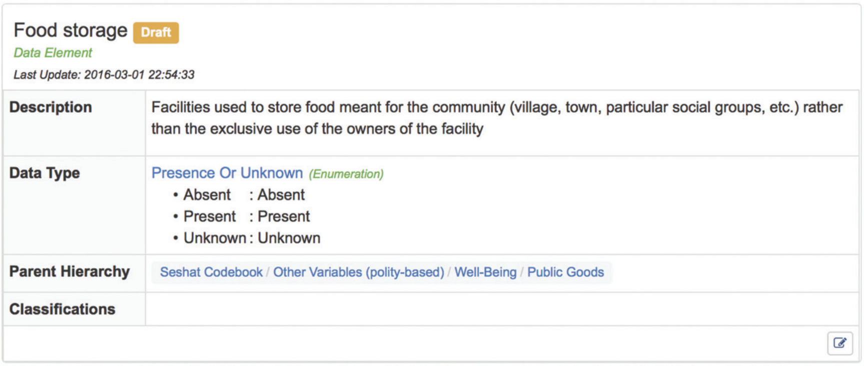
Name	Data Type	Description
Famine relief	Presence Or Unknown *(Enumeration)*	
provision of famine relief	String	state/non-state/religious-agents/mixed
Food storage	Presence Or Unknown *(Enumeration)*	. Facilities used to store food meant for the community... more
provision of food storage	String	state/non-state/religious-agents/mixed
Alimentary supplementation	Presence Or Unknown *(Enumeration)*	. Includes distribution of food and subsidized lower... more
provision of alimentary supplementation	String	state/non-state/religious-agents/mixed
Drinking water	Presence Or Unknown	. Publically-accessible fountains,

Figure 5.4 Model Catalogue interface: browsing the Seshat code book.

5.1.2.3 Editing the catalogue contents

Editing a draft data model is a simple process whereby most fields may be edited in place. Figure 5.7 shows the editing of basic data element details: name and description. All modifications are recorded. A change log is created and preserved for each model, showing the time, date, and user responsible for each change.

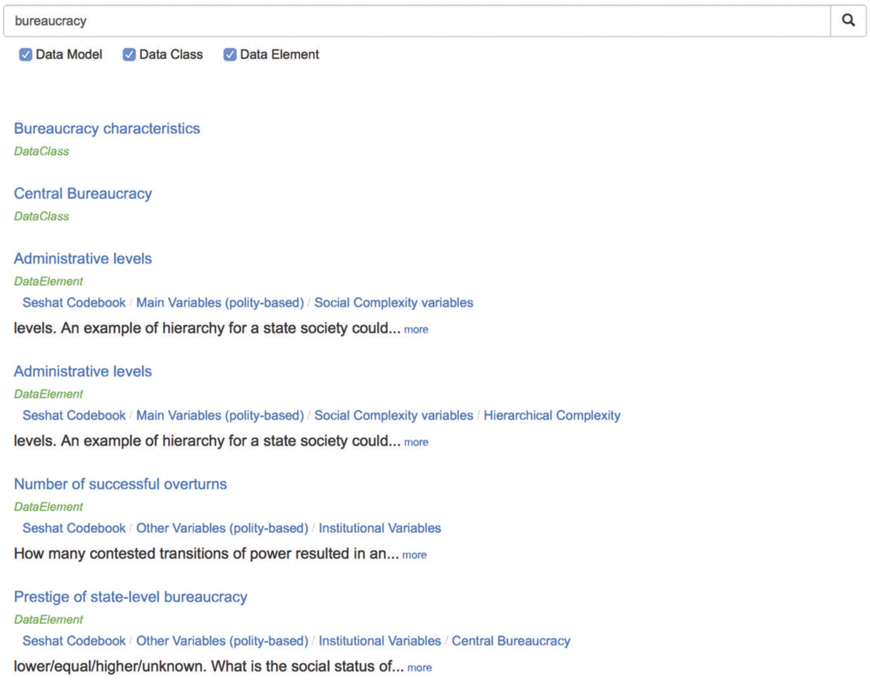
Although people may currently use the catalogue to collaboratively edit a model, simultaneous updates can cause confusion. In the next version of the system, at the point of opening a component for editing, a user will be alerted



Food storage Draft
Data Element
Last Update: 2016-03-01 22:54:33

Description	Facilities used to store food meant for the community (village, town, particular social groups, etc.) rather than the exclusive use of the owners of the facility
Data Type	Presence Or Unknown *(Enumeration)* • Absent : Absent • Present : Present • Unknown : Unknown
Parent Hierarchy	Seshat Codebook / Other Variables (polity-based) / Well-Being / Public Goods
Classifications	

Figure 5.5 Model Catalogue interface: data element view.



bureaucracy

☑ Data Model ☑ Data Class ☑ Data Element

Bureaucracy characteristics
DataClass

Central Bureaucracy
DataClass

Administrative levels
DataElement
Seshat Codebook / Main Variables (polity-based) / Social Complexity variables
levels. An example of hierarchy for a state society could... more

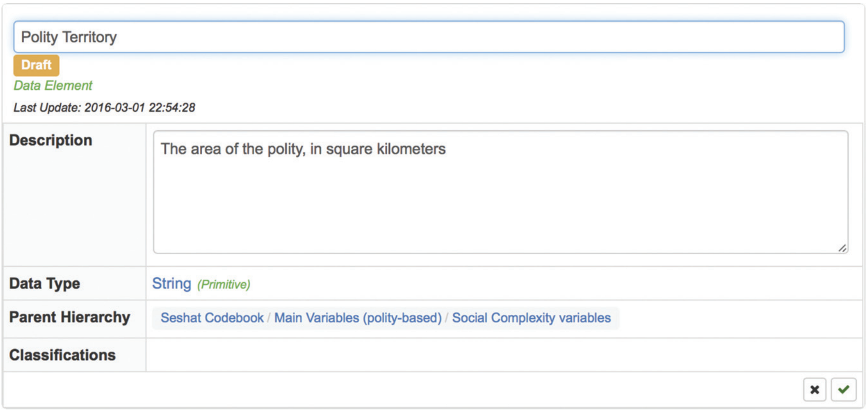
Administrative levels
DataElement
Seshat Codebook / Main Variables (polity-based) / Social Complexity variables / Hierarchical Complexity
levels. An example of hierarchy for a state society could... more

Number of successful overturns
DataElement
Seshat Codebook / Other Variables (polity-based) / Institutional Variables
How many contested transitions of power resulted in an... more

Prestige of state-level bureaucracy
DataElement
Seshat Codebook / Other Variables (polity-based) / Institutional Variables / Central Bureaucracy
lower/equal/higher/unknown. What is the social status of... more

Figure 5.6 Model Catalogue interface: keyword search for a data item.

if that same component is currently open for editing by other user(s). The identity of the other user(s) will be displayed, together with the time at which they started their edit. At the point of saving an edit upon a component, a user will be alerted if the component has been updated, by other user(s), since the


Polity Territory

Draft

Data Element

Last Update: 2016-03-01 22:54:28

Description	The area of the polity, in square kilometers
Data Type	String (Primitive)
Parent Hierarchy	Seshat Codebook / Main Variables (polity-based) / Social Complexity variables
Classifications	

Figure 5.7 Model Catalogue interface: editing a data item.

edit began. In either case, the current can choose to proceed with the edit or the save.

Once finalised, those with "write access" may further annotate a model: while annotations are potentially useful and valuable, they are not taken to contribute to the finalised interpretation of the model or model component.

The contents of a model, and any associated links, cannot be changed after finalisation. However, the lists of users with write or read access can be updated by any of those with write access.

Finalisation cannot be undone: instead, a new version must be created. This can be done by any of those with write access to the finalised model: the result is a copy of the model with a link to the existing, finalised model indicating that it is indeed a "new version".

There is no need to create a new version of a model that has not been finalised: a user with write access can simply update the contents of the model in its current "draft" state; an edit log is maintained automatically.

If two users with write access to a finalised model both create new versions, then the development will branch. A branched development may be merged by a user with write access to finalised models in both branches, creating a model that is a new version of both.

A user who does not have write access to a finalised model may create a copy of the model (or a component) that is not a new version but is instead "based upon" that item. There is no requirement that the new item should have the same intended interpretation.

When a new version of a finalised model is created, the result is a draft model with a complete copy of the finalised model contents and metadata, including all links and annotations.

Where a model or a component is the target of a classifier or label, whether the classification concerned is updated to include the new version depends upon the properties of the classification (and not the model). The options are:

- add new version to classification alongside existing version
- add new version, remove existing version
- ignore new version

In the last case, the new version can then be added manually, if required.

5.1.2.4 Administration

The Model Catalogue administrator(s) can register new users as editors or readers. They can manage models for which there are no longer any other users with write access.

An editor can create new models and add or delete annotations for existing models to which they have write access. They can create classifications referring to any items to which they have read access. They can explore models to which they have read access.

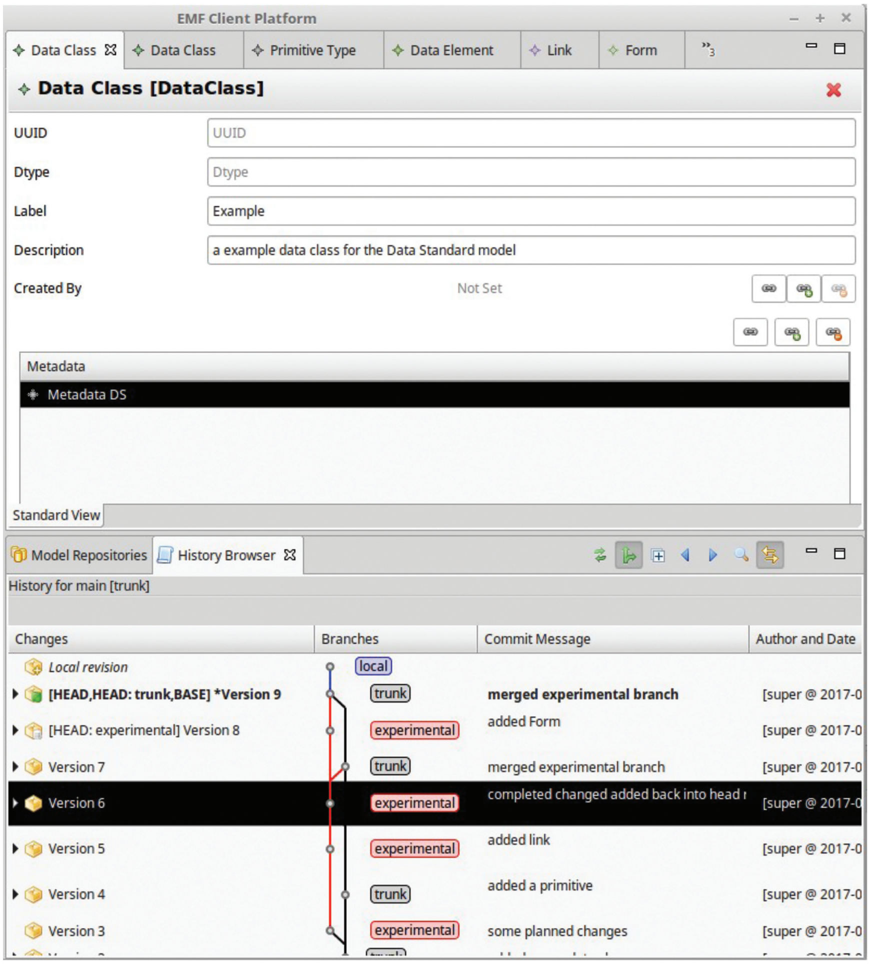
A reader can add or delete annotations for existing models to which they have write access. They can explore models to which they have read access.

Where there are several users with write access to the same model, the possibility of conflict arises. Any conflict may be addressed through interaction with the administrator(s), who are able to modify any aspect of the catalogue contents.

Any editor can create a user group. Any member of a user group can add or remove members. User groups can be included in model access control lists.

5.1.2.5 Eclipse integration and model-driven development

Core parts of the catalogue functionality are integrated with the Eclipse Modeling Framework (EMF). EMF is fundamental to the majority of model-driven development tools within Eclipse and is also used as the basis for DSLs and transformations. This allows existing model-driven tools within Eclipse to take advantage of the catalogue in order to reuse components of models, increasing the speed of development, and allowing data linking and interoperability between tools built within the framework.



Data Class [DataClass]

UUID	UUID
Dtype	Dtype
Label	Example
Description	a example data class for the Data Standard model
Created By	Not Set

Metadata

Metadata DS

History for main [trunk]

Changes	Branches	Commit Message	Author and Date
Local revision	local		
[HEAD,HEAD: trunk,BASE] *Version 9	trunk	**merged experimental branch**	[super @ 2017-0
[HEAD: experimental] Version 8	experimental	added Form	[super @ 2017-0
Version 7	trunk	merged experimental branch	[super @ 2017-0
Version 6	experimental	completed changed added back into head r	[super @ 2017-0
Version 5	experimental	added link	[super @ 2017-0
Version 4	trunk	added a primitive	[super @ 2017-0
Version 3	experimental	some planned changes	[super @ 2017-0

Figure 5.8 Model Catalogue Eclipse Integration.

Furthermore, the EMF integration allows new Model Catalogue components to be built in a MDE fashion – the screenshot in Figure 5.8 shows an automatically generated interface for interacting with the catalogue data, including automatic change management to track multi-user updates. Another auto-generated component stores all versions of every model to disk.

The Model Catalogue also has a plugin architecture, providing extension points through which new functionality can be built and dynamically integrated. For example, two key extension points are those of Importer and Exporter: developers can write their own importers and exporters using the Model Catalogue Java API to automatically document models in their own language, or to use the catalogue as an interface for compiling new models from existing ones. A developer may choose to write a data model importer that documents the usage of a no-SQL database, and an exporter that generates queries to retrieve that data and insert it into an SQL database.

Further plugins are being developed for bespoke types of data model, and custom interfaces that can be used to display and edit particular types of model in a more familiar fashion. For example, a graphical editor for UML diagrams, or a builder tool for designing data entry forms. This plugin architecture also allows custom configurations of the catalogue to be deployed – using just those plugins necessary for the context: providing a better user experience and requiring minimal system resources.

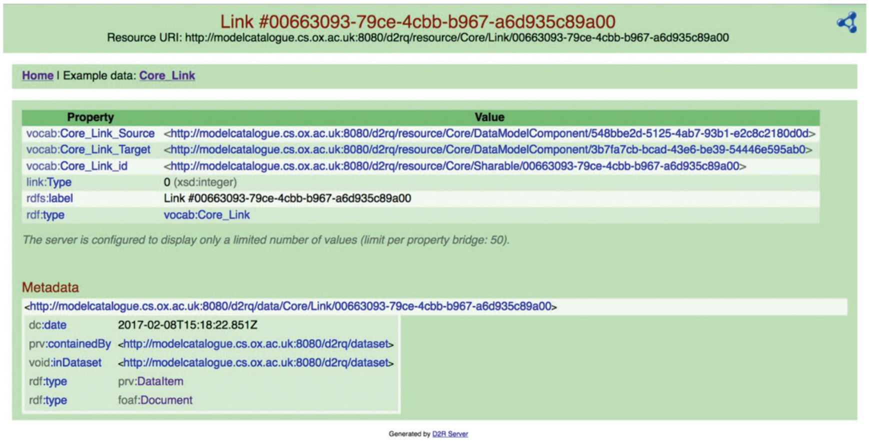
5.1.2.6 Semantic reasoning

Semantic links are created in the catalogue to associate parts of different models – typically Data Elements – to assert that they are similar in meaning or use. This allows descriptions of meaning to be reused: by asserting: "this element is the same as that one" a modeller may take advantage of definitions in other models, reducing the effort in documentation. These links also give us the formal notion of "semantic interoperability": that data from two sources may be combined for a particular purpose.

To reason about this semantic interoperability property, using off-the-shelf reasoning tools, it is useful to view the data in terms of triples. In order to do this, the D2RQ tool[1] allows users to expose internal relational data as RDF triples (see screenshot in Figure 5.9). Although the mapping requires some further customisation for easier use, the mapping is sufficient to reason about key properties of the semantic links: circularities in the transitive "same-as" link, and contradictions in definitions using the "same-as" and "not-same-as" links.

A side effect of making this representation available is that the catalogue contents can be linked to other open datasets. For example, catalogue metadata may be linked to other published artefacts with Dublin Core, provenance information may be attached with PROV, linking to existing tagging and

[1]http://d2rq.org


Link #00663093-79ce-4cbb-b967-a6d935c89a00
Resource URI: http://modelcatalogue.cs.ox.ac.uk:8080/d2rq/resource/Core/Link/00663093-79ce-4cbb-b967-a6d935c89a00

Home | Example data: Core_Link

Property	Value
vocab:Core_Link_Source	<http://modelcatalogue.cs.ox.ac.uk:8080/d2rq/resource/Core/DataModelComponent/548bbe2d-5125-4ab7-93b1-e2c8c2180d0d>
vocab:Core_Link_Target	<http://modelcatalogue.cs.ox.ac.uk:8080/d2rq/resource/Core/DataModelComponent/3b7fa7cb-bcad-43e6-be39-54446e595ab0>
vocab:Core_Link_id	<http://modelcatalogue.cs.ox.ac.uk:8080/d2rq/resource/Core/Sharable/00663093-79ce-4cbb-b967-a6d935c89a00>
link:Type	0 (xsd:integer)
rdfs:label	Link #00663093-79ce-4cbb-b967-a6d935c89a00
rdf:type	vocab:Core_Link

The server is configured to display only a limited number of values (limit per property bridge: 50).

Metadata

<http://modelcatalogue.cs.ox.ac.uk:8080/d2rq/data/Core/Link/00663093-79ce-4cbb-b967-a6d935c89a00>

dc:date	2017-02-08T15:18:22.851Z
prv:containedBy	<http://modelcatalogue.cs.ox.ac.uk:8080/d2rq/dataset>
void:inDataset	<http://modelcatalogue.cs.ox.ac.uk:8080/d2rq/dataset>
rdf:type	prv:DataItem
rdf:type	foaf:Document

Generated by D2R Server

Figure 5.9 Screenshot showing RDF representation of catalogue contents.

folksonomies through the Modular and Unified Tagging Ontology (MUTO), and design intent may be linked to model components using DIO.

To assist with this linking, namespaces can be added to metadata elements within the catalogue. These can be used to indicate fields for linking in the RDF representation, or can be used by plugins to configure generated artefacts, such as adding constraints to systems generated with Semantic Booster, or shaping XSD outputs to match existing specifications.

5.1.2.7 Automation and search

Previous implementations of metadata registries have shown that it can be difficult to encourage users to carefully document the whole data model to a level that is sufficient for potential users of the data. In particular, when dealing with models at scale, even simple tasks like finding a data element in another model to link to, or comparing multiple versions, can be complicated and time-consuming. In order to improve usability, a number of features assist modellers in using the tool effectively and efficiently.

At the heart of this effort is greater power in searching across many hundreds or thousands of data elements, in order to find related items, create semantic links between items, and import or reuse whole model components. Lucene and Solr[2] help with indexing, and allow faster and more flexible searching using keywords, related terms and intelligent suggestions. The

[2]http://lucene.apache.org

speed improvements offered by these tools make the Model Catalogue as a whole scalable for domains with large numbers of complex data models.

Finding similar elements to link to can reduce the time it takes to document a data element. To allow users to find similar items, an autosuggest feature will find potential matches across all models, or a particular model, based on datatypes, element names, and text matching in the description.

Using the semantic reasoning described above can also assist users in creating semantic links, using the transitive properties to help them find related data elements not already explicitly linked. Such reasoning can also help find relations between larger model components: for example, linking two data classes where the component data elements of each class are already linked.

Comparing different models is also something that users need extra assistance with – especially comparing multiple incremental versions of the same model. To aid users in this activity, there is a Web interface, supported by back-end API methods. Viewing two models side by side, with differences highlighted, provides a user-friendly experience that will be familiar to those with experience using traditional "diff" style tools.

5.1.3 Semantic Booster

5.1.3.1 Introduction

The data belonging to an organisation is often its most valuable asset: traditionally payroll information and customer details, but more frequently entire business models are based on the gathering and dissemination of information. The software responsible for maintaining the integrity of this data, and its consistent interpretation, will be critical to the ongoing function of the business. Moreover, organisations need to evolve and adapt, and it will be essential for the software and data to follow changes to business rules, and for the semantics of the data to remain clear and unambiguous.

Building software that is both robust and adaptable brings with it many challenges. The typical development process for robust systems for use in safety-critical applications will be slow and labour-intensive. Agile development processes are key to maintaining and evolving software, but are not effective when dealing with large complex datasets, or where the guarantee of software correctness is reliant on more than simple testing.

Automation and abstraction provide some solutions to these difficult problems. By automating part – or indeed all – of the code generation process, the influence of human error can be reduced, and the subsequent speed-up can decrease the time necessary to adapt or evolve generated, working code. By using suitable abstractions to model the software's intended function, correctness may be more carefully clarified, and the scope of updates or evolutions may be more immediately realised.

The MDE approach attempts to combine both automation and abstraction. Models may be domain-specific: comprehensible to non-technical domain-experts, with automated processes generating software components to match. In practice, however, such MDSE tools are either too specific, where models are used for not much more than customising or configuring a particular software artefact, such as in the generation of embedded systems; or too general purpose, where a wide variety of specifications may be expressed, but without the formality required for robust implementation, and in most cases where code generation must be supplemented by custom hand-written code.

The Booster tool has been written in an attempt to find the sweet spot between these two extremes. Models describe information systems: software components focussed on the correct management of business-critical data. The modelling language takes an object-oriented approach to modelling business concepts, but is supplemented with a formal, mathematical notation for describing relationships between entities, integrity constraints, business rules, and constraints upon interaction with data. The compilation process is complete: working implementations are generated with no manual intervention or addition required.

