FAKULTÄT FÜR !NFORMATIK

Nondestructive generic data transformation pipelines

Building an ETL framework with abstract data access

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering & Internet Computing

eingereicht von

Jakob Petsovits
Matrikelnummer 0127236

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung:
Betreuer: A.Univ.-Prof. Dr. Josef Küng

Wien, 07.05.2009

(Unterschrift Verfasser) (Unterschrift Betreuer)

Declaration

Jakob Petsovits
Erlengasse 13
7312 Horitschon

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

_____ _____

(Ort, Datum) (Unterschrift)

Abstract

Data integration aims to translate data between different formats with minimal loss of information, often making use of mappings and schema information. ETL (''extract, transform, load'') offers powerful transformation capabilities by chaining input data through a user-defined series of operations; accurate preservation of data is only a secondary goal.

This work presents the creation of a generic data transformation framework for PHP/Drupal that combines ETL-like data flow with a focus on data preservation. In addition to pipelines comprised of operations, two aspects more commonly known from traditional data integration are also included as integral part of the framework.

Data wrappers expose an interface to access data which is stored in its original form, enabling load/modify/save workflows in addition to import/recompose/export. Schemas describe the exposed data structures, they improve type safety and enable generic mapping functionality. These capabilities imply a set of characteristics and challenges that are discussed and implemented.

Zusammenfassung

Datenintegration verfolgt die Umwandlung von Daten zwischen verschiedenen Formaten mit einem Minimum an Informationsverlust, oft unter Zuhilfenahme von Mappings und Schemata. ETL („extract, transform, load“) bietet reichhaltige Möglichkeiten zur Datentransformation mittels einer vordefinierten Folge von Operationen; Beibehaltung der Ausgangsdaten ist jedoch zweitrangig.

Die vorliegende Arbeit befasst sich mit der Entwicklung eines generischen Datentransformations-Frameworks auf PHP/Drupal-Basis, das den Datenfluss von ETL mit einem Augenmerk auf originalgetreue Erhaltung der Daten verbindet. Zusätzlich zu Pipelines, die sich aus Operationen zusammensetzen, werden zwei Aspekte aus der traditionellen Datenintegration als wesentliche Bestandteile des Frameworks verwendet.

Datenwrapper bieten vereinheitlichten Zugriff auf Daten, die intern in ihrer ursprünglichen Form gehalten werden. Dies ermöglicht nicht nur Import/Export-Datenflüsse, sondern auch das Laden, Verändern und Speichern existierender Daten. Schemata beschreiben die Struktur dieser Daten, wodurch Typsicherheit verbessert und generische Mapping-Funktionalität ermöglicht werden kann. Die Implikationen und Problemstellungen beider Elemente werden in der Arbeit untersucht und ein Prototyp implementiert.

Acknowledgements

I would like to dedicate this thesis to my parents, Christa and Gerhard, who guided me with a lot of understanding and perseverance throughout my life, and provided me with the opportunity to focus on my studies. The level of support that I experienced cannot be taken for granted, and I deeply appreciate that you never stopped believing in me. Thanks for everything!

I wish to acknowledge Klaus Furtmüller for realizing and promoting his ideas with energy and enthusiasm, and as a consequence, for hooking me up with this thesis. Kudos go out to my work mates, Wolfgang Ziegler and Matthias Hutterer, for their valuable input on scope, feasibility and possible directions for this project, with special thanks to Wolfgang for also reviewing this document.

I owe special thanks to Prof. Josef Küng for the supervision of this thesis. His quality feedback and administrative support proved very valuable, but I especially appreciate the emphasis and help on getting things done. I couldn't have wished for a better mentor to guide me in this undertaking.

To the Drupal community and open source folks in general, I extend my gratitude for making all of my work worthwhile – it's just so much more fun if there is someone out there who can put the results into productive use.

Contents

List of Figures

Chapter 1

Introduction

1.1 Motivation

Data portability between information sources on the Web and internal data stores is a key factor that many organizations face nowadays. Handling data which can be retrieved in a variety of data formats, each implying different schemas and semantics, is crucial to a large number of use cases. Important ones are connectivity with other organizations, evolution of data formats, data analysis supporting business decisions, and backup and recovery of content and configuration data.

Data integration, the challenge that covers most of these use cases, has been researched well, and there exist numerous approaches to solve data portability issues. Given the problem space, it is no surprise that most of the solutions cover the issue of how to translate a given set of data between data formats and data structures as correct, lossless and automated as possible. Schemas can assist and ease this process considerably, and have become a central element in data integration efforts.