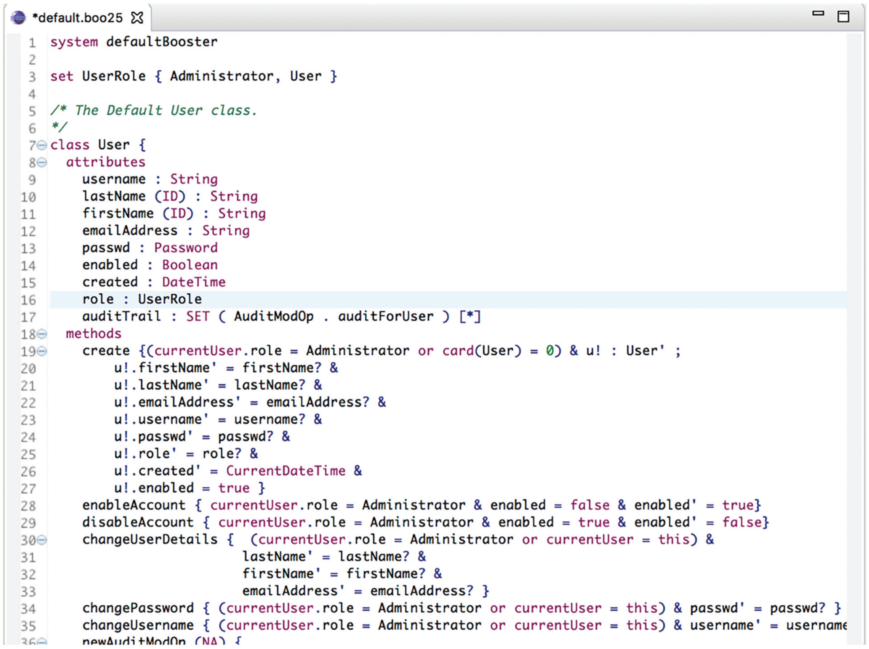
In order to make Booster more widely applicable – in particular to the domain of data-intensive systems – some key enhancements are necessary. Booster models are mostly without semantics: the meaning of entities or attributes is not recorded. This means that data collected and maintained within a Booster system may not be immediately re-usable within a different context. Although Booster is able to maintain and migrate data in the face of changing specifications, the meaning or context of these data may be lost. Booster has been integrated with the Model Catalogue, allowing metadata to be linked to each data item stored. As well as increasing the value and utility of the data, it allows domain experts to more carefully specify the functionality of the system, as well as permitting new notions of correctness.

5.1.3.2 Semantic Booster

The Booster tool takes as input a formal specification, written in the Booster language, and generates a complete working implementation. The Booster language takes inspiration from the UML, incorporating familiar object-oriented notions of classes, attributes and associations. This language is supplemented with a formal constraint language, inspired by the mathematics notations of Z and B, used in formal methodologies. These constraints can be used to define integrity constraints and business rules, in the form of class invariants, and pre- and post-conditions for methods.

The Booster language is supported by a custom editor written for the Eclipse IDE. This provides a number of features that aid developers, such as syntax highlighting, auto-suggestion, document outline, and code validation "as-you-type". Figure 5.10 shows a Booster specification being edited within the IDE.

Once a specification is completed, an automatic generator can be executed to generate an implementation. The generator consists of a number of stages, implemented as a pipeline (see Figure 5.11). In the first stage, the model is elaborated – this flattens the class hierarchy and in-lines any references to other parts of the model, essentially making explicit any default assumptions. The second stage of the pipeline is to apply a number of heuristics, to generate simple code from each constraint in the model. These heuristics have been



```
system defaultBooster

set UserRole { Administrator, User }

/* The Default User class.
*/
class User {
  attributes
    username : String
    lastName (ID) : String
    firstName (ID) : String
    emailAddress : String
    passwd : Password
    enabled : Boolean
    created : DateTime
    role : UserRole
    auditTrail : SET ( AuditModOp . auditForUser ) [*]
  methods
    create {(currentUser.role = Administrator or card(User) = 0) & u! : User' ;
        u!.firstName' = firstName? &
        u!.lastName' = lastName? &
        u!.emailAddress' = emailAddress? &
        u!.username' = username? &
        u!.passwd' = passwd? &
        u!.role' = role? &
        u!.created' = CurrentDateTime &
        u!.enabled = true }
    enableAccount { currentUser.role = Administrator & enabled = false & enabled' = true}
    disableAccount { currentUser.role = Administrator & enabled = true & enabled' = false}
    changeUserDetails {  (currentUser.role = Administrator or currentUser = this) &
                        lastName' = lastName? &
                        firstName' = firstName? &
                        emailAddress' = emailAddress? }
    changePassword { (currentUser.role = Administrator or currentUser = this) & passwd' = passwd? }
    changeUsername { (currentUser.role = Administrator or currentUser = this) & username' = username
    newAuditModOp (NA) {
```

Figure 5.10 A Booster specification edited with the Eclipse IDE.

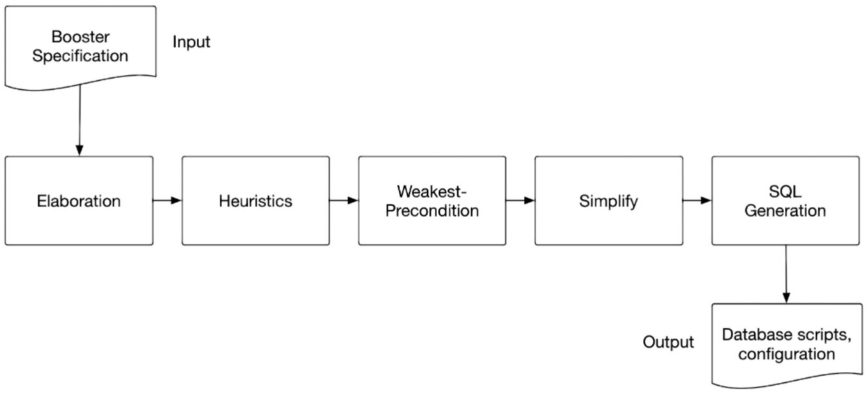


Figure 5.11 The Booster generation pipeline.

defined based on experience of developing information systems, and the code itself is written in an abstract, mathematical notation suitable for subsequent analysis.

The third stage of the pipeline is to generate additional code based on all constraints across the entire model. This process is similar to a ‘weakest-precondition’ calculation in formal methods and ensures the correctness of the final system: all business rules and integrity constraints are guaranteed to be considered and upheld in the final system.

The fourth stage of the pipeline is simplification: the previous steps generate large amounts of code, and much of it may be simplified to produce more efficient programs. The final stage is to generate a database implementation – the original implementation generates MySQL. This implementation includes database tables to store the core information, stored procedures to implement all data update methods, and additional system metadata to provide an object-relational mapping suitable for external users to interact with the system.

The completed database implementation can be used in conjunction with a bespoke API and user interface to provide a complete working system. This structure is shown in Figure 5.12; the Web-based Booster interface is shown in a screenshot in Figure 5.13.

The Booster approach embodies an approach in which the integrity of the data is all important. The rigorous calculations and code generation in the development pipeline ensure that for any form of update to the data, all business rules and constraints are considered, guaranteeing that no data integrity constraints will be invalidated as a result of any subsequent change.

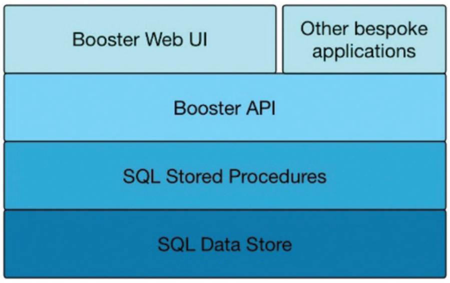
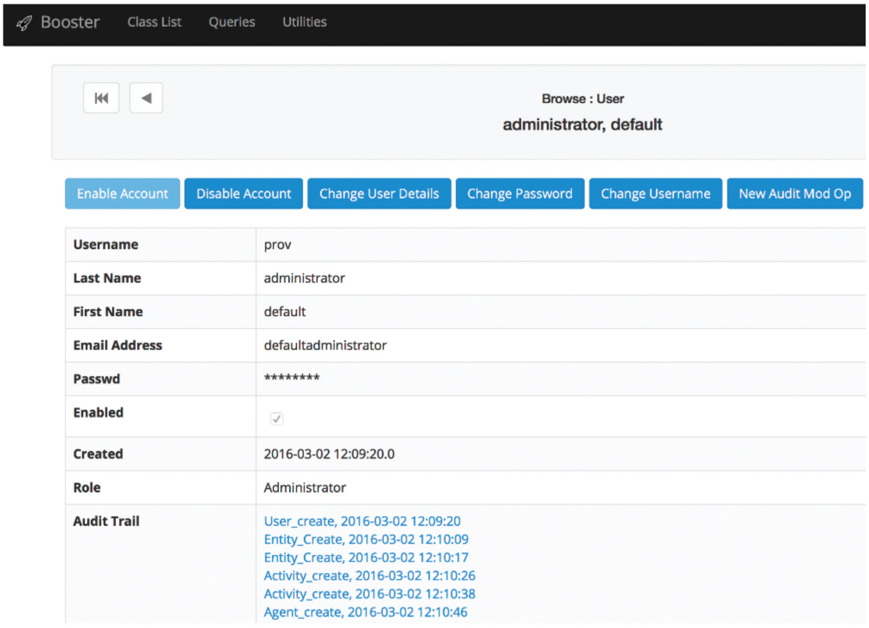


Figure 5.12 The architecture of a Booster information system.



Username	prov
Last Name	administrator
First Name	default
Email Address	defaultadministrator
Passwd	********
Enabled	✓
Created	2016-03-02 12:09:20.0
Role	Administrator
Audit Trail	User_create, 2016-03-02 12:09:20 Entity_Create, 2016-03-02 12:10:09 Entity_Create, 2016-03-02 12:10:17 Activity_create, 2016-03-02 12:10:26 Activity_create, 2016-03-02 12:10:38 Agent_create, 2016-03-02 12:10:46

Figure 5.13 The Booster Web-based user interface.

Access to the data is through a carefully managed API, which ensures that data are only manipulated in the manner specified in the original model.

In order to apply the Booster toolset to data-intensive systems, a number of enhancements have been made. By integrating Booster with the Model Catalogue, the stored semantic metadata can be used to enrich and inform the development of Booster specifications, and to ensure the consistency and reusability of the data held within Booster systems.

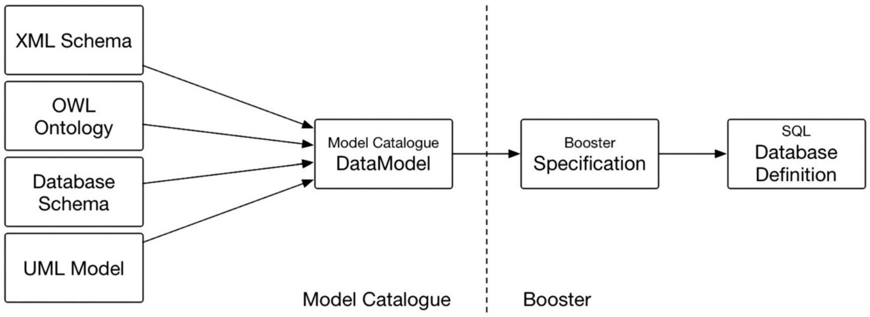


Figure 5.14 Generating Booster systems from Model Catalogue models.

Booster specification generation from Model Catalogue models

The first enhancement has been to build functionality to generate Booster specifications automatically from models described in the Model Catalogue. By using model components from the catalogue in our specification, we can ensure that the generated software can conform with existing data standards, or UML specifications, or can match data formats described using XML schema or OWL ontologies. This automation also allows domain-experts to begin generating software components without the need for development effort. Figure 5.14 shows the Semantic Booster pipeline, where platform-specific representations can be loaded into the Model Catalogue, and Booster systems generated via the Booster compiler, with no manual intermediate steps.

The structure of a model in the Model Catalogue is in many ways similar to the structure of a Booster specification. However, a number of transformations are required to take the tree-structured model and turn it into the flatter specification required for Booster. As well as this structural transformation, some more practical changes to Booster were required – in particular allowing all specification constructs to take a human-readable name, in addition to the standard identifiers required by the constraint language. By hiding system identifiers below the API level, the resulting information system is easier to use for subject-matter experts, and ambiguity may be reduced.

At the outermost level, a DataModel in the Model Catalogue is translated into a System in Booster. Every EnumerationType within the DataModel is converted into a Booster Set, using the human-readable names, and generating system identifiers if necessary. All DataClass components from the model, at any level in the hierarchy are converted into Booster Class declarations. Where one DataClass is contained within another in the catalogue DataModel, a bi-directional optional-to-one association between the two classes is created in Booster, corresponding to the notion of ownership, or composition in UML.

Every DataElement in the catalogue is translated into an Attribute in the Booster model, with multiplicities maintained. Those elements with a PrimitiveType datatype in the catalogue are mapped to the appropriate Booster primitive type. Similarly, EnumerationType elements in the catalogue get mapped to equivalent Set valued attributes in Booster. Finally, ReferenceType valued attributes are converted to bi-directional associations to the relevant class in Booster.

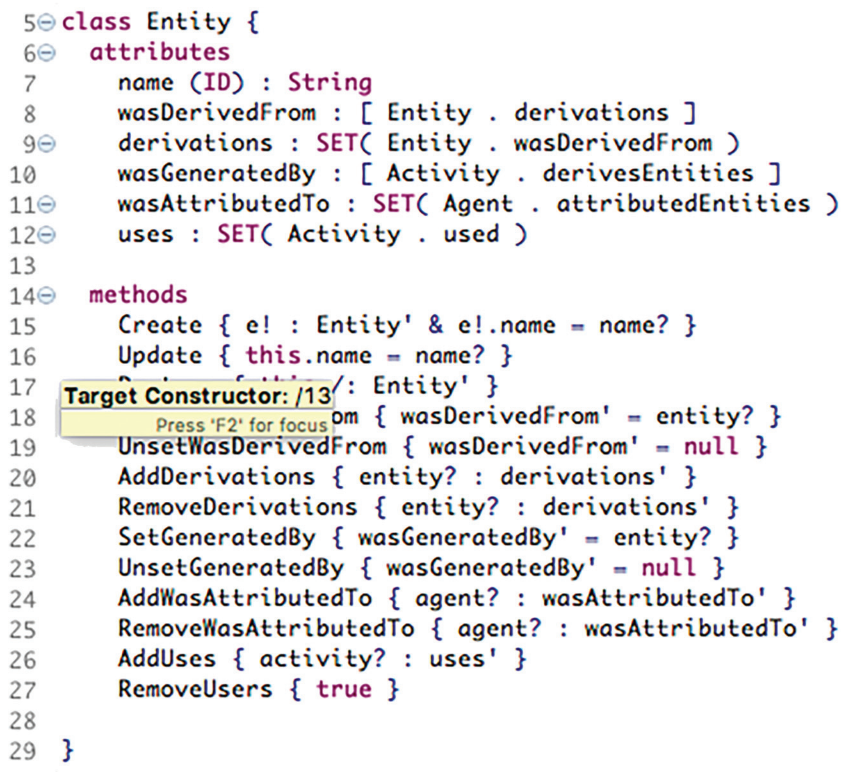
As part of this transformation, a basic collection of update methods is generated. For each Booster class, methods are created for creating, updating and destroying objects of that class. In addition, for every bi-directional association created by the transformation, methods are created for adding and removing links. Where these associations correspond to composition or aggregation, appropriate constraints are added to maintain the ownership properties. Figure 5.15 shows the generated specification for part of the PROV-DM Core Structures model.

The resulting Booster specification is suitable for generating a functional system capable of entering and storing data corresponding to the original model. In many cases, this may be sufficient, in particular with the addition of a bespoke user interface to enact particular workflows on top of the generated methods. However, the specification may also be used as the basis for a more elaborate Booster system, by using the Booster functionality for importing and overriding through inheritance. In this way, constraints and business rules may be added, along with additional methods and attributes, to provide a richer implementation, but by the nature of inheritance in Booster, still compliant with the original data model.

Model Catalogue information through the Booster interfaces

Once a Booster system has been used to capture and store data, it is important that this information can be reused. In many cases, this may require an understanding of the context of collection. This is especially helpful where data are to be combined from multiple systems, and it is important that only data items with similar definitions are combined. In order to allow such contexts to be available alongside the data, Semantic Booster needs to include functionality to allow integration with the data stored within the Model Catalogue.

It is vital that the Model Catalogue remains as the single source for metadata, rather than copies of the data being moved into the Booster-generated



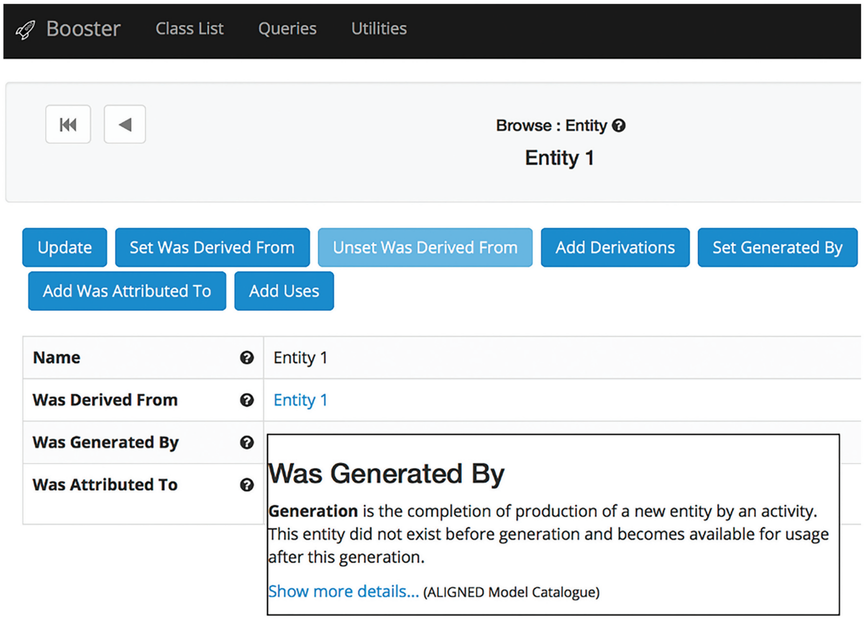
```
class Entity {
  attributes
    name (ID) : String
    wasDerivedFrom : [ Entity . derivations ]
    derivations : SET( Entity . wasDerivedFrom )
    wasGeneratedBy : [ Activity . derivesEntities ]
    wasAttributedTo : SET( Agent . attributedEntities )
    uses : SET( Activity . used )

  methods
    Create { e! : Entity' & e!.name = name? }
    Update { this.name = name? }
                  /: Entity' }
                 om { wasDerivedFrom' = entity? }
    UnsetWasDerivedFrom { wasDerivedFrom' = null }
    AddDerivations { entity? : derivations' }
    RemoveDerivations { entity? : derivations' }
    SetGeneratedBy { wasGeneratedBy' = entity? }
    UnsetGeneratedBy { wasGeneratedBy' = null }
    AddWasAttributedTo { agent? : wasAttributedTo' }
    RemoveWasAttributedTo { agent? : wasAttributedTo' }
    AddUses { activity? : uses' }
    RemoveUsers { true }

}
```

Figure 5.15 Excerpt from the Booster system generated from Prov-DM Core.

system. Although most parts of the metadata are frozen on publication (and subsequent implementation), other metadata components such as comments and links may be added after the system has been deployed. The approach taken has been to create links between components in the Booster specification such that the metadata can be seamlessly retrieved.

Metadata concerning the Model Catalogue itself is placed in a table alongside the Booster data: the Web URL to locate the catalogue, the version number and name of the catalogue, along with any lists of any user credentials required to access private models. Each Booster specification component corresponding to an element from the Model Catalogue is stored with a GUID, a link to the relevant catalogue metadata, and the credentials required for access. This includes the system itself, every class and attribute, datatypes, and enumeration values.

This model catalogue data is stored in the database alongside the data, but accessed through a bespoke set of SQL stored procedures. These are exposed through the API layer so that external applications can access them.



Name	Entity 1
Was Derived From	Entity 1
Was Generated By	
Was Attributed To	

Was Generated By

Generation is the completion of production of a new entity by an activity. This entity did not exist before generation and becomes available for usage after this generation.

Show more details... (ALIGNED Model Catalogue)

Figure 5.16 Model Catalogue information in the Booster interface.

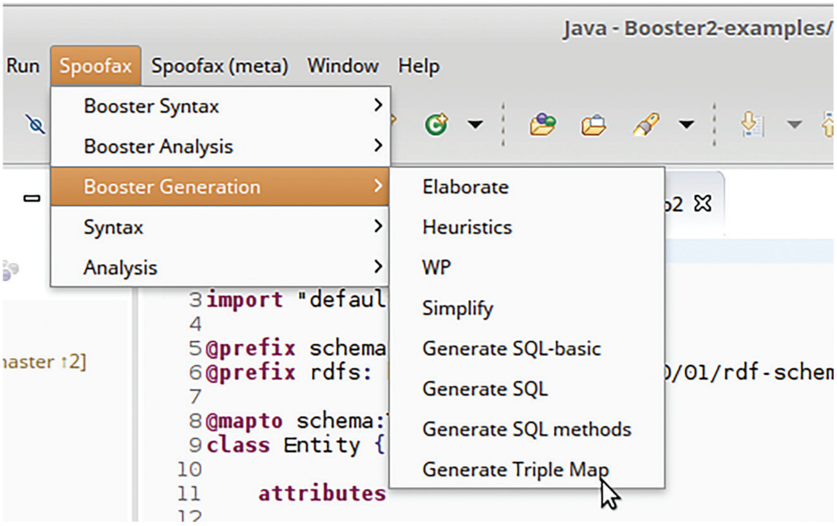
The data are also propagated through the Booster Web interface, so that users of the data can access the appropriate metadata. A new REST API call has been added to the Model Catalogue, displaying a snippet of HTML with a link for more information. This is shown in Figure 5.16.

Semantic Booster: Booster data as triples

A key requirement of Semantic Booster is that system data must be accessible as RDF Triples. Data systems generated by the original Booster tool have employed a standard relational schema for data persistence, and standard stored procedures for operations on data. In Semantic Booster, a new target had been developed: “Generate Triple Map” is to be used alongside the standard relational generation (Figure 5.17).

This additional transformation generates a mapping specified in the W3 standard R2RML language.[3] This mapping reflects the Booster specification, and includes standard simplifications for Booster constructs of associations and inheritance, creating a lightweight wrapper for the Booster database schema. The R2RML standard has made this extension tractable:

[3]https://www.w3.org/TR/r2rml



```
import "defaul
@prefix schema
@prefix rdfs:                  )/01/rdf-schen
@mapto schema:
class Entity {
    attributes
```

Figure 5.17 Semantic Booster – generation menu in the Eclipse IDE.

the functionality has been enabled using an additional transformation, rather than a ground-up rewrite of the pre-existing transformations.

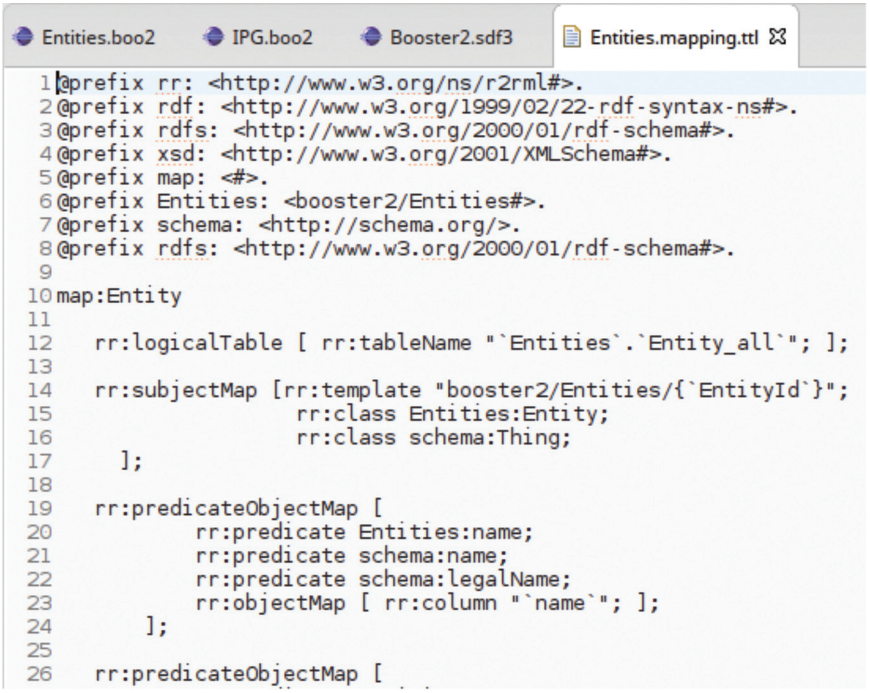
The R2RML schema can be used to present a 'live' view of the data or to extract the whole dataset as a data 'dump'. Furthermore, the data are opened to a range of existing 'semantic Web' tools which can deal with the data in this triple format.

In the live view, any changes made by Booster operations are automatically available in the triples. No complex update or synchronisation is required as the triple-view of the data is derived directly from the relational data, and existing Booster functionality is not affected.

User-Specified semantic mappings

Typically, RDF triples have associated-type information in a semantic schema, typically via an RDF schema or an OWL ontology. The schema gives users a grounding by which to explore, classify, and query the objects, data and relationships in a dataset. Typical queries over a semantic dataset might find all elements with a particular semantic type, relationship type or a pattern of types, using the SPARQL query language. The schema and types of a data system are defined on a per-system basis by data engineers.

For a system such as Semantic Booster to be useful to data engineers, the types and schemas of the system must be customisable. In an R2RML mapping, each element must have a type, and a default set of types is created by



Entities.boo2 IPG.boo2 Booster2.sdf3 Entities.mapping.ttl

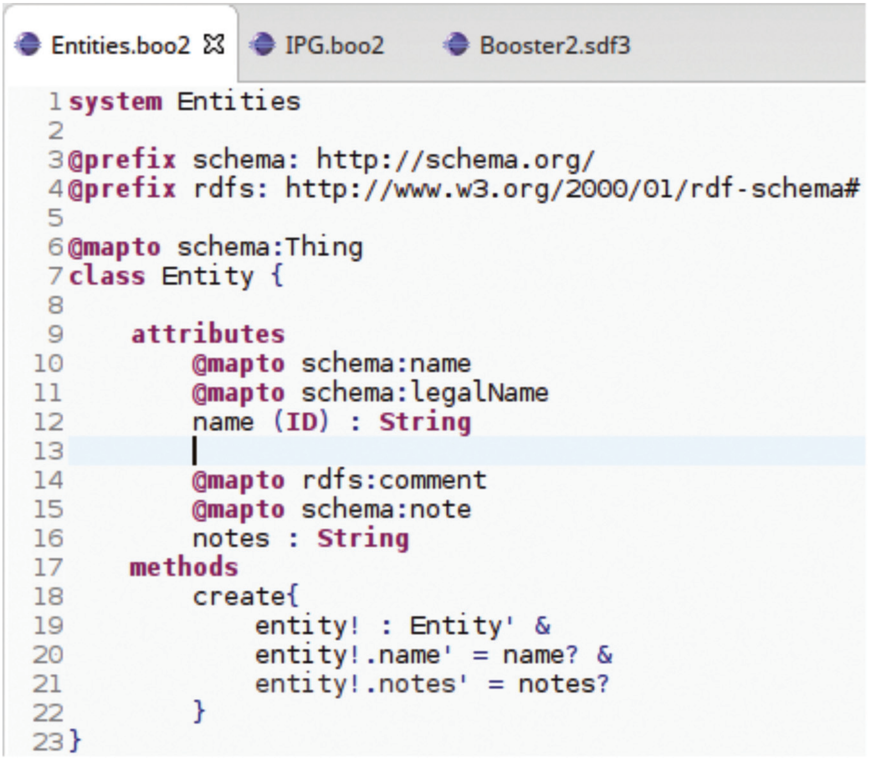
```
@prefix rr: <http://www.w3.org/ns/r2rml#>.
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>.
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#>.
@prefix xsd: <http://www.w3.org/2001/XMLSchema#>.
@prefix map: <#>.
@prefix Entities: <booster2/Entities#>.
@prefix schema: <http://schema.org/>.
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#>.

map:Entity

   rr:logicalTable [ rr:tableName "`Entities`.`Entity_all`"; ];

   rr:subjectMap [rr:template "booster2/Entities/{`EntityId`}";
                  rr:class Entities:Entity;
                  rr:class schema:Thing;
     ];

   rr:predicateObjectMap [
          rr:predicate Entities:name;
          rr:predicate schema:name;
          rr:predicate schema:legalName;
          rr:objectMap [ rr:column "`name`"; ];
      ];

   rr:predicateObjectMap [
```

Figure 5.18 Semantic Booster – generated R2RML file.

the “Generate Triple Map” target. A sample generated R2RML file is shown in Figure 5.18. In Semantic Booster, the types of each element can also be specified via annotations in the Booster specification. An annotation syntax has been added to the standard Booster notation, as shown in Figure 5.19, lines 6, 10–11 and 14–15. Annotations on each class, attribute and relationship of a Booster specification can be specified by the engineer. The generated R2RML mappings, and in turn the data, will hold the type information from the annotation for those data elements. As with semantic data, URIs are used for types and a prefix mechanism is provided so that URIs may be shortened in a specification, shown in Figure 5.19 lines 3–4. This annotation mechanism is itself extensible, so that enhancements to the R2RML specification may be further customised to enhance the relational-triple mapping

Integration with the Model Catalogue

In order that the Booster tool can be applied in the context of large-scale data engineering, it is important that data managed by a Booster system can be adequately understood, re-purposed, and combined with other data sources.

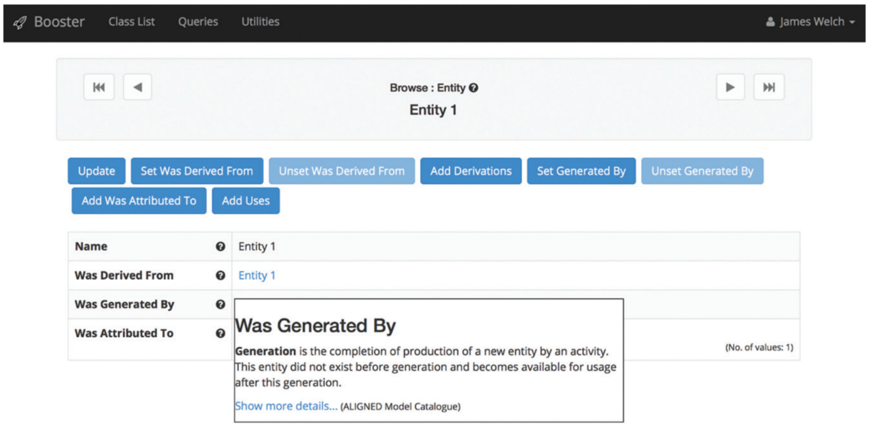


```
system Entities

@prefix schema: http://schema.org/
@prefix rdfs: http://www.w3.org/2000/01/rdf-schema#

@mapto schema:Thing
class Entity {

    attributes
        @mapto schema:name
        @mapto schema:legalName
        name (ID) : String

        @mapto rdfs:comment
        @mapto schema:note
        notes : String
    methods
        create{
            entity! : Entity' &
            entity!.name' = name? &
            entity!.notes' = notes?
        }
}
```

Figure 5.19 Booster specification with semantic annotations.

To facilitate this, Booster is integrated with the Model Catalogue. A typical software development process using the combined toolset might start with the import of a data standard, or the metadata for an existing dataset, into the Model Catalogue. These descriptions can be reused and extended in the definition of a new data model, and exported as a new Booster specification. Figure 5.20 shows Booster with Model Catalogue semantic annotations. The Booster specification contains hooks back to the original definitions, such that the generated system can store links back to the definitions, and provide them to users at the data-entry interface. This can improve the quality of data entry, and ensure consistency across multiple systems using equivalent definitions. Finally, an existing Booster specification may be re-imported into the catalogue and annotated for further reuse.

The round-tripping provided by this new functionality allows the Model Catalogue to be used as an enhanced IDE for model-driven development. The



Name	Entity 1
Was Derived From	Entity 1
Was Generated By	
Was Attributed To	

Was Generated By

Generation is the completion of production of a new entity by an activity. This entity did not exist before generation and becomes available for usage after this generation.

Figure 5.20 Booster user interface showing semantic annotations from the Model Catalogue.

extended reuse of data components promotes greater reuse of data, and can lead to improvements to the quality of data, and the adherence to standard definitions.

Eclipse Booster IDE

During the second phase of the ALIGNED project, the Booster tool has been upgraded to use the latest versions of the Spoofax language engineering workbench – from version 1.x to the newly released v2.0. Spoofax provides rich editing support for Booster specifications inside Eclipse and includes syntax highlighting, specification outline views and type checking. The upgrade brings amongst other changes: improved compilation and transformation times, simpler project layouts and support for running transformations without Eclipse, via the Sunshine tool. Running Booster transformations outside of Eclipse is a key component of the Semantic Booster Development and Deployment Kit (sBDK) discussed in Section 4. The enhanced underlying environment has allowed the development of more powerful editors and syntax checking, and allows the Booster tool itself to be more easily updated to meet new requirements or support further functionality.

Semantic Booster development and deployment kit: sBDK

In order to support developers and end users of Semantic Booster systems, a number of features have been created and combined to form a Semantic Booster Development and Deployment Kit (sBDK).

Semantic Booster Docker Container

Creating a useable system using Semantic Booster and Eclipse can be operationally challenging, requiring many non-trivial steps to setup a workable system. The complexity has only increased since the additional developments for the ALIGNED project partners. Several interdependent software packages must be installed and configured correctly for a Semantic Booster system to work as intended.

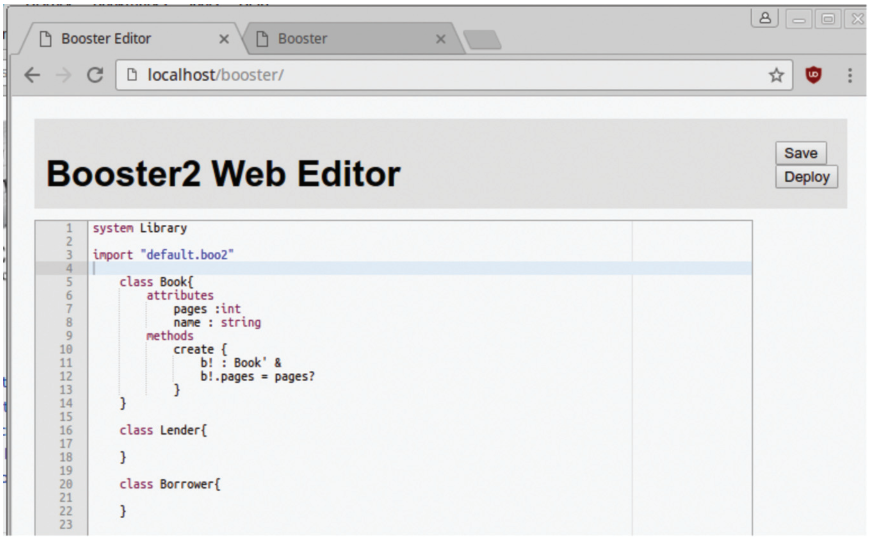
To simplify and streamline the creation and maintenance of data-intensive systems with Semantic Booster, a Docker container has been developed. Docker[4] is a framework for building applications as containers, making them portable and easily deployed without complicated configuration or system set-up. Docker is used to automate deployment of all software needed for a Semantic Booster system whilst isolating the software from the host operating system. The container takes a single parameter: a Booster specification that the user has previously created: either using the tool support provided in the Semantic Booster Eclipse editor, or exported as an artefact from the Model Catalogue. The Docker container will execute a number of scripts to configure and create tools to access and manage the system. A Web-based Booster editor is provided, along with the Booster Native Data exploration tool and a number of semantic data exploration tools.

Additional parameters can be provided to the Docker initialisation to perform additional system configuration, including the pre-population of the generated Semantic Booster system and the ability to persist, backup and restore the database between different runtimes of the system: features vital to the smooth operation of the system within the ALIGNED use case. Parameters have been added to the container to regenerate only the methods of a Semantic Booster system. The Semantic Booster Docker container is available on GitHub and is in use by the Wolters Kluwer IPG use case.

Semantic Booster web editor

A new feature for Semantic Booster is the Web-based editor for Semantic Booster specifications. The editor is less sophisticated than the existing Eclipse-based tooling and therefore is expected to only be used for iteration to an existing specification or system. The Semantic Booster Web editor is shown in Figure 5.21. The editor also provides automatic generation and redeployment functionality, such that a system previously created by the Docker

[4]http://docker.com



```
system Library

import "default.boo2"

    class Book{
        attributes
            pages :int
            name : string
        methods
            create {
                b! : Book' &
                b!.pages = pages?
            }
    }

    class Lender{

    }

    class Borrower{

    }

```

Figure 5.21 Semantic Booster Web-based editor.

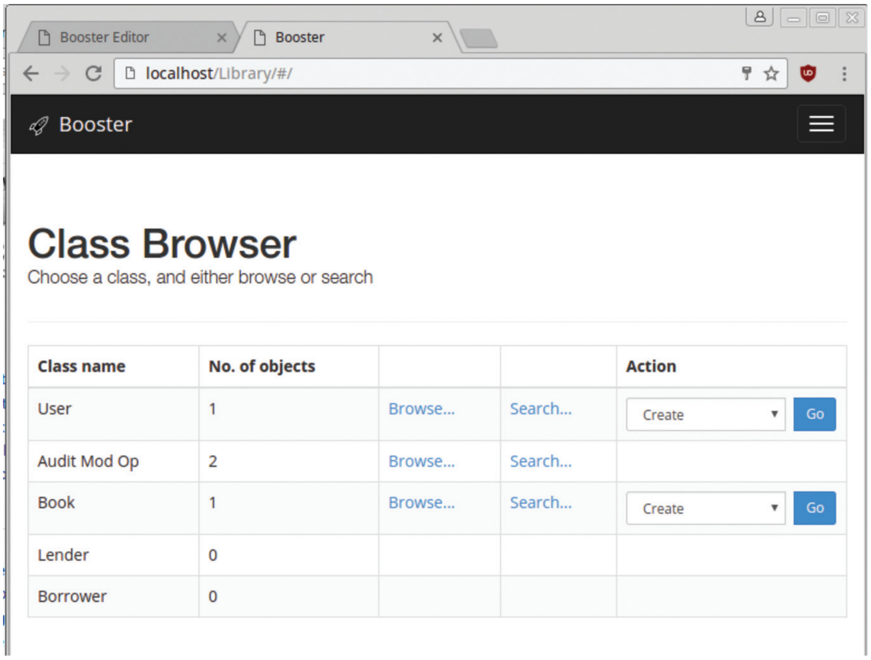
container can be recreated. This uses the Sunshine mechanism of the Spoofax language, to execute Booster transformations outside the Eclipse IDE.

Default Booster web explorer

As well as the Web editor, the Docker container also deploys the default Booster Web Data Explorer (Figure 5.22) from where the data in a Booster system can be explorer and operations invoked. The data explorer can be configured to link to a model catalogue for providing metadata annotations, and is configured to expose information available through the semantic mappings.

Semantic data exploration: D2RQ, Snorql

The D2RQ semantic Web data exploration tool and the Snorql SPARQL query tool (illustrated in Figure 5.23) are pre-configured in the Docker container. These tools use the R2RML mapping created by Semantic Booster to present the data in a Booster system as triples. This allows for interoperability between the software engineering and data engineering worlds. D2RQ provides a data browser – an RDF equivalent to the default Booster data explorer. Snorql provides a standard SPARQL endpoint to the RDF triples: as a W3C standard for querying semantic Web data, this allows a variety of



Class Browser

Choose a class, and either browse or search

Class name	No. of objects			Action
User	1	Browse...	Search...	Create ▾ Go
Audit Mod Op	2	Browse...	Search...	
Book	1	Browse...	Search...	Create ▾ Go
Lender	0			
Borrower	0			

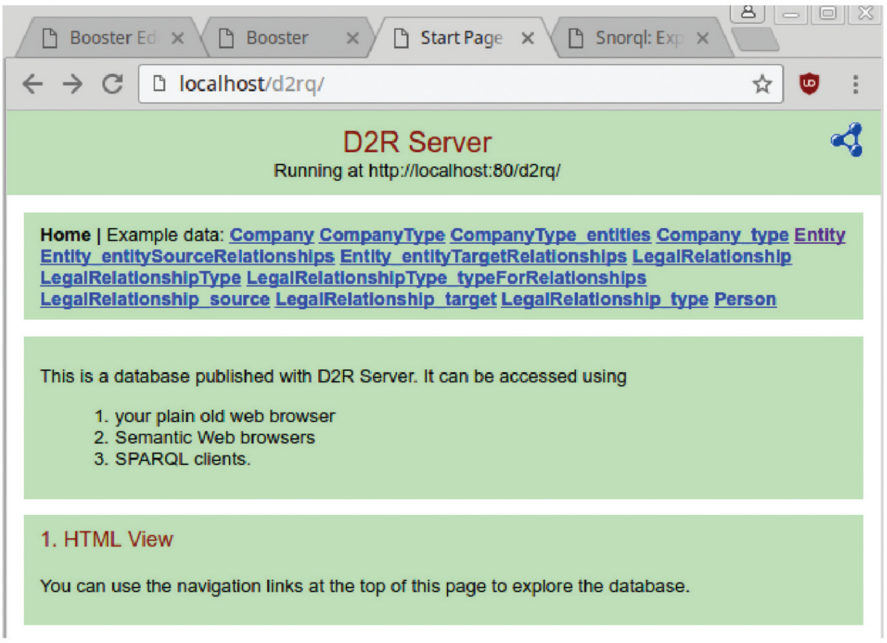
Figure 5.22 Default Booster data explorer.

standard semantic Web tools to interact with the data stored within a Booster system.

The mapping generated is specific to the Booster specification and considers the semantic mapping annotations defined for classes and attributes. In addition, some simplifications are made to better represent Booster inheritance and association.

Semantic data visualisation tool: D3Sparql

A second semantic Web-based data exploration tool has been included in the Semantic Booster Docker container: D3Sparql. Semantic Web data, as triples forms a conceptual graph of data. D3Sparql visualises the results of a SPARQL query as an interactive graph, as shown in Figure 5.8. End users can make use of the tool to interact with the RDF data: domain experts may make use of pre-defined SPARQL queries to provide dashboard-style views or high-level summary metadata; technical users may make use of the built-in Snorql tool to develop complex queries for searching or investigation. Figure 5.24 shows D3Sparql in action.



D2R Server
Running at http://localhost:80/d2rq/

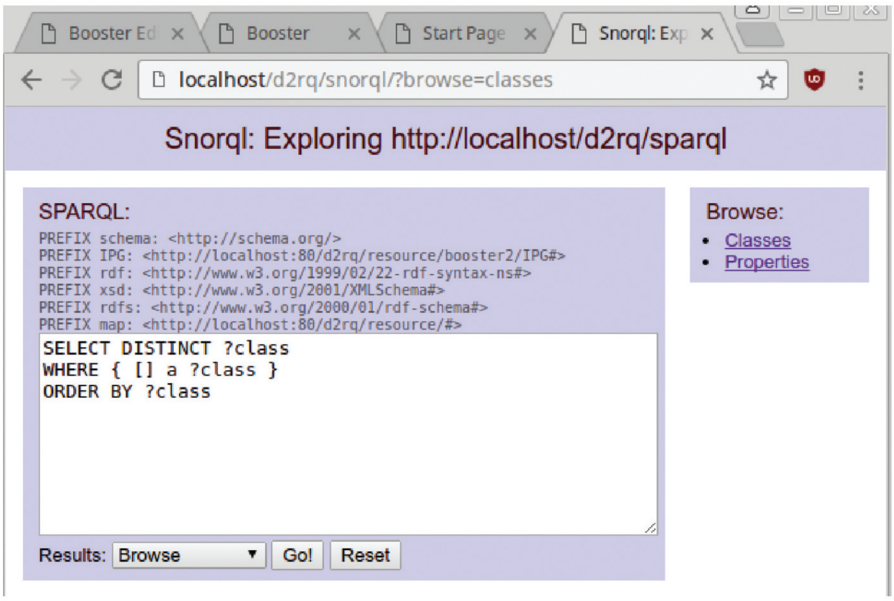
Home | Example data: Company CompanyType CompanyType_entities Company_type Entity Entity_entitySourceRelationships Entity_entityTargetRelationships LegalRelationship LegalRelationshipType LegalRelationshipType_typeForRelationships LegalRelationship_source LegalRelationship_target LegalRelationship_type Person

This is a database published with D2R Server. It can be accessed using

1. your plain old web browser
2. Semantic Web browsers
3. SPARQL clients.

1. HTML View

You can use the navigation links at the top of this page to explore the database.