While the data integration community has traditionally focused on preservation of the given data, efforts in data warehousing concentrate on transforming it to structures that can easily be queried, in order to facilitate analysis rather than interoperability. A central process in data warehousing is called ETL – ''extract, transform, load'' – and provides means to chain fine-grained transformations and other operations on the imported data. These operations usually act on primitive or simple data types, which promises high performance and makes it easy to mix and match single pieces in order to construct a complex rule set. It is however common practice for data warehouses to only store data

that is important for the subsequent analysis. Therefore, software in this domain mostly lacks in areas where traditional data integration techniques shine: schema mapping, coupled transformations, losslessness and, as a consequence, high-quality output in a given target format.

This thesis documents the creation of a generic data transformation framework that integrates concepts of both worlds, resulting in a new set of characteristics as it combines schema information and abstracted data storage with fine-grained operations on simple as well as complex data types. The resulting prototype has been implemented as extension (known as "module") to the open source CMS and web application framework Drupal, in order to prove the feasibility of the approach and to highlight issues that may arise when putting the ideas into production code.

1.2 Outline of the document

In chapter 2 we present related literature, highlighting various aspects of data integration research that are relevant for this work.

Chapter 3 shows the state of the art in its practical form by providing an overview of existing software in the areas of ETL and previous attempts at bringing data integration capabilities to Drupal. This chapter is completed with a survey of popular data formats, emphasizing their different traits with regards to data structure and schema handling.

In chapter 4, the concept is put into practice by first investigating a number of use cases and key requirements that the basic architecture needs to accomodate, taking the insights from the previous chapters into account. Based on these constraints, the framework's architecture is formulated and the design of the Drupal implementation is presented. Challenges and particularities in the process of implementing the prototype are outlined.

Chapter 5 then wraps up the results by detailing the traits of the architecture and the implementation, exposing strengths and weaknesses of the approach.

The thesis is concluded with chapter 6 which summarizes all the findings from the previous chapters, and provides a short outlook on future directions and challenges that this approach may take.

Chapter 2

Literature

2.1 Foundations of data integration

According to [Len02], data integration is "*the problem of combining data residing at different sources, and providing the user with a unified view of these data*". Research on data integration has largely focused on improving interoperability between applications and data formats, with special emphasis on conversions between relational, (semi-) structured and object-oriented data storage.

In order to make this happen, [BP96] has identified four steps that need to be tackled:

1. Acquire an abstract definition of the objects that are to be integrated while preserving as much of the original information as possible.
2. Match the data definitions to identify their commonalities and differences, i.e. finding common attributes of the source and target data structures and those attributes that are ignored in either of the two representations.
3. Make the source object structure structurally compatible with the target one by possible re-ordering and restructuring attributes, ignoring irrelevant attributes and adding those that are missing.
4. Ensure domain attributes compatibility, as the same piece of data can have different representations in the source and target structures – this step makes sure that the data element is represented in the format that is required for the target structure.

[BP96] then goes on to describe a data model that can contain a generalized representation of the original source data. Each element in such a data structure can be either an

atomic data type such as integer, float and string, or a constructed type that can contain further elements. The three kinds of constructed types are labelled tuples, sets, and (disjoint) unions of types, allowing to compose a tree structure of data elements. Based on this data model, a number of reversible structure rewriting rules are introduced that deal with step 3 from the above list.

[ACM97] presents a slightly different data model that addresses some of the shortcomings of the one in [BP96] and shows its applicability for representing relational, semi-structured and object-oriented data. In this model, data trees are constructed out of labelled vertexes, where the label describes the name of the element that it represents (for vertexes that contain children), or the data value itself (for leaf vertexes). Furthermore, each vertex is assigned a unique vertex identifier that makes it possible to represent references to other vertexes in the data tree – or data forest, as is employed by this paper. This data model also caters for the needs of ordered lists by explicitely capturing the order of each vertex's child elements, which is an important property for various data formats, including XML.

Correspondence between vertexes is described by a set of declarative predicates that cover mappings between the source and target structures as well as the actual transformation of data elements. Mappings are represented by *is(&1, &2)* predicates, whereas transformations are covered by any n-ary relations. The *concat(&12, &1, &2)* relation is given as example. Also, the *cons()* and *merge()* relations are introduced in order to support unbounded set and bag constructs which would not fit into the given minimal data model otherwise.

[MZ98] shows that the use of associated schemas for the source and target data structures can ease the specification and customization of data conversions, as opposed to the previously mentioned approaches which concerned themselves with conversions that are only based on the pure data. The idea is that, for a conversion that preserves the original data, the source and target schemas will resemble each other for the largest part, whereas their corresponding data representations are likely to exhibit much more differences. Consequently, the use of schemas on both sides of the translation enables schema matching – the automated generation of schema mappings. The authors employ a relatively simple semi-automatic matching algorithm that works sufficiently well to guess most mappings right; the user is asked to help on the ones that the algorithm cannot figure out on its own.