Snorql: Exploring http://localhost/d2rq/sparql

SPARQL:

```
PREFIX schema: <http://schema.org/>
PREFIX IPG: <http://localhost:80/d2rq/resource/booster2/IPG#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX map: <http://localhost:80/d2rq/resource/#>
SELECT DISTINCT ?class
WHERE { [] a ?class }
ORDER BY ?class
```

Results: Browse | Go! | Reset

Browse:
- Classes
- Properties

Figure 5.23 D2RQ and SNORQL for exploration of Semantic Booster data.

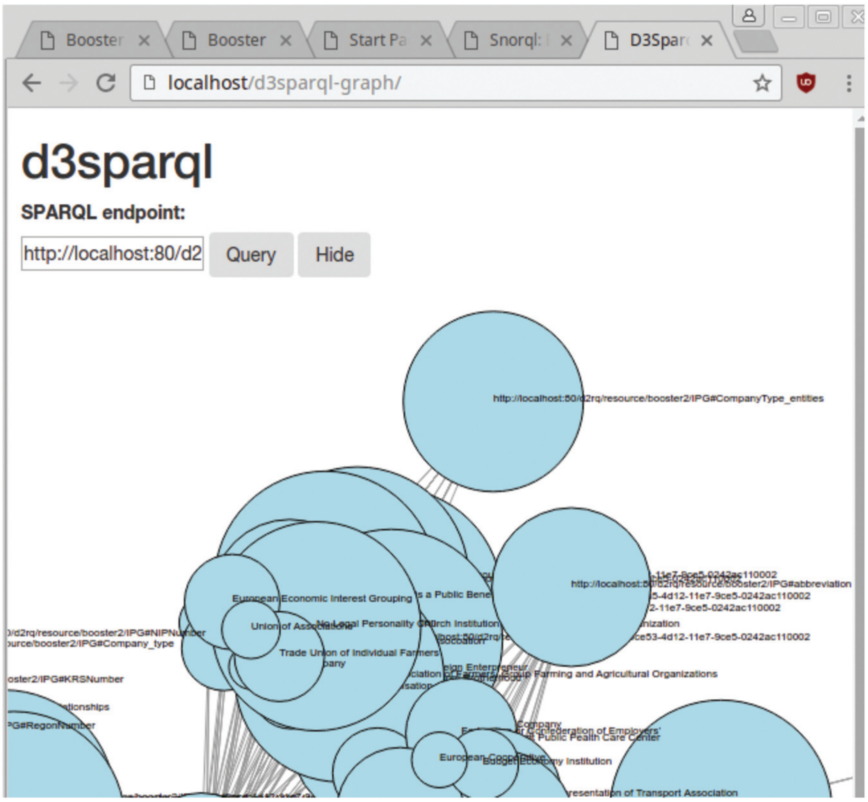

Booster
Booster
Start Pa
Snorql:
D3Spar
localhost/d3sparql-graph/
d3sparql
SPARQL endpoint:
http://localhost:80/d2
Query
Hide
http://localhost:80/d2rq/resource/booster2/IPG#CompanyType_entities
European Economic Interest Grouping
Trade Union of Individual Farmers
Farming and Agricultural Organizations
European Cooperative
Public Health Care Center
resentation of Transport Association

Figure 5.24 d3sparql for visual exploration of Semantic Booster data.

Semantic data validation tool: RDFUnit

The RDFUnit tool has been integrated into the Semantic Booster Docker Container, as shown in Figure 5.25. RDFUnit is used for the validation of data and allows end users to validate RDF triple data against a suite of specifications of data quality constraints, written in SPARQL. The tool checks that the constraints hold post-hoc, in contrast to the correct by construction approach of Semantic Booster. RDFUnit is configured to use the D2RQ endpoint, allowing constraints to be written against up-to-date Booster data.

5.2 RDFUnit

RDFUnit is an RDF validation framework inspired by test-driven software development. The test case definition language of RDFUnit is SPARQL,

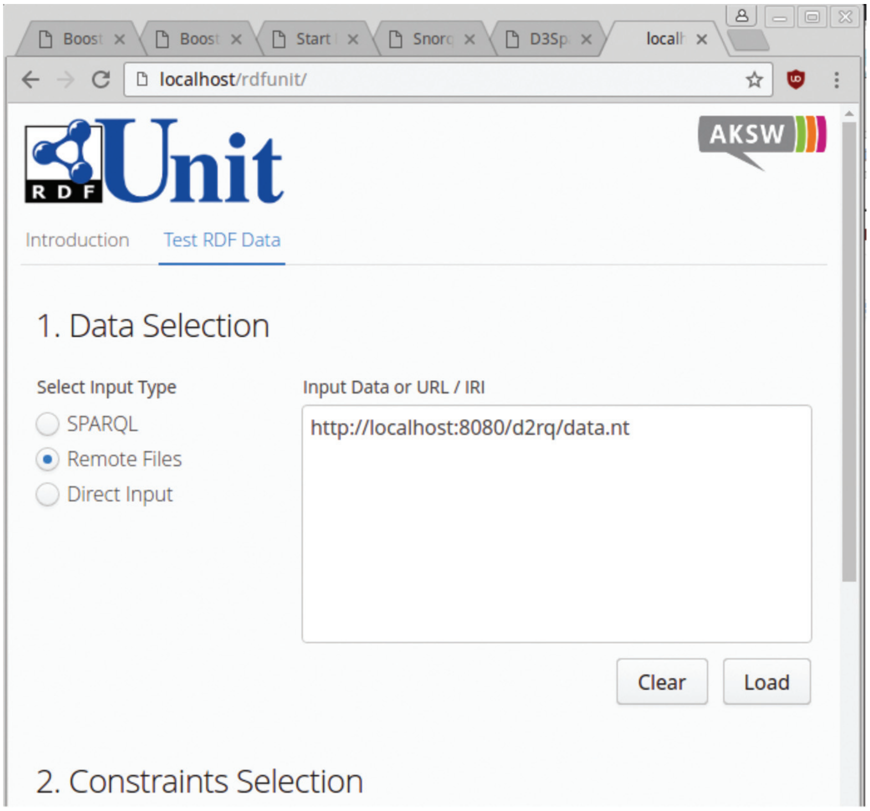


Figure 5.25 RDFUnit Web interface.

which is convenient to directly query for identifying violations. For rapid test case instantiation, a pattern-based SPARQL-Template engine is supported where the user can easily bind variables into patterns. RDFUnit has a Test Auto Generator (TAG) component. TAG searches for schema information and automatically instantiates new test cases. Schema information can be in the form of RDFS or OWL axioms that RDFUnit translates into SPARQL under Closed World Assumption (CWA) and Unique Name Assumption (UNA). Other schema languages such as SHACL[5], IBM Resource Shapes[6] or Description Set Profiles[7] are also supported. For a full overview of

[5]https://www.w3.org/TR/shacl/
[6]https://www.w3.org/Submission/shapes/
[7]http://dublincore.org/documents/dc-dsp/

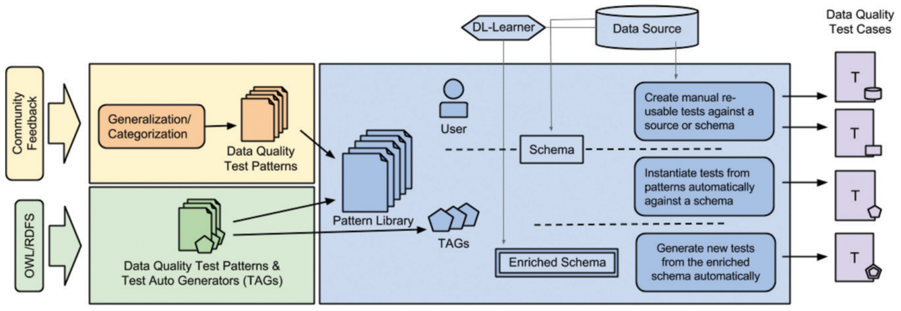

Community Feedback
OWL/RDFS
Generalization/ Categorization
Data Quality Test Patterns
Data Quality Test Patterns & Test Auto Generators (TAGs)
DL-Learner
Data Source
Data Quality Test Cases
User
Schema
Pattern Library
TAGs
Enriched Schema
Create manual re-usable tests against a source or schema
Instantiate tests from patterns automatically against a schema
Generate new tests from the enriched schema automatically
T

Figure 5.26 RDFUnit architecture.

RDFUnit's data testing and verification capabilities see Kontokostas et al. Figure 5.26 shows the RDFUnit architecture.[8]

5.2.1 RDFUnit Integration

The following subsections describe three ways RDFUnit-based data testing and verification can be integrated into software engineering workflows. The JUnit runner with annotations provides a very easy and well integrated option but does not give room for flexibility beyond testing an input dataset to a fixed schema. The JUnit XML Report gives room for greater flexibility by utilising the complete RDFUnit command line options. Finally, the custom Maven-based integration gives software engineers a way to fine-tune the way they want to automate their data testing and verification options.

JUnit Runner integration with Java annotations

JUnit allows other testing frameworks to extend JUnit with custom Runners[9] tailored for specific testing. A custom JUnit Runner was implemented, RdfUnitJunitRunner[10], which can be used to define JUnit tests for validating RDF datasets against a schema, by adding Java annotations to a JUnit test.

[8]D. Kontokostas, P. Westphal, S. Auer, S. Hellmann, J. Lehmann, R. Cornelissen, and A. Zaveri. Test-driven Evaluation of Linked Data Quality, Proc. 23rd International Conference on World Wide Web, pp. 747–758, DOI 10.1145/2566486.2568002, 2014.

[9]https://github.com/junit-team/junit4/wiki/test-runners

[10]https://github.com/AKSW/RDFUnit/tree/master/rdfunit-junit

An example RDFUnit/JUnit test is the following:

```
@RunWith(RdfUnitJunitRunner.class)
@Schema(uri = "schema.ttl")
public static class TestRunner {
  @TestInput
  public RDFReader getInputData() {
                       return new RdfModelReader(
                       RdfReaderFactory.createResourceReader(
                               "/inputmodels/data.ttl" ).read()); }
}
```

Where data.ttl is an RDF data file (using the @TestInput annotation) tested by a JUnit test against schema.ttl (using the @Schema annotation).

For every automatically generated RDFUnit test, a separate JUnit test is generated that validates the input dataset for a specific violation. The reporting of validation errors is integrated with JUnit reports, thus providing the means to display them through IDEs like IntelliJ or with build tools like Maven.

5.2.1.1 JUnit XML report-based integration

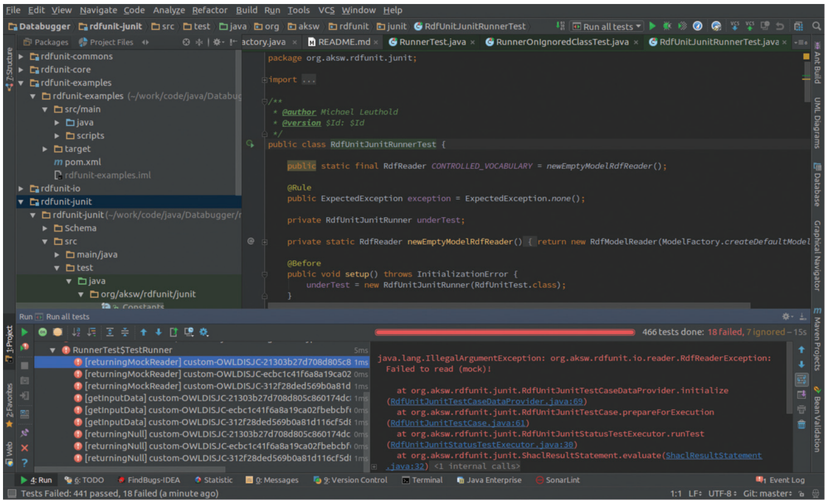
JUnit uses a specific XML schema[11] to communicate the test results to IDEs or build tools. For cases when defining an RDFUnit/JUnit test is not an option (i.e., the files are not accessible from the build system with Java code), the RDFUnit results can be converted to the JUnit XML Schema. In these cases, developers can run RDFUnit as a command line tool or through custom code, expecting validation results in the JUnit XML Schema. Build systems, such as Bamboo, can then be configured to look at specific locations for such XML files and report the RDFUnit validation results with the existing unit test error reporting tools.[12] Figure 5.27 shows the RDFUnit report in the IntelliJ IDE. Figure 5.28 shows the report in the Bamboo build system.

5.2.1.2 Custom apache maven-based integration

When the input data or schema graph are not simple input files, but generated through custom procedures, the aforementioned methods are not easy to apply. For those cases, RDFUnit can be used as a Java library, fine-tuned for custom input or more sophisticated Jenkin reports. This was the case with the JURION demo, where RDFUnit was used to validate if the output of specific XSLT scripts adhered to the JURION Schema. All results were archived to

[11]https://svn.jenkins-ci.org/trunk/hudson/dtkit/dtkit-format/dtkit-junit-model/src/main/resources/com/thalesgroup/dtkit/junit/model/xsd/junit-4.xsd

[12]https://github.com/AKSW/RDFUnit/wiki/Using-RDFunit-with-Bamboo


File Edit View Navigate Code Analyze Refactor Build Run Tools VCS Window Help
Databugger rdfunit-junit src test java org aksw rdfunit junit RdfUnitJunitRunnerTest
Run all tests
actory.java README.md RunnerTest.java RunnerOnIgnoredClassTest.java RdfUnitJunitRunnerTest.java
Packages Project Files
rdfunit-commons
rdfunit-core
rdfunit-examples
rdfunit-examples (~/work/code/java/Databug
src/main
java
scripts
target
pom.xml
rdfunit-examples.iml
rdfunit-io
rdfunit-junit
rdfunit-junit (~/work/code/java/Databugger/r
Schema
src
main/java
test
java
org/aksw/rdfunit/junit
Constants
Run Run all tests
4: Run 6: TODO FindBugs-IDEA Statistic 0: Messages 9: Version Control Terminal Java Enterprise SonarLint Event Log
Ant Build UML Diagrams Database Graphical Navigator Maven Projects Bean Validation
1: Project 2: Favorites Web 7: Structure

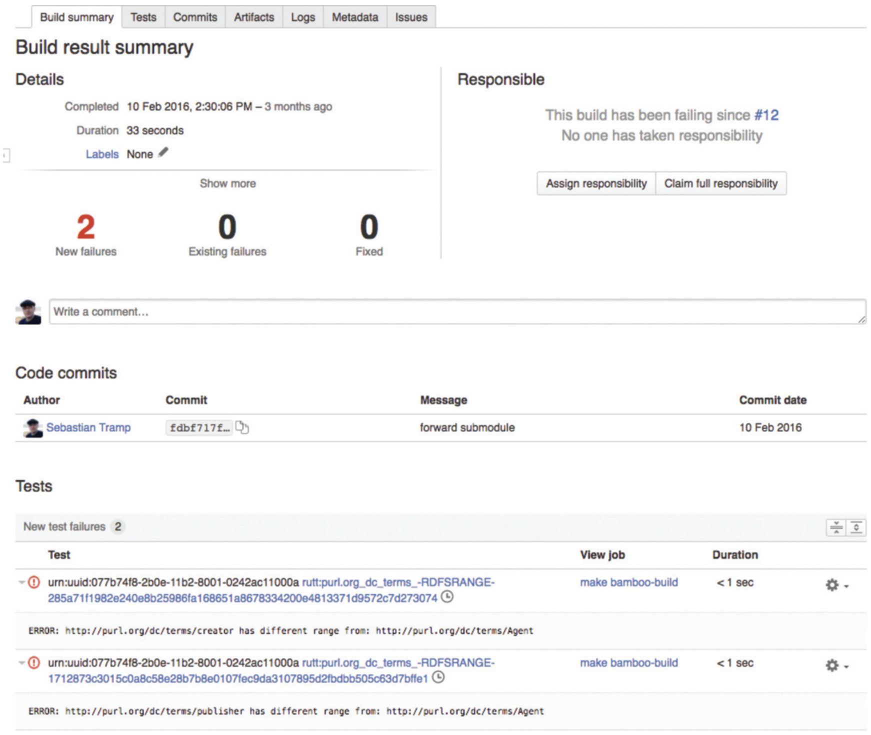
```
package org.aksw.rdfunit.junit;

import ...

/**
 * @author Michael Leuthold
 * @version $Id: $Id
 */
public class RdfUnitJunitRunnerTest {

    public static final RdfReader CONTROLLED_VOCABULARY = newEmptyModelRdfReader();

    @Rule
    public ExpectedException exception = ExpectedException.none();

    private RdfUnitJunitRunner underTest;

    private static RdfReader newEmptyModelRdfReader() { return new RdfModelReader(ModelFactory.createDefaultModel

    @Before
    public void setup() throws InitializationError {
        underTest = new RdfUnitJunitRunner(RdfUnitTest.class);
    }
```

```
466 tests done: 18 failed, 7 ignored – 15s

RunnerTest$TestRunner 5ms
[returningMockReader] custom-OWLDISJC-21303b27d708d805c8 1ms
[returningMockReader] custom-OWLDISJC-ecbc1c41f6a8a19ca02 0ms
[returningMockReader] custom-OWLDISJC-312f28ded569b0a81d 1ms
[getInputData] custom-OWLDISJC-21303b27d708d805c860174dc 1ms
[getInputData] custom-OWLDISJC-ecbc1c41f6a8a19ca02fbebcbf 0ms
[getInputData] custom-OWLDISJC-312f28ded569b0a81d116cf5d 1ms
[returningNull] custom-OWLDISJC-21303b27d708d805c860174dc 0ms
[returningNull] custom-OWLDISJC-ecbc1c41f6a8a19ca02fbebcbf 0ms
[returningNull] custom-OWLDISJC-312f28ded569b0a81d116cf5d 1ms
```

```
java.lang.IllegalArgumentException: org.aksw.rdfunit.io.reader.RdfReaderException:
  Failed to read (mock)!

   at org.aksw.rdfunit.junit.RdfUnitJunitTestCaseDataProvider.initialize
(RdfUnitJunitTestCaseDataProvider.java:69)
   at org.aksw.rdfunit.junit.RdfUnitJunitTestCase.prepareForExecution
(RdfUnitJunitTestCase.java:61)
   at org.aksw.rdfunit.junit.RdfUnitJunitStatusTestExecutor.runTest
(RdfUnitJunitStatusTestExecutor.java:30)
   at org.aksw.rdfunit.junit.ShaclResultStatement.evaluate(ShaclResultStatement
.java:32) <1 internal calls>
```

```
Tests Failed: 441 passed, 18 failed (a minute ago)          1:1  LF:  UTF-8:  Git: master:
```

Figure 5.27 RDFUnit report from the IntelliJ IDE.



Build result summary

Details

Completed 10 Feb 2016, 2:30:06 PM – 3 months ago
Duration 33 seconds
Labels None

2 New failures
0 Existing failures
0 Fixed

Responsible

This build has been failing since #12
No one has taken responsibility

Code commits

Author	Commit	Message	Commit date
Sebastian Tramp	fdbf717f...	forward submodule	10 Feb 2016

Tests

New test failures 2

Test	View job	Duration
urn:uuid:077b74f8-2b0e-11b2-8001-0242ac11000a rutt:purl.org_dc_terms_-RDFSRANGE-285a71f1982e240e8b25986fa168651a8678334200e4813371d9572c7d273074	make bamboo-build	< 1 sec
ERROR: http://purl.org/dc/terms/creator has different range from: http://purl.org/dc/terms/Agent		
urn:uuid:077b74f8-2b0e-11b2-8001-0242ac11000a rutt:purl.org_dc_terms_-RDFSRANGE-1712873c3015c0a8c58e28b7b8e0107fec9da3107895d2fbdbb505c63d7bffe1	make bamboo-build	< 1 sec
ERROR: http://purl.org/dc/terms/publisher has different range from: http://purl.org/dc/terms/Agent		

Figure 5.28 Example Bamboo overview from an RDFUnit JUnit XML report.

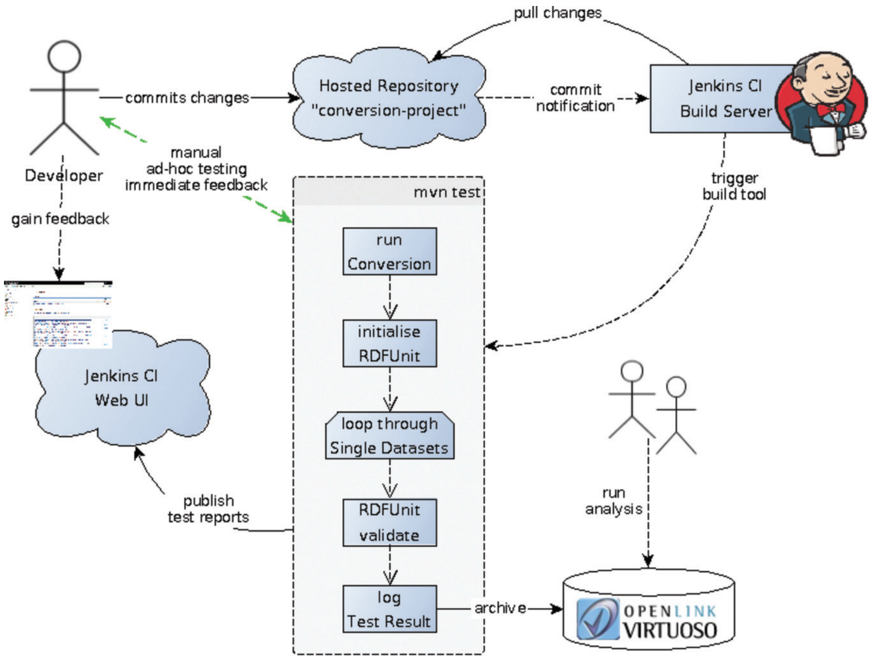


Figure 5.29 Custom JUnit integration with RDFUnit as a library for JURION Use Case in ALIGNED.

a triple store for post-processing analysis (see Image 3). A research paper was published that describes this use case in detail.[13] Figure 5.29 shows the integration of RDFUnit in JURION.

5.2.1.3 The shapes constraint language (SHACL)

The Shapes Constraint Language is a language to validate RDF graphs against a set of constraints. These constraints are formalised as shapes and other constructs expressed in the form of an RDF graph. The language features and approaches occurring in the current specification of SHACL were in part inspired by the SPIN[14] and Shape Expressions (ShEx).[15] The current

[13]Dimitris Kontokostas, Christian Mader, Michael Leuthold, Christian Dirschl, Katja Eck, Jens Lehmann and Sebastian Hellmann. Semantically Enhanced Quality Assurance in the JURION Business Use Case. ESWC 2016, Crete. Available at: http://link.springer.com/chapter/10.1007/978-3-319-34129-3_40

[14]http://spinrdf.org/

[15]E. Prud'hommeaux, J. E. Labra Gayo and H. Solbrig. Shape expressions: an RDF validation and transformation language, 10th International Conference on Semantic Systems, pp. 32–40, 2014.

revision of the specification for SHACL is published by the W3C[16] with complementary material available in a GitHub repository.[17]

SHACL Core defines frequently needed features to formulate common constraints for RDF graphs. SHACL Core Constraints are defined by parameterising Constraint Components that are templates for checks for a specific required property of an RDF nodes (e.g., unique occurrence of a property value associated with a specific property, for instance only one foaf:age value for a given foaf:Person). One or several of such constraints are associated with target RDF nodes to validate against in a SHACL Shape. SHACL Shapes are expressed as RDF resources and aggregated in a Shapes Graph. An RDF graph to be checked for conformance against a Shapes Graph (the Data Graph) is provided to a Validation Engine that produces a Validation Report. The Validation Report states whether the Data Graph conforms to the Shapes Graph, listing violations of individual RDF nodes against shapes detected during the validation process in case of non-conformance.

5.2.1.4 Comparison of SHACL to schema definition using RDFUnit test patterns

The original declarative approach to create data tests with RDFUnit (i.e., formulating data constraints without composing SPARQL queries or SPARQL query fragments directly) involves selecting and parameterising an RDFUnit Test Pattern. These test patterns bear several conceptual and functional similarities to both SHACL Shapes and SHACL Constraint Components. Test patterns also define parameters to be set to transform a test pattern into a concrete test case. In contrast to SHACL, the parameters of an RDFUnit test pattern do not only specify the expected constraints for applicable RDF nodes but also often influence the sets of RDF nodes the test is applied to. The clearer separation of these concerns in SHACL increases modularity and thus allows more flexible reuse of parts of shape definitions. As an additional advantage, SHACL Shapes can be defined recursively, i.e., more complex shapes can be composed by combining simpler shapes. For example, a shape S1 can define that all its values nodes for a property P must conform to a shape S2. Also, multiple individual restrictions can be combined to a conjunction (target nodes must conform to S1 and S2), different acceptable shape alternatives can be expressed by disjunction (target nodes must conform to S1 or S2) and shape constraints can be inverted/negated (target nodes must not

[16]https://www.w3.org/TR/shacl/

[17]https://github.com/w3c/data-shapes

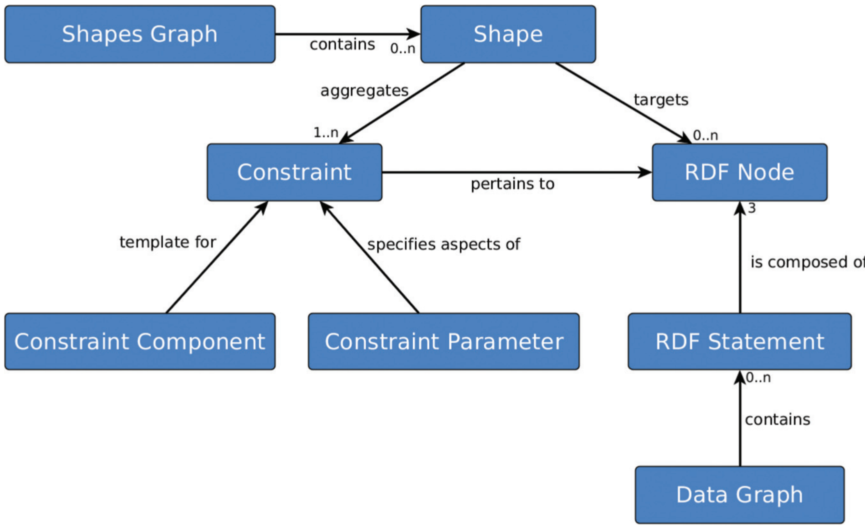


Figure 5.30 Overview for Fundamental Concepts of SHACL.

conform to S1). SHACL provides for a range of ways to define the focus nodes for a shape, i.e., class membership, explicit nodes, subjects and objects of a predicate. In the current set of RDFUnit's test patterns, targeted notes are predominantly only scoped by class membership or property appearance. In general, however, scoping in RDFUnit is more flexible as it is defined directly in SPARQL and there are no limitations. For example, there is no way to define in SHACL the constraint that all entities must have a label. Figure 5.30 shows an overview of the fundamental concepts of SHACL.

5.2.1.5 Comparison of SHACL to auto-generated RDFUnit tests from RDFS/OWL axioms

In addition to a direct instantiation of test patterns in manual test suites, RDFUnit already offers capabilities to create data test suites automatically from RDFS and OWL axioms pertaining to the vocabulary used in the RDF data to be tested (Test Auto Generators). This approach enables utilisation of modelling efforts of RDF vocabulary providers that specified aspects of intended semantics of their vocabularies in RDFS or OWL. As both of these modelling languages, when interpreted in line with the corresponding W3C specifications, are more tailored towards inference as opposed to constraint formulation, basic principles of the semantics and assumptions of the language have to be modified for data quality testing scenarios (especially the application of the Closed World Assumption and a weakened Unique Name

Assumption). However, no unanimous and detailed specification for such alternative semantics has been formulated and standardised to date.

In contrast, the semantics of each language element for SHACL Core is clearly defined in the corresponding W3C Proposed Recommendation and was designed specifically for the purpose of prescriptive constraint formulation. Furthermore, SHACL semantics are solely based on the notion of RDF Graphs, a conceptually much simpler and more approachable model for new adopters of Semantic Web technologies compared to OWL based on Description Logics.

5.2.1.6 Progress on the SHACL specification and standardisation process

ALIGNED was actively involved in co-authorship and revision of the W3C Working Group for SHACL. Since then, several stages of the W3C Recommendation Track Process[18] were passed, by opening the draft for reviewers' comments, discussing or addressing these, and putting forward an implementation report[19] about several prototypical implementations of SHACL. Hence, SHACL is now documented as a W3C Proposed Recommendation.

5.2.1.7 SHACL support in RDFUnit

RDFUnit currently contains implementations for all non-complex core constraint components (i.e., excluding logical constraint components and qualified cardinality restrictions). All variants of target declarations are implemented as well. This provides a substantial, albeit incomplete, subset of SHACL Core that already allows formulating graph constraints for many use cases. RDFUnit also supports SHACL SPARQL, i.e., defining custom constraints and constraint components by SPARQL fragments. This approach provides the whole flexibility and expressive power of that query language.

To evaluate both correctness of the validation logic of the implemented SHACL subset, a runner for the SHACL test suite[20] has been implemented. The tests can be run via a custom JUnit Runner within an IDE for quick feedback cycles about improvements or regressions on conformance during the continued work for a complete coverage of SHACL feature in RDFUnit. Additionally, an RDF document reporting on the test outcomes for the SHACL test suite using the EARL[21] vocabulary can be generated, in a format

[18]https://www.w3.org/2004/02/Process-20040205/tr.html

[19]http://w3c.github.io/data-shapes/data-shapes-test-suite/

[20]https://github.com/w3c/data-shapes/tree/gh-pages/data-shapes-test-suite/tests

[21]https://www.w3.org/TR/EARL10-Schema/

```
@prefix doap: <http://usefulinc.com/ns/doap#> .
@prefix earl: <http://www.w3.org/ns/earl#> .

<http://aksw.org/Projects/RDFUnit>

  a doap:Project ;
  a earl:Software ;
  a earl:TestSubject ;
  doap:developer <aksw.org/DimitrisKontokostas> ;
  doap:name "RDFUnit" ;
.
[
  a earl:Assertion ;
  earl:assertedBy <aksw.org/MarkusAckermann> ;
  earl:result [
     a earl:TestResult ;
     earl:mode earl:automatic ;
     earl:outcome earl:passed ;
   ] ;
  earl:subject <http://aksw.org/Projects/RDFUnit> ;
  earl:test <urn:x-shacl-test:/core/complex/personexample> ;
].
```

Figure 5.31 Excerpt of an EARL test report for the SHACL test suite.

suitable to automatically generate a compliance overview for the implementation report for SHACL. Figure 5.31 shows a sample from an EARL test report.

5.3 Expert Curation Tools and Workflows

Linked Data and semantic technologies enable the creation of rich, integrated knowledge models which describe particular domains using standardised languages such as RDF, RDFS and OWL. These technologies have obvious attractions for dataset curators as not only do they provide a range of ways in which complex relationships between entities can be specified and embedded in the data, but they also render the data amenable to sophisticated analytic techniques such as inference and other automated reasoning approaches and allow the construction of sophisticated queries which automatically combine data from different sources into a unified knowledge model. However, datasets are commonly curated by domain experts who have an intimate knowledge of the real-world domain being modelled but rarely have expertise or training in knowledge engineering or semantic technologies. Furthermore, all indications suggest that semantic modelling is a skill that is more difficult to acquire than computer programming, requiring considerable investment of time and effort. If we want to make this technology accessible to domain experts we need to develop paradigms, processes, workflows, APIs, and software tools which bridge the gap between their knowledge of the domain and the complexities of the underlying semantic models that they are manipulating.

This section describes the process and workflow models, developed within the ALIGNED project that are designed to bridge the gap between dataset curators, Linked Data and semantic Web technologies. The goal of this work is to define semi-automated methods and tools that involve human expert curators in the loop, while minimising their workload and the requirement that they understand the underlying semantic technologies. It builds upon and extends ALIGNED's system integrity enforcement framework by generating data curation workflows and dedicated user interfaces where domain experts can efficiently verify and approve data as part of a publication pipeline which incorporates both automated and human-based quality controls. It uses and extends ALIGNED's meta-modelling work, utilising the metamodel's schema validation ontology while providing fine-grained workflow models for core curation activities (e.g., adding instance data, updating schema, etc).

5.3.1 Requirements

This section defines the workflow requirements that a linked-data curation system must support in order to support domain experts in curating high-quality datasets. Common features identified in ALIGNED's use cases have been translated into the low-level system requirements necessary to provide a data curation system which can support user-level requirements.

5.3.1.1 Graduated application of semantics

Much of the Linked Data available for harvesting is loosely structured, often schema-free and based on reuse of terms from common vocabularies. However, in order to provide dataset quality enforcement, it is necessary to produce a rich, highly structured, and precisely defined schema and ensure that all instance data comply with the schema. Therefore, a Linked Data curation platform should provide support for the management of loosely structured linked-data documents and their gradual transformation into highly structured, schema-conformant high-quality knowledge graphs.

5.3.1.2 Graph – object mapping

Graphs are the knowledge representation form that underlies all Linked Data and semantic technologies. However, when it comes to human management of data, object models are ubiquitous – for example: entity relationship models, UML data models, database records, structured JSON documents. This amounts to a fundamental paradigm difference: in the object model a dataset is conceived as a collection of objects/entities, each of which has a collection of properties, which may be complex and may include links to other objects in the dataset; in the graph model, the dataset is conceived of as a collection of nodes with labelled, directed edges connecting them. In order to support dataset management by non-knowledge engineers, it is necessary to provide a curation interface which allows them to treat the dataset as if it was a collection of objects and takes care of mapping these objects to the underlying graph representation. This object interface should support, at a minimum the following functions:

5.3.1.3 Object/document level state management and versioning

In order to provide a functional object-based data curation interface, the system should provide basic state management and versioning support on a per-object level. That is to say that the system should provide the dataset curator with the ability to change the state of a data-object (e.g., by deleting it, or publishing it) and have the system accurately map this to a modification of the triples making up the object's representation in the underlying knowledge graph. It should also be possible to view and link to previous versions of particular data-objects.

5.3.1.4 Object-based workflow interfaces

In order to allow curators to manage updates to the graphs that they manage, object-based user interfaces must be provided which should display graph-updates as updates to specific data objects. Similarly control interfaces must be provided which allow curators to change the state of specific data objects and automatically translate that into graph updates.

5.3.1.5 Integrated, automated, constraint validation

A core focus of ALIGNED's research has been the development of constraint validation and error detection services and tools to support automated data quality analysis and enforcement. To make these services accessible to domain expert curators, they must be integrated into workflows which correctly trigger the appropriate validation processes in response to curator-driven actions which cause updates to the underlying graph. These processes should be, to as great an extent as possible, invisible to the curator.

5.3.1.6 Result interpretation

The major exception to the above is that, in certain cases, the results of constraint validation will indicate a situation which requires user-intervention (e.g., an error in the dataset schema) and must be reported back to the user. In such cases the system should, to as great an extent as is possible, map the error from the underlying graph model into the object model used by the curation platform.

5.3.1.7 Deferred updates

From a workflow point of view, automated and human tasks have very different characteristics: fast, synchronous, reliable (from an execution point of view) and typically semantically simple, versus slow, asynchronous, unreliable and often semantically complex. The most basic feature that is necessary to support these characteristics of human processes is deferred updates. That is to say that the curation system should allow updates to curated dataset to be deferred – stored and executed at a temporal distance which may be considerable. This is necessary to support the most basic content approval pipeline – where updates to the dataset must be approved by the curator before they are actually carried out. Deferred updates complicate curation processes considerably, because they can be invalidated by updates that happen between their definition and their acceptance. However, they are necessary in order to provide simple, efficient interfaces for curators, allowing them to, for example, simply click approve, to enact a complex, multi-faceted graph update that has been requested by another process in the system (whether human or automated).

5.3.2 Workflow/Process Models

This section describes the system dynamic models that have been developed in the ALIGNED project in order to produce a data curation system which meets the above requirements and is capable of providing a practically useful curation service for domain expert dataset curators. These models define how human curation actions and activities are integrated with automated processes to provide quality control of the dataset. Figure 5.32 shows the symbols used in the following diagrams.

5.3.2.1 Process model 1 – linked data object creation

JSON objects submitted to the API are stored as objects in a Linked Data document store and require approval by human curators and validation by the

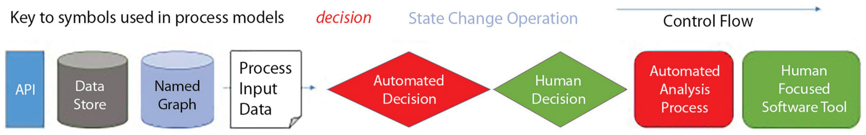


Figure 5.32 Key to workflow/process models.

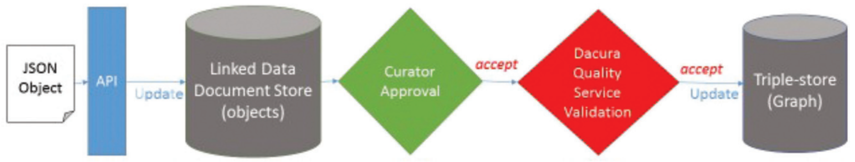


Figure 5.33 Process Model 1 – Object Creation.

automated Dacura Quality Service before being published to the triple-store. This process model is shown in Figure 5.33.

5.3.2.2 Process model 2 object – linked data object updates

Process model 2 shows how updates to Linked Data objects which are accepted by the curator but fail validation are either saved to the deferred update queue (if the object is published) or executed on the Linked Data object store (if not published). If the update passes validation, it is saved to both the Linked Data object store and the triple-store. This allows objects to be iteratively updated without having to pass DQS validation (DQS results are returned in an informational capacity).

Updates to JSON documents are subject to curator approval. Those updates which receive a ‘pending’ status from the approval process are saved to the deferred update queue. Those updates which receive an ‘accept’ status are processed by the automated DQS validation service. If the update receives a ‘reject’ status from this process, and the document being updated is in a ‘published’ state, then the update is saved to the deferred update status. If the update receives an ‘accept’ status, or the document being updated is not in a published state, the update is executed on the document in the Linked Data store. If the update receives an ‘accept’ status and the document is in a published state, the update is executed on the triple-store version of the document. This process model is shown in Figure 5.34.

5.3.2.3 Process model 3 – updates to deferred updates

Deferred updates which are approved by the human curator (state changed from ‘pending’ to ‘accept’) are first analysed by an automated consistency

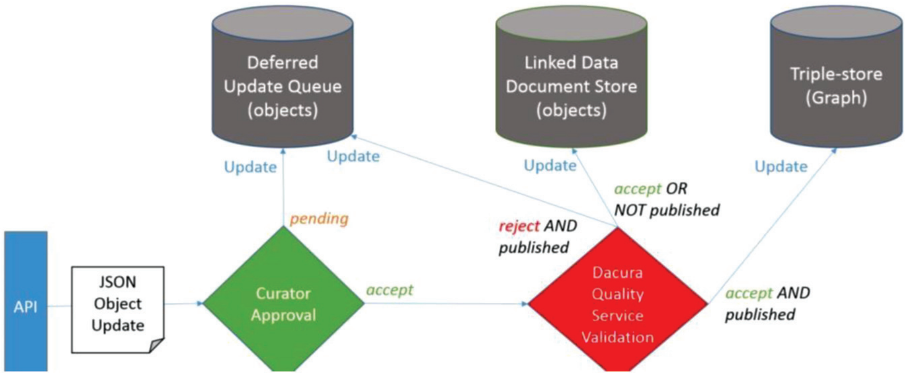


Figure 5.34 Process Model 2 – Object Update.

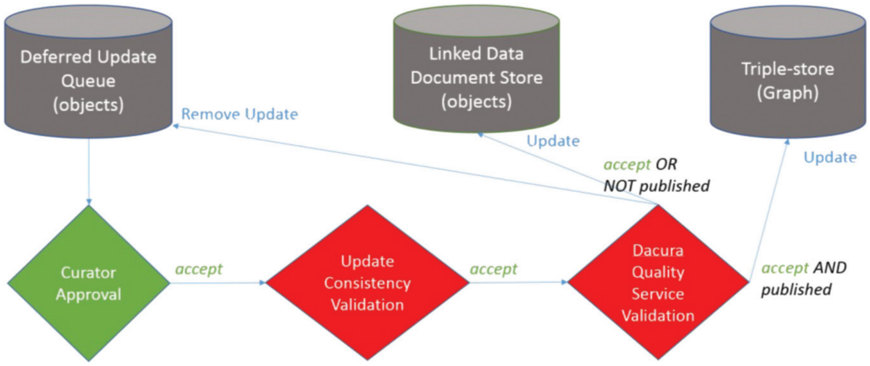

Deferred Update Queue (objects)
Linked Data Document Store (objects)
Triple-store (Graph)
Remove Update
Update
Update
accept OR NOT published
Curator Approval
accept
Update Consistency Validation
accept
Dacura Quality Service Validation
accept AND published

Figure 5.35 Process Model 3 – Updates to deferred update.

checking process, which ensures that no intervening updates have rendered the deferred update invalid. If the update is validated by this process, it is validated by the DQS process. If it correctly validates or if the updated object is not in 'published' state, the update is removed from the queue and the object is updated in the document store. If the update is validated and the updated object is in the "published" state, the graph representation of the object is updated in the triple store. This process model is shown in Figure 5.35.

5.3.2.4 Process model 4 – schema updates

Schema updates received by the API are analysed by the automated ontology dependency analysis process, which identifies the list of ontologies needed to validate schema updates and the list of ontologies needed to validate

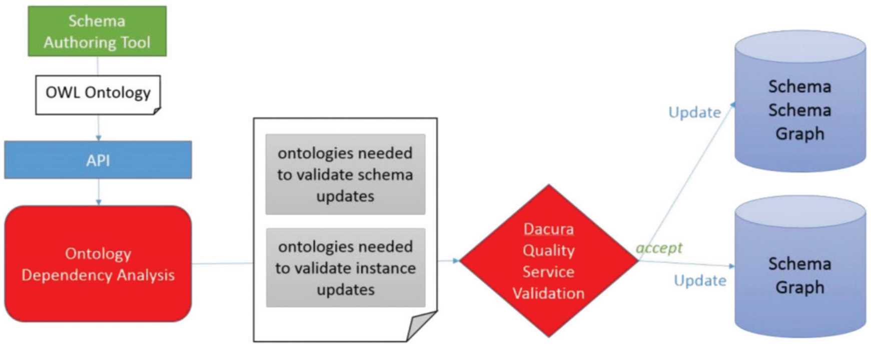

Schema Authoring Tool
OWL Ontology
API
Ontology Dependency Analysis
ontologies needed to validate schema updates
ontologies needed to validate instance updates
Dacura Quality Service Validation
accept
Update
Update
Schema Schema Graph
Schema Graph

Figure 5.36 Process model 4 – Schema Updates.

instance updates. The validity of the resulting graphs is checked using the automated Dacura Quality Service process and if successful, and the ontologies are published to the respective graphs. This process model is shown in Figure 5.36.

5.3.2.5 Process model 5 – validating schema updates

Schema updates are validated by the DQS process in two stages. First, the updates to the schema are validated by the DQS, with the schema graph serving as the instance graph and the schema schema graph serving as the schema. If this update is validated, the update to the schema is validated against the instance graph, with the schema graph serving as the schema. If either validation process fails (status: reject), the updates to the schema schema graph are rolled back. If both successfully validate, the schema graph is updated. This process model is shown in Figure 5.37.

5.3.2.6 Process model 6 – named graph creation

Instance data objects can be configured to map to a graph representation that spans multiple named graphs. Each named graph needs a schema against which the instance data will be validated. Temporary graphs are first constructed to validate the submitted ontology with the DQS, if it validates successfully, three named graphs are created – and the relevant ontologies are published to the schema schema graph and the schema graph. Updates to the graph's schema follow Process Model 4 – schema updates. This process model is shown in Figure 5.38.