Building on the data model from [ACM97], the authors derive a similar model that

represents the corresponding schema, with only a few schema-related adaptions. For the schema model, they use a graph rather than a forest to simplify the description of recursive types, and the vertexes – now representing single schema elements, or ''types'' – are amended with additional information on the relationship between the element and its components. Quoting their paper, that information describes the following properties:

(1) whether the element in question is a root *type, i.e. whether roots of the data forest can be assigned this type, (2) what are the possible labels of data vertices of this type (for leaf vertices this will determine the possible domain of data values), (3) can a data vertex of this type be referenced by other vertices (i.e. the vertex id can be the value of some leaf node in the forest), (4) what is the allowed number (range) of children of a data vertex of this type, (5) are the children of a data vertex of this type ordered or not, (6) are some of the component types optional (this is useful for describing union types and optional attributes), (7) is the sub-tree rooted at a node of this type allowed to have arbitrary structure (useful to describe semi-structured data), and (8) whether vertices of this type actually appear in the data graph or are just ''virtual'', that is, describing a set of which no parent element actually exists.*

The matching algorithm then makes use of a repository of translation rules, one of which is assigned to each mapped data element. Each rule specifies which source and target types it can handle, and a fixed priority value. In case multiple rules apply to the same combination of source and target element types, the rule with the highest priority value is selected.

2.2 Wrapping data

The literature listed in the previous section assumes that the complete set of data is already available to the translation tools. However, especially with an increasing amount of data that is only available in the Web, that assumption might not always be true. As opposed to a complete data tree, the authors of [CHS+95] proclaimed as aim of their middleware system Garlic ''*to provide applications users with the benefits of a database with a schema (...), but without actually storing (at least the bulk of) the data within the Garlic system proper*''.

In their middleware architecture, they distinguish between "*a repository type, which is a particular kind of data management software, and a repository instance, which is a particular data collection that it manages*". Repository types are implemented by wrappers which encapsulate the source data and fulfill two tasks essentially: "*a) to describe the data in its repository and b) provide the mechanisms by which users and the Garlic middleware engine may retrieve that data.*" [HMN+99].

Wrappers provide a way to decouple the data integration system from the specific methods of retrieving the source data, and are able to present the available information in a different form than is available, e.g. by parsing web pages into a pre-defined structure. Garlic allows to further restructure the retrieved data into composite objects called *object views*, preserving the identity of the original objects by referencing those from each property in the object view. Both repository instances and object views can be queried with Garlic's extended SQL dialect.

To work on an incomplete set of data comes at the cost of giving up strong object identity and difficulties in handling legacy references, as integrity constraints and the possible volatility in the source repository gives reason not to store those permanently. Both aspects are detrimental to the ability of reliably resolving object references, which caused the authors to introduce the notion of "weak identity", "*denoting an object uniquely but not necessarily immutably within the scope of a Garlic database*". Legacy references on the other hand are simply not allowed to be stored as such persistently in the local data representation, and need to be worked around or resolved within the scope of a query.

[LRO96] makes an attempt to optimize remote queries by introducing *source descriptions* specifying the domain and range of values that can be retrieved from that source, as well as its query capabilities. Knowing the kind of data that can be expected, a query engine can restrict the number and extent of remote requests by constructing an optimized query plan.

An important property of sources that are denoted in this manner is the potentially incomplete set of results, i.e. a source might only contain a subset of all possible answers even if value domain, range and query capabilities match – the example given by the authors is a car dealer's web site that can look up cars for sale that have been manufactured after 1992 but even in that range can only retrieve a minor portion of the set of cars that would be available for sale worldwide. Furthermore, the query capabilities of an information source might also be restricted, so that the source for the

mentioned car dealer site can for example process a detailed query but cannot retrieve all cars in the site's database. Source description based systems must therefore assume to work on only a fraction of the whole.

The authors of [LRO96] tackle the problem with a data model that includes relations, classes and class hierarchies, as well as attributes (single-valued or multi-valued) that belong to a class. Given such a model, queries can be defined as set of predicates over the relations of their global schema called *world view*. The given example query looks as follows:

$$\begin{aligned} q(m,p,r) \quad \leftarrow \quad & CarForSale(c), Category(c, sportscar), Year(c,y), y \geq 1992, \\ & Price(c,p), Model(c,m), ProductReview(m,y,r) \end{aligned}$$

In the remainder of the paper, the authors refine this approach by providing means to describe possible contents and capabilities of information sources, query plans for accessing these, and means to optimize such query plans. Finally, answers for the query are retrieved by retrieving answers from the information sources, according to the query plan, from the respective wrappers (appearing under the name ''interface programs'' there).