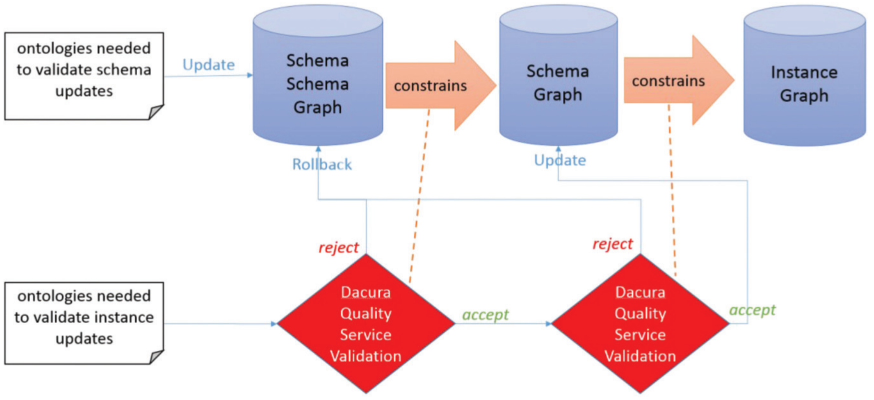

ontologies needed to validate schema updates
Update
Schema Schema Graph
constrains
Schema Graph
constrains
Instance Graph
Rollback
Update
reject
reject
ontologies needed to validate instance updates
Dacura Quality Service Validation
accept
Dacura Quality Service Validation
accept

Figure 5.37 Process model 5 – Validating schema updates.

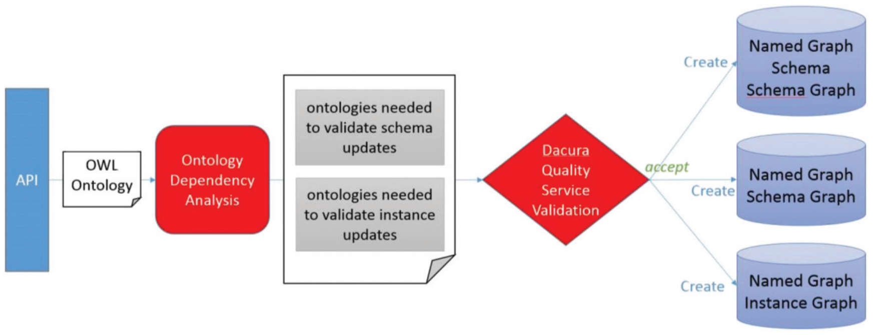

API
OWL Ontology
Ontology Dependency Analysis
ontologies needed to validate schema updates
ontologies needed to validate instance updates
Dacura Quality Service Validation
accept
Create
Create
Create
Named Graph Schema Schema Graph
Named Graph Schema Graph
Named Graph Instance Graph

Figure 5.38 Process Model 6 – Named Graph Creation.

5.3.2.7 Process model 7 – instance data updates and named graphs

This process extends and specialises Process Model 1 and Process Model 2, by providing fine-grained detail of the "update" triple-store operation. Updates to instance data objects may map to a graph representation that is distributed across multiple named graphs. Instance data updates to each named graph are validated sequentially by the automated DQS service (using the schema graph that has been configured for that named graph – see Process Model 6 – Named Graph Creation). If the update is validated across all graphs, and the updated object is in published state, the update is published

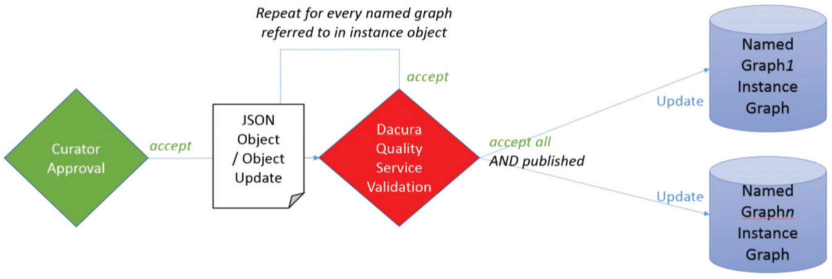

Repeat for every named graph referred to in instance object
Curator Approval
accept
JSON Object / Object Update
accept
Dacura Quality Service Validation
accept all AND published
Update
Named Graph1 Instance Graph
Update
Named Graphn Instance Graph

Figure 5.39 Process model 7 – instance data updates in named graphs.

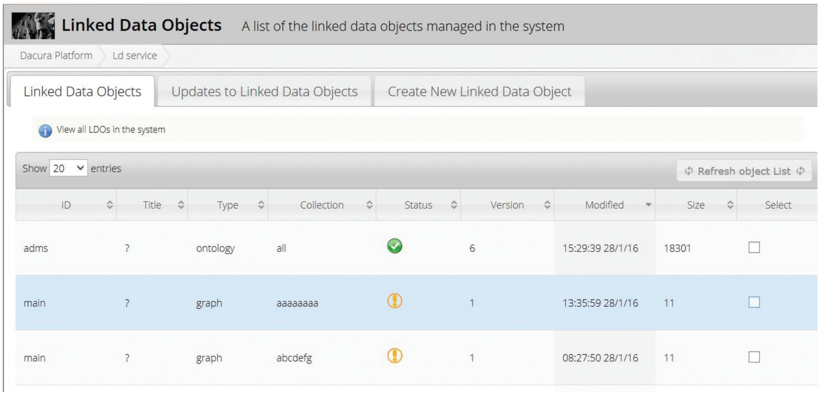
to all relevant named graph instance graphs. This process model is shown in Figure 5.39.

5.4 Dacura Approval Queue Manager

The Dacura Approval Queue Manager is a Web-based GUI tool which allows dataset curators to interact with the object creation, object updating and deferred updating processes. It allows curators to view the approval queue of new objects and updates to objects and to approve or reject object creation requests and object update requests in bulk. The Dacura Approval Queue Manager can be seen in Figure 5.40.

5.5 Dacura Linked Data Object Viewer

The Dacura Linked Data Object Viewer is a Web-based GUI tool which allows dataset curators to view the Linked Data objects that they are


Linked Data Objects A list of the linked data objects managed in the system

Dacura Platform › Ld service

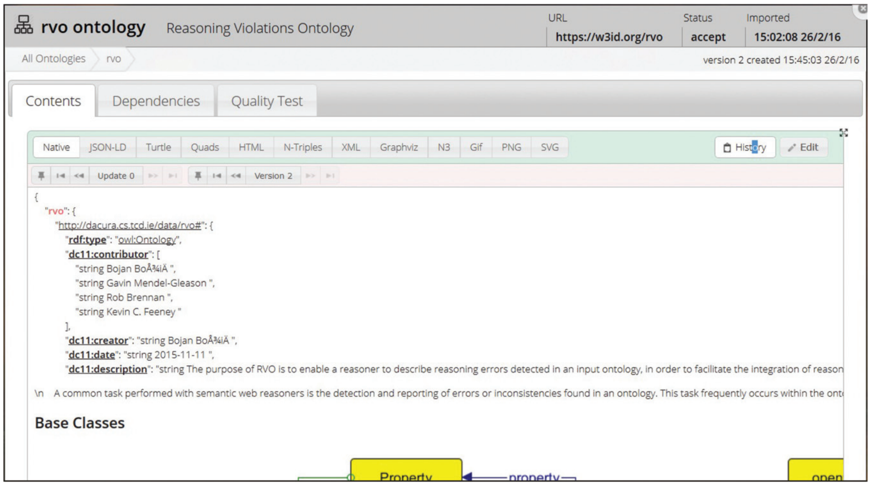
Linked Data Objects | Updates to Linked Data Objects | Create New Linked Data Object

View all LDOs in the system

Show 20 entries · Refresh object List

ID	Title	Type	Collection	Status	Version	Modified	Size	Select
adms	?	ontology	all	✓	6	15:29:39 28/1/16	18301	☐
main	?	graph	aaaaaaaa	!	1	13:35:59 28/1/16	11	☐
main	?	graph	abcdefg	!	1	08:27:50 28/1/16	11	☐

Figure 5.40 Screenshot of Dacura Linked Data Approval Queue Manager Tool.



```
{
  "rvo": {
    "http://dacura.cs.tcd.ie/data/rvo#": {
      "rdf:type": "owl:Ontology",
      "dc11:contributor": [
        "string Bojan BoÅ¾iÄ ",
        "string Gavin Mendel-Gleason ",
        "string Rob Brennan ",
        "string Kevin C. Feeney "
      ],
      "dc11:creator": "string Bojan BoÅ¾iÄ ",
      "dc11:date": "string 2015-11-11 ",
      "dc11:description": "string The purpose of RVO is to enable a reasoner to describe reasoning errors detected in an input ontology, in order to facilitate the integration of reason
\n   A common task performed with semantic web reasoners is the detection and reporting of errors or inconsistencies found in an ontology. This task frequently occurs within the ont
```

Figure 5.41 Screenshot of Dacura Linked Data Object Viewer Tool showing version browsing toolbar.

managing, browse their history, and manage their metadata and contents on an individual object basis, while maintaining a correct mapping to the object's underlying graph representation. The Dacura Linked Data Object Viewer can be seen in Figure 5.41.

5.5.1 CSP Design of Seshat Workflow Use Case

We formally specified our workflow in CSPM, a dialect of CSP (Communicating Sequential Processes) with the assistance of FDR4 The CSP Refinement Checker[22]. CSPM gives a very rich language for the specification of processes and communication, but we limit ourselves in the model to a very restricted subset with a view to later creating simple user-interfaces for the specification of alternative workflow approaches. In addition, we hope to use the specification to explore properties of the workflow model and potentially create refinements with versioning in a later iteration.

In natural language, the document curation use case can be described thus:

A user may load a candidate object into the system. It is then in a 'pending' state. From the pending state, a candidate may be checked by the DQS (Dacura Quality Service) server. The DQS server will either pass or fail the candidate. If the candidate passes DQS's inspection, it is sent to an 'ok' state. From the 'ok' state, it is possible to review or edit. If the user chooses to

[22]https://www.cs.ox.ac.uk/projects/fdr/

review, they may either accept as is and it is placed into an 'accepted' state. From an 'accepted' state, the candidate may be published. If the candidate reviewer likes, they may edit the document, sending it back to a 'pending' state. If the DQS system fails to pass a candidate, it is sent to a 'fail' state from which the user must edit the candidate before it can go back to 'pending'. Additionally, from an 'ok' state which is edited, the candidate is passed back to a 'pending' state.

5.5.2 Specification

We show in Table 5.1 the specification in CSPm of the above natural language description. We show the model with only one document for presentation purposes as little changes by increasing the number of available DOCIDS.

Table 5.1 CSPm specification of workflow

```
DOCIDS = {0 .. 2}
channel load, check, fail, edit,
   pass, store, accept, decline, review, publish, unpublish
: DOCIDS
WFS(i) = load.i -> PENDING(i)

PENDING(i) = check.i -> DQS(i)

DQS(i) = fail.i -> FAILED(i)
   |~|pass.i -> OK(i)

FAILED(i) = edit.i -> PENDING(i)

OK(i) = edit.i -> PENDING(i)
   [] review.i -> REVIEW(i)

REVIEW(i) = accept.i -> ACCEPTED(i)
   [] edit.i -> PENDING(i)

ACCEPTED(i) = publish.i -> PUBLISH(i)
   [] edit.i -> PENDING(i)

PUBLISH(i) = unpublish.i -> ACCEPTED(i)

CHOOSEDOC = |~|i : DOCIDS @ WFS(i)
assert CHOOSEDOC :[deadlock free [F]]
```

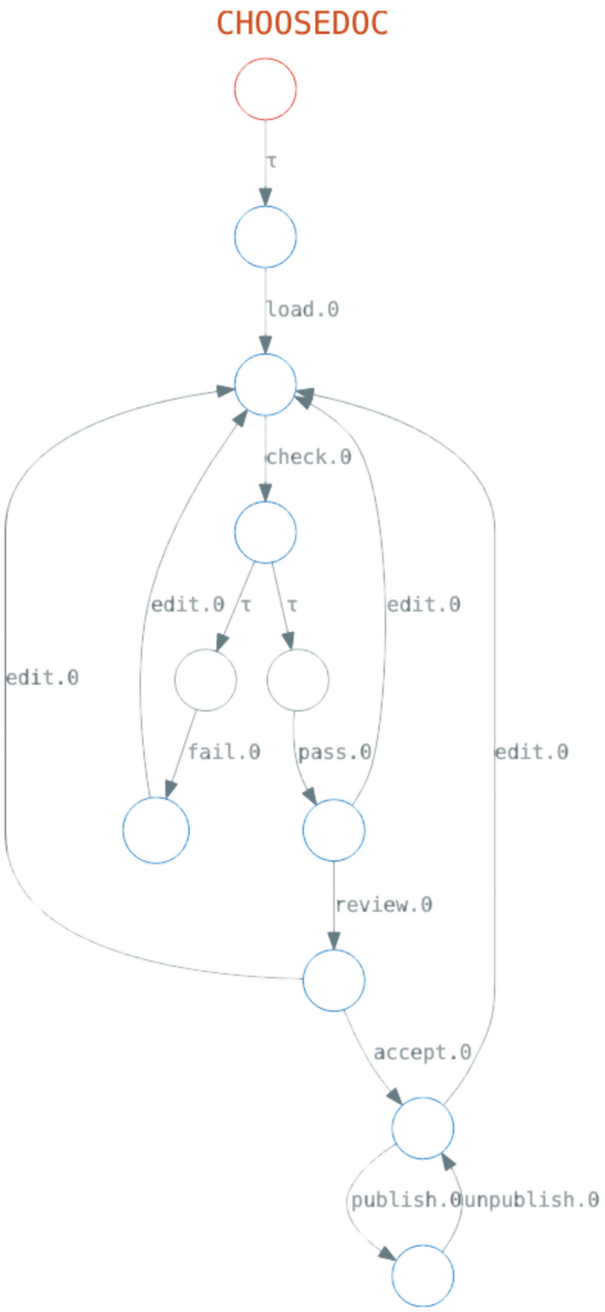


Figure 5.42 Automatically generated workflow diagram from CSPm specification.

The specification can be read as having a number of states which are parameterised by the document id which they refer to and having one or a number of actions which can be taken from those states. In the case of DQS(i), there are two possible actions which are chosen by an internal choice. In all other instances, the choice is an external (user) choice.

The example of Figure 5.42 shows the automatically generated workflow diagram from the CSPm specification. It is clear from inspection that the document is always available for transitions to some new state and we can see clearly how the workflow takes place.

5.6 Dacura Quality Service

The Dacura Quality Service (DQS) is a service for managing a triple-store and ensuring its ongoing consistency. The triple-store is an RDF graph which is stored using the ClioPatria server. ClioPatria provides a durable representation of the graph which can be accessed and updated transactionally. These features constitute the ‘A’,‘I’ and ‘D’, of ACID (Atomic, Consistent, Isolated, Durable) which are generally considered fundamental design principles for enterprise databases.

DQS extends this feature set with the ‘C’, Consistency. Consistency of the graph is described using OWL. This ontology is interpreted as constraints over the graph. Failure to meet the constraints specified in OWL leads to a counter-example of satisfaction of the constraints, or a witness of failure. These witnesses are then reported over the API to the client which provides the client with manual or automatic remedial actions.

The DQS software is provided as a plugin to ClioPatria and interaction with DQS takes place over an HTTP (HyperText Transfer Protocol) API (Application Programming Interface). The API exchanges information about triples and witnesses of failure in the widely used JSON object format.

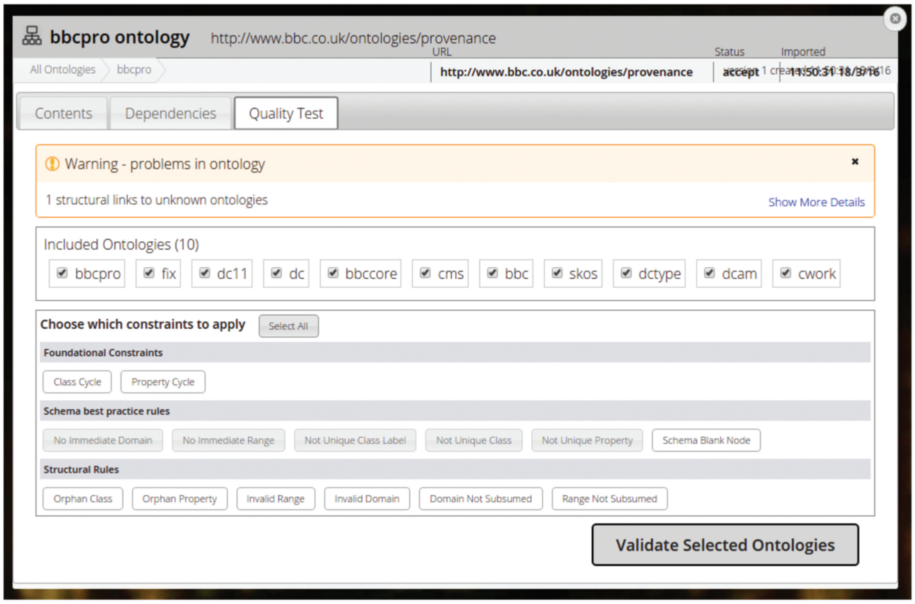
The DQS service is used by Dacura to ensure that data quality of curated data is consistent on an ongoing basis. Since the data must be amenable to constant update by data practitioners, and the data must be available for analytics in a consistent and coherent format, it is imperative that basic data consistency constraints be maintained. DQS provides a straightforward framework for assisting Dacura in maintaining these constraints.

Dacura is the main consumer of the Dacura Quality Service. However, it is completely modular and therefore could be used in other projects which would like to manage consistency constraints using OWL ontologies.

5.6.1 Technical Overview of Dacura Quality Service

The DQS Service is implemented as a plugin for ClioPatria, which is written in the prolog programming language. Prolog provides a seamless interface to the RDF triple-store as a predicate which can then be combined for the purposes of reasoning. A number of reasoning tasks are carried out by a list of predicates, which can be accessed by calling a number of pre-defined HTTP endpoints, carrying appropriate JSON POST variables which describe the relevant graphs, updates to those graphs, and various reasoning activities which should be undertaken.

Currently, the primary consumer of the DQS service is the Dacura platform which provides a user interface to the service, allowing the user to select the relevant ontologies, and instance data to be checked, and the various constraints which should be checked. The interface for schema checking is shown in Figure 5.43.


bbcpro ontology http://www.bbc.co.uk/ontologies/provenance

URL: http://www.bbc.co.uk/ontologies/provenance
Status: accept
Imported: 11:50:31 18/3/16

Warning - problems in ontology

1 structural links to unknown ontologies

Included Ontologies (10)

☑ bbcpro ☑ fix ☑ dc11 ☑ dc ☑ bbccore ☑ cms ☑ bbc ☑ skos ☑ dctype ☑ dcam ☑ cwork

Choose which constraints to apply

Foundational Constraints: Class Cycle, Property Cycle

Schema best practice rules: No Immediate Domain, No Immediate Range, Not Unique Class Label, Not Unique Class, Not Unique Property, Schema Blank Node

Structural Rules: Orphan Class, Orphan Property, Invalid Range, Invalid Domain, Domain Not Subsumed, Range Not Subsumed

Figure 5.43 Dacura platform Quality Test Interface that calls the DQS.

5.6.2 Dacura Quality Service API

The API is structured as a series of HTTP endpoints which are accessed through POST requests. The POST requests have a number of variables communicated in JSON and with some standard translations for RDF URIs and literals. We first describe this general format of RDF encoding in quads, and then the specific format of some shared POST variables.

5.6.2.1 Resource and interchange format

Inserts and deletes in the DQS system are managed through supplying quads which specify the RDF triples, and their associated graph. These are encoded in JSON which is a widely accepted format.

5.6.2.2 URI

An RDF URI resource is described as a JSON string. For instance, the following string represents the "label" property:
"http://www.w3.org/1999/02/22-rdf-syntax-ns#label"

5.6.2.3 Literals

Literals are composite objects which cannot be represented directly as a string. The format for a literal is formatted as one of the two:
{"data":"2015-06-08T12:30:00","type": "http://www.w3.org/2001/XMLSchema#dateTime"}
or
{"data": "This is a string", "lang":"en"}

5.6.2.4 Literal types

xdd:coordinatePolygon
The coordinate polygon type is represented in as a list of doubles. An informal grammar is as follows:
xdd:coordinatePolygon := [float1, float2, . . . floatn]

xdd:coordinatePolyline
The coordinate polygon type is represented in as a list of doubles identically to a coordinate polygon but with a semantics of a non-closed region. An informal grammar is as follows:
xdd:coordinatePolyline := [float1, float2, . . . floatn]

xdd:gYearRange
The xdd:gYearRange is a (possibly degenerate) range of years, with the first year smaller than or equal to the second.
xdd:gYearRange := [gYear] | [gYear1, gYear2]

xdd:integerRange
The xdd:integerRange is a (possibly degenerate) range of integers, with the first integer smaller than or equal to the second.
xdd:integerRange := [integer] | [integer1, integer2]

xdd:decimalRange
The xdd:decimalRange is a (possibly degenerate) range of decimal numbers of arbitrary precision, with the first number smaller than or equal to the second.
xdd:integerRange := [decimal] | [decimal1, decimal2]

5.6.2.5 Quads

Quads are described as lists of strings or JSON representations of resources.

```
[
"resource1",
"resource2",
"resource3",
"graph"
]

[
"resource4",
"resource5",

{
"data": "Hello world",
"lang": "en-utf8"
},
"graph"
]


[
"resource6",
"resource7",
{
"data": "2015-06-08T12:30:00",
"type": http://www.w3.org/2001/XMLSchema\#dateTime
},
"graph"
]
```

5.6.2.6 POST variables

There are a number of post variables whose format is shared amongst the various endpoints. Many endpoints require a “pragma” JSON object to be posted in the post variables, which specifies the instance graph, “instance”, the schema graph, “schema” and associated tests. It also takes a “commit” flag, which will store the changes if the tests are successful.

pragma: {“tests”:“all”,“schema”:“schemaGraphName”,“instance”:“instance GraphName”, “commit”: “true”}

In order to perform updates, we specify all quads (as described above) which are to be deleted, and then inserted. Deletes happen prior to inserts. Modification of either schema, instance or both, is possible merely by specifying the appropriate schema and instance graphs.

update: {“insert”: QUADS, “delete”: QUADS}

Example:

update: {“insert”:[[“resource1”, “resource2”,“resource3”, “instance”],[“resource6”, “resource7”, {“data”:“2015-06-08T12:30:00”, “type”: “http://www.w3.org/2001/XMLSchema#dateTime”}, “instance”]]}

5.6.2.7 Tests

A number of the API endpoints require that tests be passed to define which constraints are considered when consistency is required of the triple store. The tests are divided into two categories. One for schema constraints, all of which are suffixed with “SC”, and one for instance constraints which are suffixed with “IC”.

Users can specify a JSON list of constraints for the “test” field of a pragma, or send the string “all” which will run every available test. Specifying tests which are not available has no effect. We give the exhaustive list of tests below.

5.6.2.8 Required schema tests

These tests for class cycles in the subsumption hierarchy for classes and properties respectively. They are required for any further tests to take place as non-cyclicity is assumed in the other predicates.

“classCycleSC”, “propertyCycleSC”

5.6.2.9 Schema tests

These three tests check to see if there is a class for a given URI, which does not need to be inferred, or that a given property has a defined range and domain which is not inferred.

“noImmediateClassSC”, “noImmediateDomainSC”, “noImmediateRangeSC”

These three tests check uniqueness of definitions. In particular, the first is useful to avoid overlapping labels which can lead to confusion in interfaces which utilise the labels for display.

“notUniqueClassLabelSC”, “notUniqueClassSC”, “notUniquePropertySC”

Does the schema contain blank nodes?

“schemaBlankNodeSC”

Annotations can be used to black out various properties such that they are not reasoned over, but this test will issue an error if this is being done.

“annotationOverloadSC”

A class (property respectively) is used without definition (inferred or otherwise)

“orphanClassSC”, “orphanPropertySC”

Check for invalid ranges or domains.

“invalidRangeSC”., “invalidDomainSC”

Check to see if domain and range subsumption leads to inconsistency.

“domainNotSubsumedSC”, “rangeNotSubsumedSC”

Check to see if properties are used as both datatype and object properties simultaneously in violation of OWL.

“propertyTypeOverloadSC”

Instance Tests

Check to see if property has no defined domain (range respectively).

“noPropertyDomainIC”, “noPropertyRangeIC”

Check to see if blanknodes are being used?

“instanceBlankNodeIC”

Check to see if edges are valid under the given schema rules. Related classes, properties and restrictions as well as a number of assertions are all checked against the edges of the instance graph for conformance.

“invalidEdgeIC”

Check to see if instances have no defined class.

“edgeOrphanInstanceIC”

Check functional (inverse functional) property assertions for correctness relative the instance graph.

“notFunctionalPropertyIC”, “notInverseFunctionalPropertyIC”

Check that properties are defined.

“localOrphanPropertyIC”

5.6.2.10 Errors

The DQS API returns errors which are specified in a JSON format and which are described in the Reasoning Violations Ontology (RVO). RVO has been developed in the ALIGNED project and is fully described in Chapter 3 and is available online[23]. Further details have been published at the third Workshop on Linked Data Quality. [24]

5.6.2.11 Endpoints

/dacura/schema
POST variables: pragma, update
Requires: pragma.schema, pragma.tests, pragma.commit

Endpoint for schema updates.

/dacura/instance
POST variables: pragma, update
Requires: pragma.instance, pragma.schema, pragma.tests, pragma.commit

Endpoint for simultaneous schema and instance updates.

[23]http://aligned-project.eu/data/rvo_documentation.html
[24]“Describing Reasoning Results with RVO, the Reasoning Violations Ontology”, Bojan Bozic, Rob Brennan, Kevin Feeney and Gavin Mendel-Gleason, 3rd Workshop on Linked Data Quality, co-located with ESWC 2016, Crete, 30 May 2016.

/dacura/schema_validate
POST variables: pragma
Requires: pragma.schema, pragma.tests

Endpoint for testing validity of an already existing schema

/dacura/validate
POST variables: pragma
Requires: pragma.instance, pragma.schema, pragma.tests

Endpoint for testing validity of already existing instance/schema pair.

/dacura/test
POST variables: N/A
Requires: N/A

Runs the internal testing suite.

/dacura/entity
POST variables: entity, schema, instance
Requires: entity, schema, instance

Returns all entities in the given instance graph for the given schema.

/dacura/entity_frame
POST variables: class, schema, instance
Requires: class, schema, instance

Returns the frame associated with a given entity instance, filled with its respective values. The 'class' post variable is the URI of a valid class in the schema provided by the post variable 'schema'.

/dacura/class_frame
POST variables: class, schema
Requires: class, schema

Returns the frame associated with a given entity instance, filled with its respective values. The 'lass' is the URI of a valid class in the given schema.

/dacura/class
POST variables: schema
Requires: schema

Endpoint for obtaining information on all defined classes in a given schema.

/dacura/dacura_entity_property_frame
POST variables: schema, instance, property, entity
Requires: schema, instance, property, entity

Endpoint returns a filled frame for a given entity and property when supplied with the entity URI, the schema and instance graphs and the necessary property URI.

/dacura/subsumes
POST variables: schema,class
Requires: schema, class

Endpoint returns a list of all classes which are subsumed by the supplied class.

DQS is now relatively stable and most changes will involve bug-fixes. The most recent source code is released open-source as a plugin, available at https://github.com/GavinMendelGleason/dacura. The Dacura system will continue to maintain and update the plugin as it is required for important data curation functionality in Dacura.

5.7 Linked Data Model Mapping

5.7.1 Interlink Validation Tool

The Interlink Validation Tool is designed to be used in a scenario where a specific source dataset is being maintained. This source dataset contains interlinks to external target datasets. As the source dataset and the target datasets evolve over time, the maintainers of the source dataset need to ensure that none of the existing interlinks have become invalid due to the evolution of the datasets.

The Interlink Validation Tool was initially validated in the ALIGNED DBpedia use case. It was identified that DBpedia does not include interlink validation during its release process (activities involved when a new version of DBpedia is to be released). This can result in invalid interlinks being published in the DBpedia release, reducing overall dataset quality. The Interlink Validation Tool provides a lightweight approach to reduce the number of invalid interlinks that could get published in a dataset. While the tool does not repair interlinks, it does highlight, which interlinks have become invalid and which resource (the source dataset resource or target dataset resource) has caused it to become invalid. This information can then be used by other tools in a software and data engineering toolchain, to help in the interlink repair process. The tool was deployed in the DBpedia environment for the v.2015-10 release and discovered 53,418 invalid interlinks[25].

[25]https://sourceforge.net/p/dbpedia/mailman/message/34980754/

As an input, the tool takes a set of interlinks between the source dataset and a target dataset. The resources in the interlinks are compared to their respective datasets to discover which interlinks are still valid and which are invalid. The tool outputs a set of valid and invalid interlinks along with two log files. One log file is a human readable log indicating, which set of interlinks have been checked and which interlinks were discovered as invalid. The other log file records similar information but is encoded in RDF and uses the ALIGNED Metamodel, especially the DLO and the DBpedia use case specific ontology (crowd-sourced public datasets) to describe the activities, entities and agents in the log. This means that the RDF logs (produced by the Interlink Validation Tool) can be consumed by the ALIGNED Unified Governance Tools in an ALIGNED tool chain.

The tool has already been deployed live in the DBpedia environment for the v.2015-10 release and discovered 53,418 invalid interlinks[26].

5.7.1.1 Interlink validation

The tool validates interlinks between two or more datasets through the use of standard SPARQL[27] query templates. RDF interlinks are typically expressed as a single triple linking resources in one (source) dataset with resources in another (target) dataset. Interlinks can be validated in two ways:

Source resources of the interlinks are only checked against their respective dataset. While this is useful when it is not possible to access the target dataset, it does only validate the source resources meaning that target resources could still be invalid.

Both source and target resources are checked against their respective datasets.

Figure 5.44 shows the process of interlink validation. Since SPARQL queries are used to validate interlinks, a SPARQL query endpoint and a local triple-store are required. It is assumed that the interlinks to be validated are stored in a named graph in the local triple-store and the source dataset is stored in a separate named graph. The query templates work by accessing the source resources (subject of the triple) and target resources (object of the triple). Validation is done in the following way:

When only the source resources of the interlinks are to be validated, they are compared to the source dataset to see, if those resources exist. If the source

[26]https://sourceforge.net/p/dbpedia/mailman/message/34980754/

[27]The query language for RDF, https://www.w3.org/TR/rdf-sparql-query/

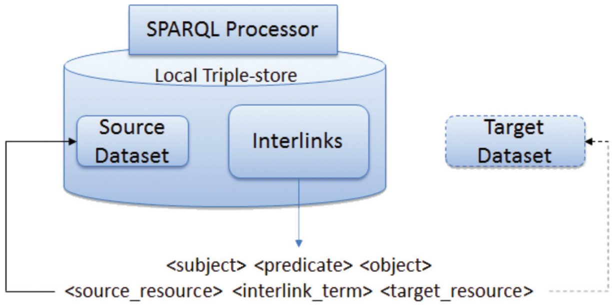

SPARQL Processor
Local Triple-store
Source Dataset
Interlinks
Target Dataset
<subject> <predicate> <object>
<source_resource> <interlink_term> <target_resource>

Figure 5.44 Interlink Validation Process.

resource of an interlink does not exist in the source dataset, then that interlink is classified as invalid.

When both the source and target resources of the interlinks are to be validated, then these resources are compared to their respective source and target datasets to see, if they exist. If either the source or target resource of an interlink does not exist in their respective dataset, then that interlink is classified as invalid. External target datasets can be accessed through loading them temporarily into the local triple-store or through remote access via a federated SPARQL query.

While this approach for validating interlinks is lightweight (relying on standard SPARQL queries), it does have a drawback. This approach cannot detect interlinks that have become invalid due to a resource merge or resource split event,[28] that can occur to resources in evolving datasets. A resource merge is done, when two or more resources from a dataset merge into a single resource and a resource split is done, when a single resource splits into two or more resources. A situation can arise where a resource merge or split takes place and the original resource identifier does not change. Therefore, a resource may have changed semantically, but is syntactically still the same. It is this particular situation, where a resource changes semantically but not syntactically, where this approach for detecting invalid interlinks will fail. In practice, given the dynamic nature of data on the Web, supporting distributed maintenance of data, without detecting resource merge or split events is still very valuable.

[28]Dos Reis, J. C., Pruski, C., Da Silveira, M. and Reynaud-Delaître, C. “Analyzing and supporting the mapping maintenance problem in biomedical knowledge organization systems.” In Proc. of the Workshop on Semantic Interoperability in Medical Informatics collocated with the 9th Extended Semantic Web Conference, pp. 25–36, 2012.

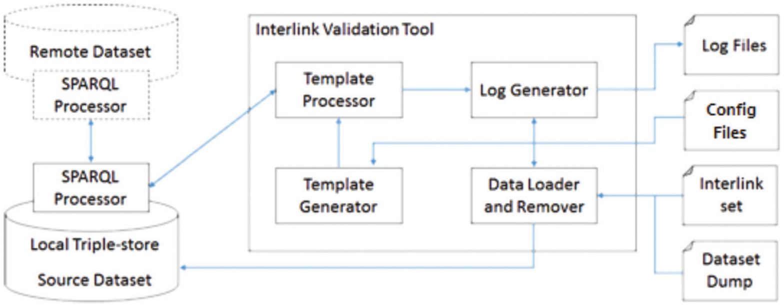


Figure 5.45 Operation of the Interlink Validation Tool. The arrows indicate the flow of information/data among the different components.

5.7.1.2 Technical overview

In this subsection, the operation of the tool is described. Figure 5.45 displays the operation of the tool and its different components.

The interlink validation tool reads in two configuration files. The first configuration file contains parameters about accessing the source dataset in the local-triple-store and some details about the source dataset itself (that will be used in the log files). The second configuration file is where a user specifies the details about each set of interlinks (between the source dataset and multiple external target datasets) that are to be validated. The parameters in this file are described in detail in the next section but they allow a user to set: the name of the external target dataset, the location of the interlink set to be validated, flags specifying validation behaviour and scope, and federated SPARQL query details.

When all the parameters are set, the tool can be run. The tool first gets the location of the interlink set from the second configuration file and loads it into a named graph in the local triple-store.

Next, based on the third parameter in the second configuration file, a SPARQL query template is generated – for example, if the external dataset is to be accessed via a federated query, then a federated query call will be included in the query template, with the external SPARQL endpoint URI provided by parameter 4.

If the third parameter specifies that an external dataset is to be accessed from a dataset dump file, then this dump file is retrieved and loaded into a named graph in the local triple-store.

Next, the template processor sends the query template to the SPARQL endpoint of the local triple-store for execution. The source resources are always checked against the source dataset. The execution results are then returned to the template processor, which sends the results to the log generator.

Then based on the parameters specified in both configuration files, and the execution results sent from the template processor, two log files are produced. One log file is a human readable log, describing which set of interlinks have been checked and which interlinks were discovered as invalid. The other log file records similar information, encoded in RDF and uses the ALIGNED Design life cycle Ontology to describe the logs.

Finally, the tool removes all temporary created data loaded into the local triple-store. The tool will repeat this process for all interlink sets specified in the second configuration file.

User guide

This subsection provides a guide on how to use the Interlink Validation Tool.

The tool is a Java program designed to be run in a UNIX environment via the command line. The current prototype of the tool is designed to be used with a Virtuoso[29] triple-store only. The tool consists of three files and three directories:

```
The 'interlink_validator.java' file
The 'iv_config.txt' file
The 'external_datasets.txt' file
The 'valid' directory
The 'invalid' directory
The 'temp' directory
```

5.7.1.3 Configuration via iv_config.txt

The iv_config.txt file contains seven parameters that need to be set. These parameters are:

Parameter1 (p1=): The file path to Virtuoso's isql utility. This is necessary to be able to load data into the local triple-store and execute SPARQL queries.

Parameter2 (p2=): Virtuoso's dba password. Similar to the above point, this is needed in order to access Virtuoso's triple-store.

Parameter3 (p3=): The graph name where the local dataset is stored in the triple-store. This specifies the location of the local dataset where the source dataset resources in the interlinks to be validated will be compared against.

Parameter4 (p4=): The file path where the (human readable) log file will be generated. If this parameter is not the log file, then it will be generated in the same directory as well.

[29]http://virtuoso.openlinksw.com/

Parameter5 (p5=): The file path where the RDF log file will be generated. If this parameter is not the log file, then it will be generated in the same directory as well.

Parameter6 (p6=): A URI to provide reference to the source dataset. This is used to refer the source dataset and is used in the RDF log generated by the tool.

Parameter7 (p7=): A URI to provide reference to the source dataset can be accessed. The reference can be a Web page containing dump files or a SPARQL endpoint. This is also used in the RDF log generated by the tool.

Each parameter is to be provided on a separate line in this file.

5.7.1.4 Configuration via external_datasets.txt

The external_datasets.txt file contains parameters to be set. Up to six parameters can be set for each set of interlinks that are to be validated:

Parameter1 (p1=): Provide a name for the external dataset that will appear in the log files.

Parameter2 (p2=): Provide a URI or file path to the file containing the interlinks that are to be validated.

Parameter3 (p3=): State whether the external dataset will be accessed through:

A federated query {F}.

A federated query with a named graph {FG}

A named graph {G] in the local triple-store.

A dump file {D}

None {N}, which means that only the source dataset resources in the interlinks will be validated.

Parameter4 (p4=): Depending on the setting done in p3, the following options are available:

If “F” was stated for parameter 3, provide the external dataset SPARQL endpoint URI.

If “FG” was stated for parameter 3, provide the external dataset SPARQL endpoint URI along with the named graph URI (see Parameter5).

If “G” was stated for parameter 3, provide the graph name where the external dataset is stored, in the local triple-store.

If “D” was stated for parameter 3, provide a URI or a file path to the dump file of the external dataset.

If “N” was stated for parameter 3, parameter 4 can be left blank.

Parameter5 (p5=): Depending on the setting done in p3, the following options are available:

If “FG” was stated for parameter 3, then provide the named graph URI, where the external dataset is stored.

Parameter6 (p6=): [Optional Parameter] Provide a URI to reference the external target dataset. This will be used in the RDF log generated by the tool.

One set of parameters must be provided per line in the file and each parameter must be separated by a “ ” (blank space).

5.7.1.5 Execute the interlink validator tool

When the two configuration files (iv_config.txt and external_datasets.txt) have been configured, the tool can be executed. To execute the tool, use the following command:

```
java InterlinkValidator
```

After a successful execution, two log files will be generated in the specified location. In addition, files containing the valid interlinks and the invalid interlinks will be generated in the respective directories.

5.7.2 Dacura Linked Model Mapper

The Dacura Linked Data Model Mapping service has been developed to help users to create rich ontological models from semi-structured HTML input and then to automate the harvesting of instance data that conform to the model, again sourced from semi-structured HTML input. This process involves a series of structural and semantic mappings to be applied on both sides – in generating the model and generating the instance data input mapping.

The service is designed to be used in a scenario where a data model is implicitly defined in a HTML page with markup used to identify labels of properties (e.g., <h3> or <strong> tags). This is a common scenario where a wiki or other CMS is used to collate a structured dataset. Unfortunately, from a machine's point of view, the data are semi-structured at best – the structure is designed primarily to be human-interpretable and we cannot even assume that the HTML is well-formed, never mind that there will be consistency in tags used or their attributes. Nevertheless, in almost all real cases, it will be possible to identify some pattern used in the HTML that can be mapped to a feature of the model.

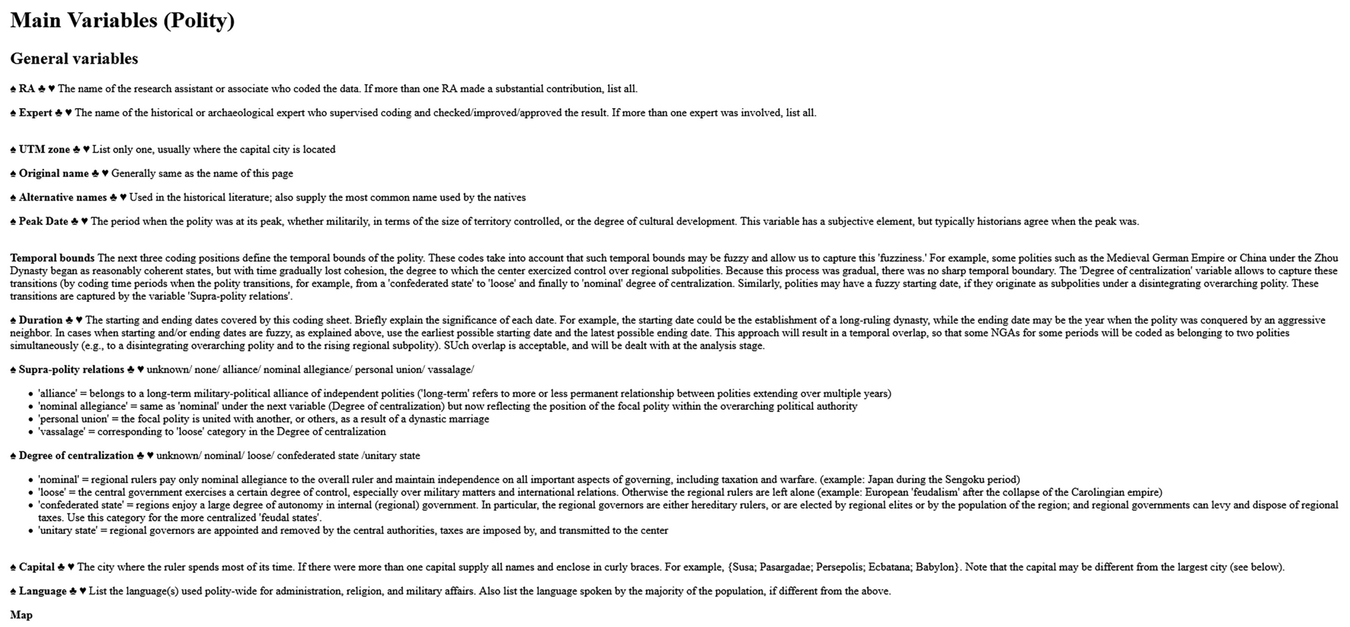
The service was developed to support the ALIGNED Seshat use. Seshat researchers have collected a large quantity of data using a wiki. The data as a list of variables, organised into sections delineated by a variety of HTML tags (Figure 5.46) with variables identified by a label between special characters and variable values having a special syntax, which captures uncertainty and disagreement and temporal scoping, followed by free html containing citations and commentary on the value.

The wiki worked well as a tool for collecting a large volume of data by a distributed team of researchers – over 150,000 facts were collected on the wiki. However, the process of extracting and cleaning data from the wiki for analysis became overwhelming over time. Thus, the goal of the Linked Data model mapper service is to automate the process of importing both the model and the instance data from the wiki to generate a structured, semantic format that is ready for analysis.

The tool allows users to map from a semi-structured wiki data-model to a rich structured semantic model. However, it cannot create structure from nothing – thus if the user wishes to use a highly structured data model with complex containment relationships, this should be defined by the user before importing the model by creating the necessary classes and properties to bind the object's basic containment structure together.

When the service is used to add a new property to the data model, the system generates a location pattern which is associated with the property. This pattern is then used to locate and import instance data elsewhere on the wiki. The service uses ALIGNED metamodel ontologies RVO, PROV, and the Seshat domain ontology. The tool has been deployed live in the Seshat use-case and was used to create Seshat's first public release of data in April 2017.[30]

[30]http://dacura.scss.tcd.ie/seshat/



Main Variables (Polity)

General variables

♠ **RA** ♣ ♥ The name of the research assistant or associate who coded the data. If more than one RA made a substantial contribution, list all.

♠ **Expert** ♣ ♥ The name of the historical or archaeological expert who supervised coding and checked/improved/approved the result. If more than one expert was involved, list all.

♠ **UTM zone** ♣ ♥ List only one, usually where the capital city is located

♠ **Original name** ♣ ♥ Generally same as the name of this page

♠ **Alternative names** ♣ ♥ Used in the historical literature; also supply the most common name used by the natives

♠ **Peak Date** ♣ ♥ The period when the polity was at its peak, whether militarily, in terms of the size of territory controlled, or the degree of cultural development. This variable has a subjective element, but typically historians agree when the peak was.

Temporal bounds The next three coding positions define the temporal bounds of the polity. These codes take into account that such temporal bounds may be fuzzy and allow us to capture this 'fuzziness.' For example, some polities such as the Medieval German Empire or China under the Zhou Dynasty began as reasonably coherent states, but with time gradually lost cohesion, the degree to which the center exercized control over regional subpolities. Because this process was gradual, there was no sharp temporal boundary. The 'Degree of centralization' variable allows to capture these transitions (by coding time periods when the polity transitions, for example, from a 'confederated state' to 'loose' and finally to 'nominal' degree of centralization. Similarly, polities may have a fuzzy starting date, if they originate as subpolities under a disintegrating overarching polity. These transitions are captured by the variable 'Supra-polity relations'.

♠ **Duration** ♣ ♥ The starting and ending dates covered by this coding sheet. Briefly explain the significance of each date. For example, the starting date could be the establishment of a long-ruling dynasty, while the ending date may be the year when the polity was conquered by an aggressive neighbor. In cases when starting and/or ending dates are fuzzy, as explained above, use the earliest possible starting date and the latest possible ending date. This approach will result in a temporal overlap, so that some NGAs for some periods will be coded as belonging to two polities simultaneously (e.g., to a disintegrating overarching polity and to the rising regional subpolity). SUch overlap is acceptable, and will be dealt with at the analysis stage.

♠ **Supra-polity relations** ♣ ♥ unknown/ none/ alliance/ nominal allegiance/ personal union/ vassalage/

- 'alliance' = belongs to a long-term military-political alliance of independent polities ('long-term' refers to more or less permanent relationship between polities extending over multiple years)
- 'nominal allegiance' = same as 'nominal' under the next variable (Degree of centralization) but now reflecting the position of the focal polity within the overarching political authority
- 'personal union' = the focal polity is united with another, or others, as a result of a dynastic marriage
- 'vassalage' = corresponding to 'loose' category in the Degree of centralization

♠ **Degree of centralization** ♣ ♥ unknown/ nominal/ loose/ confederated state /unitary state

- 'nominal' = regional rulers pay only nominal allegiance to the overall ruler and maintain independence on all important aspects of governing, including taxation and warfare. (example: Japan during the Sengoku period)
- 'loose' = the central government exercises a certain degree of control, especially over military matters and international relations. Otherwise the regional rulers are left alone (example: European 'feudalism' after the collapse of the Carolingian empire)
- 'confederated state' = regions enjoy a large degree of autonomy in internal (regional) government. In particular, the regional governors are either hereditary rulers, or are elected by regional elites or by the population of the region; and regional governments can levy and dispose of regional taxes. Use this category for the more centralized 'feudal states'.
- 'unitary state' = regional governors are appointed and removed by the central authorities, taxes are imposed by, and transmitted to the center

♠ **Capital** ♣ ♥ The city where the ruler spends most of its time. If there were more than one capital supply all names and enclose in curly braces. For example, {Susa; Pasargadae; Persepolis; Ecbatana; Babylon}. Note that the capital may be different from the largest city (see below).

♠ **Language** ♣ ♥ List the language(s) used polity-wide for administration, religion, and military affairs. Also list the language spoken by the majority of the population, if different from the above.

Map

Figure 5.46 Example of seshat code book page.

5.7.3 Model Mapper Service

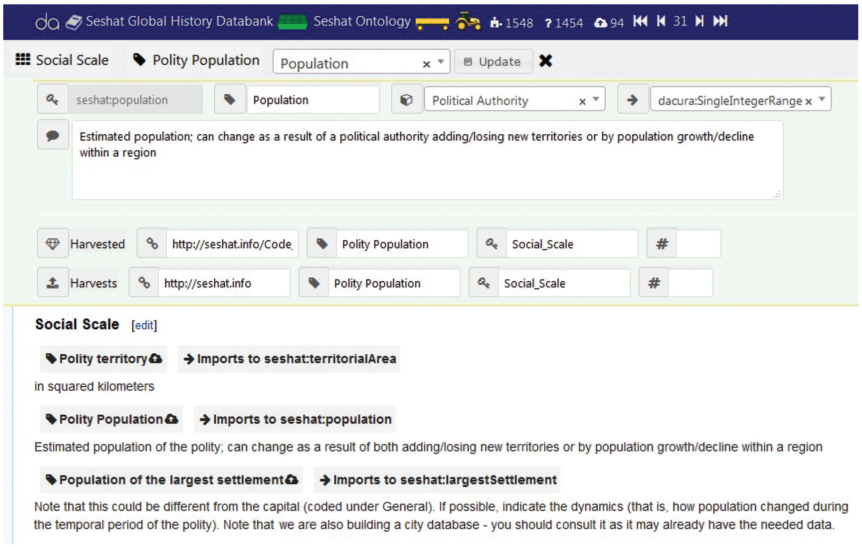
This section provides an overview of the service: first, we outline how the modelling tool creates mappings between a model and HTML patterns. Then we outline how the harvesting tool uses these patterns to automate the harvesting of semi-structured data.

5.7.3.1 Modelling tool – creating mappings

Dacura's Modelling tool, shown in Figure 5.47, enables users to create the structure of the dataset from existing sources. In this case, we used the Seshat code book page as the basis of our model. Dacura associates the imported properties with the pattern of the HTML that they were imported from. It uses this pattern later to automatically find and import data from the rest of the wiki into the structured model, as shown in Figure 5.48.

5.7.3.2 Importing semi-structured data with data harvesting tool

The Dacura data harvesting tool can be run on any Web page. When it loads, it attempts to fit the data on the page to the shape of the model using the patterns associated with the model that were created upon import. It uses Dacura's quality control API to test different possibilities in order to identify the best fit, as shown in Figure 5.49. It can even automatically correct mistakes.



Estimated population; can change as a result of a political authority adding/losing new territories or by population growth/decline within a region

Social Scale [edit]

Polity territory → Imports to seshat:territorialArea

in squared kilometers

Polity Population → Imports to seshat:population

Estimated population of the polity; can change as a result of both adding/losing new territories or by population growth/decline within a region

Population of the largest settlement → Imports to seshat:largestSettlement

Note that this could be different from the capital (coded under General). If possible, indicate the dynamics (that is, how population changed during the temporal period of the polity). Note that we are also building a city database - you should consult it as it may already have the needed data.

Figure 5.47 Importing a model from semi-structured HTML source.

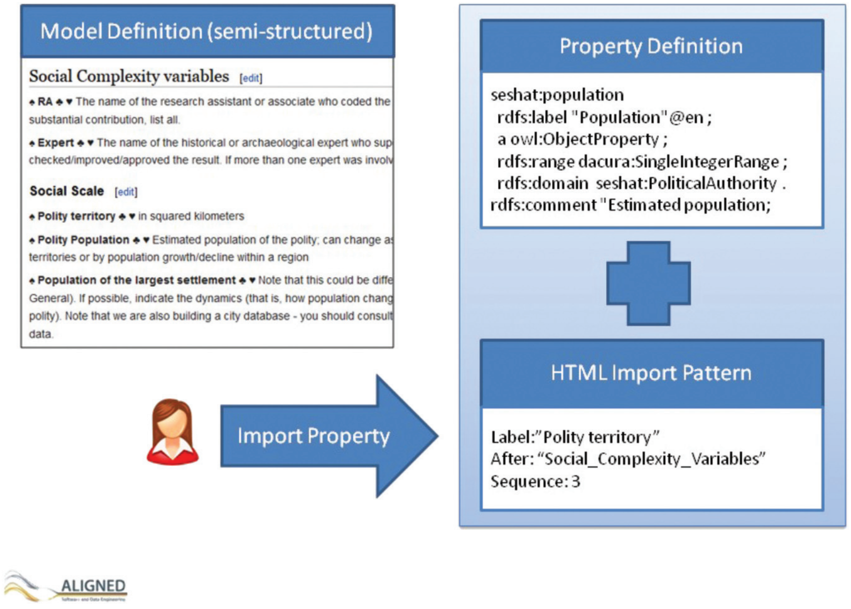

Model Definition (semi-structured)
Social Complexity variables [edit]
♠ RA ♣ ♥ The name of the research assistant or associate who coded the substantial contribution, list all.
♠ Expert ♣ ♥ The name of the historical or archaeological expert who sup checked/improved/approved the result. If more than one expert was involv
Social Scale [edit]
♠ Polity territory ♣ ♥ in squared kilometers
♠ Polity Population ♣ ♥ Estimated population of the polity; can change a territories or by population growth/decline within a region
♠ Population of the largest settlement ♣ ♥ Note that this could be diffe General). If possible, indicate the dynamics (that is, how population chang polity). Note that we are also building a city database - you should consult data.
Import Property
Property Definition
seshat:population
rdfs:label "Population"@en ;
a owl:ObjectProperty ;
rdfs:range dacura:SingleIntegerRange ;
rdfs:domain seshat:PoliticalAuthority .
rdfs:comment "Estimated population;
HTML Import Pattern
Label:"Polity territory"
After: "Social_Complexity_Variables"
Sequence: 3
ALIGNED

Figure 5.48 Process for associating property definitions in a model with a pattern within a semi-structured HTML page.

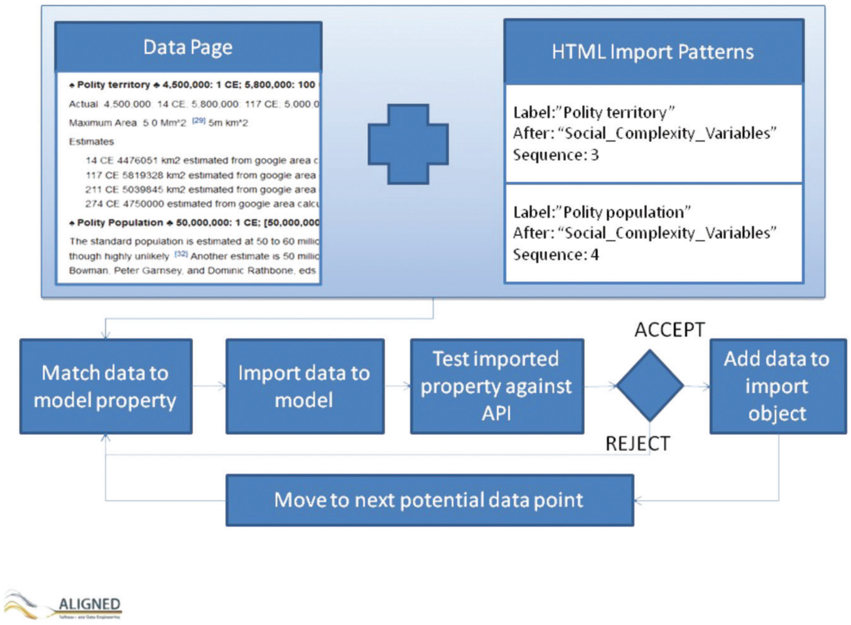
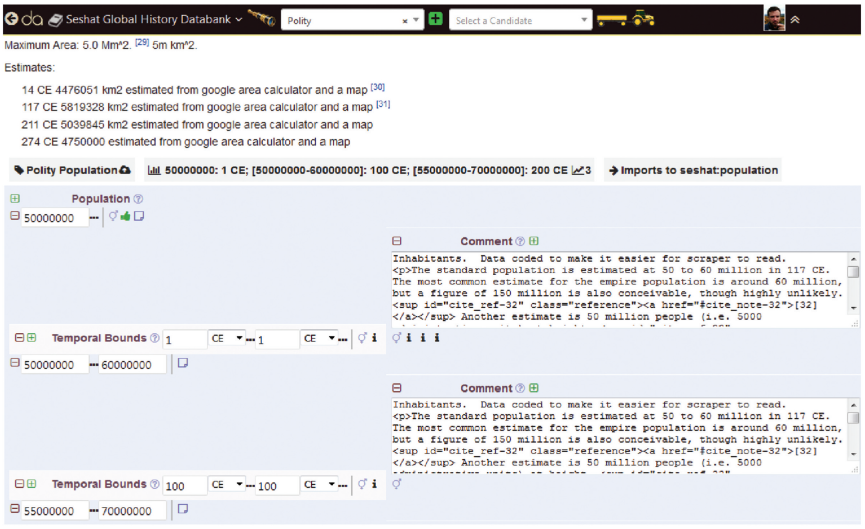

Data Page
♠ Polity territory ♣ 4,500,000: 1 CE; 5,800,000: 100
Actual 4,500,000 14 CE 5,800,000 117 CE 5,000,0
Maximum Area 5.0 Mm^2 [29] 5m km^2
Estimates
14 CE 4476051 km2 estimated from google area c
117 CE 5819328 km2 estimated from google area
211 CE 5039845 km2 estimated from google area
274 CE 4750000 estimated from google area calcu
♠ Polity Population ♣ 50,000,000: 1 CE; [50,000,000
The standard population is estimated at 50 to 60 millio though highly unlikely [32] Another estimate is 50 millio Bowman, Peter Garnsey, and Dominic Rathbone, eds
HTML Import Patterns
Label:"Polity territory"
After: "Social_Complexity_Variables"
Sequence: 3
Label:"Polity population"
After: "Social_Complexity_Variables"
Sequence: 4
Match data to model property
Import data to model
Test imported property against API
ACCEPT
REJECT
Add data to import object
Move to next potential data point
ALIGNED

Figure 5.49 Process for using patterns to extract data from semi-structured html pages.



Maximum Area: 5.0 Mm^2. [29] 5m km^2.

Estimates:

14 CE 4476051 km2 estimated from google area calculator and a map [30]
117 CE 5819328 km2 estimated from google area calculator and a map [31]
211 CE 5039845 km2 estimated from google area calculator and a map
274 CE 4750000 estimated from google area calculator and a map

Polity Population | 50000000: 1 CE; [50000000-60000000]: 100 CE; [55000000-70000000]: 200 CE 3 | Imports to seshat:population

Population
50000000

Comment

```
Inhabitants.  Data coded to make it easier for scraper to read.
<p>The standard population is estimated at 50 to 60 million in 117 CE.
The most common estimate for the empire population is around 60 million,
but a figure of 150 million is also conceivable, though highly unlikely.
<sup id="cite_ref-32" class="reference"><a href="#cite_note-32">[32]
</a></sup> Another estimate is 50 million people (i.e. 5000
```

Temporal Bounds 1 CE 1 CE
50000000 60000000

Comment

```
Inhabitants.  Data coded to make it easier for scraper to read.
<p>The standard population is estimated at 50 to 60 million in 117 CE.
The most common estimate for the empire population is around 60 million,
but a figure of 150 million is also conceivable, though highly unlikely.
<sup id="cite_ref-32" class="reference"><a href="#cite_note-32">[32]
</a></sup> Another estimate is 50 million people (i.e. 5000
```

Temporal Bounds 100 CE 100 CE
55000000 70000000

Figure 5.50 Screenshot showing results of automated importing of semi-structured HTML data into structured model.

It rewrites the Web page to show the user what the data would look like if it were imported into Dacura, as shown in Figure 5.50. This allows users to visualise and refine the mappings to ensure that as much data as possible can be imported. Once the user is happy with the mapping, they can import all the data from the entire wiki with a single click.

5.8 Model-Driven Data Curation

The model-driven data curation interfaces provide tools for the automatic generation of data-curation interfaces. These interfaces enable the creation of ontological models and the update of data, which respects these models with a high level of agility and flexibility of model.

The interface specifications, known herein as frames, are generated automatically from ontologies.

These ontologies are specified in RDF/OWL. The frames are generated by the Dacura Quality Service.

These frames are consumed by the Dacura platform, which utilises it both for its back-end management and for the more user-focussed Dacura console. Utilising the Dacura console, users can introduce new data or edit existing data from DQS via entry forms generated by javascript from the frame specification. Additionally, the model itself can be incrementally updated from the Dacura console in architect mode (for users with suitable permissions).

The software facilitates the Seshat use-case, which requires that we are able to import, track, update and delete from a large existing dataset (on a wiki) which is highly unstructured, into a highly structured format suitable for mathematical analysis of various historical trends. The software has already been utilised in modification of the ontologies developed in ALIGNED and has improved the agility of our model development, and consequently the automatically generated user interfaces.

Highly structured Linked Data often suffers from poor quality. Hence, the software helps to guarantee strong data quality standards by the structure of the user interface itself. Furthermore, it can be enhanced by quality checks after data have been constructed.

While this code is used in the Seshat use-case, its flexibility makes it broadly applicable to a wide range of data-curation uses. This would include any use case in which there needs to be model flexibility and data entry via the Web and especially collection of human or automatically facilitated collection of information from highly unstructured data sources.

The current implementation is a basis for further development, which will include enriching the user interface with additional data entry types, which enhance the user experience of data entry. This will include the ability to describe territories on maps, the inclusion of data ranges and autocomplete comboboxes for entering pre-existing objects. Additionally, richer constraints will be checked on the client side using code auto-generated from restrictions given in frame specifications.

We begin with the specification of frames which are generated by DQS. We then describe the production of the user-interface elements from these frames.

5.8.1 Dacura Quality Service Frame Generation

The Dacura Quality Service has been extended to produce frames, which constitute specifications for user interfaces derived automatically from ontologies, which are described using RDF/OWL. The service is structured as a plugin to the ClioPatria semantic Web server and providing a number of new API endpoints which allow clients to interact with the ontology.

We briefly describe the structure of frames, which are detailed as abstract datatypes using JSON. We will then describe the API which is used to obtain frames and the data associated with them for a given ontology.

All code and API endpoints documentation for DQS is available on Github.[31]

5.8.2 Frames for UserInterface Design

Frames are specified using JSON[32]. This provides a useful interchange format for Web APIs, for which there is tool support available in virtually every modern programming language.

Frames give information about an object, the classes they are associated with and which properties are accessible to them given the ontological specification. Since, in general, it is possible for the entire RDF graph to be transitively accessible to a given class, we further restrict the generation of frames to truncate the graph at any object which has been described as a dacura:Entity (that is, the given class is an owl:subclassOf dacura:Entity). This gives us a fragment of the graph which is amenable to the creation of a usable dataentry interface.

In every case, we give the domain and range of the properties associated with a given class. If the range is a class which is not a dacura:Entity type then we include the frame associated with that class. If it is a datatype, we give back sufficient information to aid in the construction of the userinterface element. This includes the datatype, which is entered along with a potential restriction on that type, which further constrains its behaviour.

5.8.3 SemiFormal Frame Specification

In Table 5.2, we demonstrate the grammar of frames in a variant of EBNF, which describes the JSON objects that are produced by the DQS framework in accordance with a given ontology.

First, we describe some of the idioms used in our EBNF, which has been modified to reflect the use of JSON as the objects of interest. This should be considered indicative of the actual format useable by software engineers who are working with the object, rather than as a strictly formal specification.

There are two primary formats that are returned for frames. One is the purely abstract empty object associated with a class for use as a template for user interfaces, and the second is a filled frame, which is a frame that fills

[31] Dacura Quality Service Cliopatria plugin https://github.com/GavinMendelGleason/dacura

[32] The JavaScript Object Notation (JSON) Data Interchange Format RFC 7159 http://rfc7159.net/

such an abstract object with concrete triples from the instance graph for the given class.

In Table 5.2, the Frame syntactic element provides the toplevel object which is returned in JSON format by the endpoints. The dominValue and

Table 5.2 Dacura Quality Service Frame Grammar

```
Language := "en" | ...
XSDType := "xsd:integer" | "xsd:gYear" | ...
Literal :=
   { "lang" : Language,
       "data" : "..."}
   | { "type" : XSDType,
      "data" : "..."}
OwlProperty := URI
OwlClass := URI
Op := "and" | "or" | "not" | "xor"
PropertyRestriction :=
    true
   | { "type" : Op, "operands" : [PropertyRestriction] }
   | { "mincard" : ℕ, "valuesFrom" : OwlClass }
   | { "maxcard" : ℕ, "valuesFrom" : OwlClass }
   | { "card" : ℕ, "valuesFrom" : OwlClass }
   | { "hasValue" : OwlClass }
   | { "allValuesFrom" : OwlClass }
   | { "someValuesFrom" : OwlClass }
Property :=
   { "type" : "objectProperty",
      "domain" : OWLClass,
      "property" : OWLProperty,
      "range" : OWLClass,
      <"label" : Literal >, <"comment" : Literal >,
      <"domainvalue" : Value(PropertyType) >,
      <"frame" : FRAME >,
      <"restriction" : PropertyRestriction > }
   | { "type" : "datatypeProperty",
      "domain" : OWLClass,
      "property" : OWLProperty,
      "range" : OWLClass,
      <"label" : Literal >, <"comment" : Literal >,
      <"domainValue" : URI >,
      <"rangeValue" : Literal >,
      <"restriction" : PropertyRestriction > }
   | { "type" : "restriction",
        "property" : OWLProperty,
        "restriction" : PropertyRestriction }
PropertyFrame := [Property]
LogicalFrame := {"type" : Op, "operands" : [Frame]}
OneOfFrame := {"type" : "oneOf", "elements" : [URI]}
EntityFrame := {"type" : "entity", "class" : URI, <"domainValue" : URI>}
Frame := LogicalFrame | PropertyFrame | OneOfFrame | EntityFrame
```

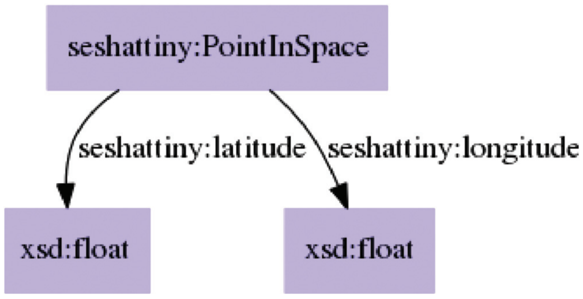


Figure 5.51 Graphical Representation of ontology fragment.

rangeValue elements are optional, and are only returned when querying for filled frames.

Schematically, Frames are used to produce empty forms with the appropriate userinterface elements for the data, while filled frames are used to create prepopulated entry forms, in the event that the data for an object is already known.

The optional "label" and "comment" fields are not essential in all cases, but are used in the automatic production of userinterface element labels and tool tips when present. Figure 5.51 shows an ontology fragment.

5.8.4 Frame API Endpoints

We briefly note here API endpoints used in the DQS for the generation and manipulation of frames.

/dacura/entity_frame
POST variables: class, schema, instance
Requires: class, schema, instance
Returns: Frame |Error

Returns the frame associated with a given entity instance, filled with its respective values. The 'class' post variable is the URI of a valid class in the schema provided by the post variable 'schema'.

/dacura/class_frame
POST variables: class, schema
Requires: class, schema
Returns: Frame |Error

Returns the frame associated with a given class. The ‘class’ is the URI of a valid class in the given schema.

/dacura/element_annotation
POST variables: schema, instance, property, element
Requires: schema, property, element
Returns: Frame |Error
The endpoint returns a Frame associated with a given annotation in the annotation graph given by ‘instance’ and associated with the data element ‘element’.