2.3 Schema matching

Schema matching is the problem of creating a correlation between elements of source and target schemas, the result of which is a set of element mappings (called *alignment*, or more commonly just ''mapping'') that are further detailed through a confidence measure in the [0, 1] range and the assumed relation type of the correlation. Building on previous work from [RB01], Shvaiko and Euzenat [SE05] research the state of the art in schema matching by defining the matching problem, classification criteria for schema matchers and techniques that they employ, as well as reviewing existing matching systems. They note the similarity between schema and ontology matching, pointing out that the primary difference between those is only the presence of information about the meaning of data:

On the one side, schema matching is usually performed with the help of techniques trying to guess the meaning encoded in the schemas. On the other side, ontology matching

systems (primarily) try to exploit knowledge explicitly encoded in the ontologies. In real-world applications, schemas/ontologies usually have both well defined and obscure labels (terms), and contexts they occur, therefore, solutions from both problems would be mutually beneficial.

They distinguish between *elementary* matchers and *combinations* of matchers, the latter employing multiple elementary ones. *Instance-based* matchers are not further detailed in the survey; unlike *schema-based* matchers they rely on mostly statistical methods examining example data.

The main contribution of [SE05] is the classification of elementary schema-based matching approaches which the authors break apart into three layers. The *Granularity/Input Interpretation Layer* draws a line between *element-level* and *structure-level* match granularity, and then introduces a second split between *syntactic* (e.g. label name), *external* (e.g. predefined thesauri) and *semantic* (formal) interpretations of the input data, i.e. this layer defines itself though its approach to processing the given schema information. The *Kind of Input Layer* is concerned with which kind of data the algorithms work on, which according to the authors comprises *terminological* (strings), *structural* (structure) or *semantic* (models) data. They further distinguish between *string-based* and *linguistic* methods for terminological input, as well as between *internal* and *relational* considerations on structural input. Finally, the most fine-grained classification level is the *Basic Techniques Layer* which contains categories for each concrete method of interpreting the input information.

The survey also shows that the majority of the state of the art matching tools strongly relies on *syntactic* methods, with string-based and language-based approaches in particularly wide usage on the *element-level* side. *Structure-level syntactic* methods are more diverse on that matter. Moreover, schema matchers tend to use thesauri and dictionaries as *external* methods, whereas the same category is occupied by WordNet [MBF+90] for ontology matchers. Semantic methods are rarely employed.

2.4 ETL and data flows

ETL was conceived by the data warehouse community as a superset of the data cleaning problem, which according to [RD00] is closely related to data transformation and deserves similar treatment. These tasks (“T” for transformation) are preceded by fetching

the input data from external sources into a data staging area (“E” for extraction) and followed by storing the transformed data into the data warehouse (“L” for loading). Due to the focus on data warehouses, the data staging area and the transformation primitives performed on it are situated in the realm of (mostly relational) databases.

[VSGT03] defines a generic metamodel for ETL activities, specifying the elements that constitute an ETL framework. This model includes *types* (defined by a name and a value domain), *schemas* which consist of a flat list of name/type pairs called *attributes*, and *RecordSets* that are basically the concrete rows (value lists) corresponding to a schema. Furthermore, *functions* are specified as instances of *function types*, and along with schemas, attributes or constant values can be passed as parameters to an *Elementary Activity* which defines the actual transformative operation with a list of input and output slots and execution behavior. Lastly, the model contains a number of relationship types to capture the structure of the processed data in a relational way.

Deriving from this metamodel, [VSGT03] introduces the concept of a *Template Layer* and a *Schema Layer*, the latter being the concrete manifestation of an ETL scenario and the former describing a more specialized set of the above elements. In particular, Elementary Activities are classified into five groups of operation templates: *filters*, *unary* and *binary operations*, *file operations*, and *transfer operations*. Processing an ETL scenario essentially boils down to sequential processing of such operations on a number of RecordSets.

An attentive look reveals another kind of flow-based data transformation software that is very similar in concept to traditional ETL, though typically very different in its aims and use cases: the combination and re-purposing of data available from different web sources, termed “mashups” by the internet community. Popular use cases for mashups include customized RSS feeds or a Google map showing a custom set of locations or travel routes. Instead of transformations optimized for performance and flexibility, this kind of software often focuses on ease of use and improved presentation of existing data – a peculiarity that stems from the capabilities of web browsers and JavaScript as well as the target audience which for many tools includes non-commercial users.

[TAR07] identifies user interactions at runtime of a transformation job as further characteristic distinguishing mashups from data warehouse solutions, and proposes more advanced data integration capabilities for mashups. The architecture suggested there is composed of components for querying remote or local data sources, components such as `fuse`, `aggregate` or `union` for transforming query results, and a query engine

for creating and executing query plans (complete with a query/match cache that aids performance and iterative execution). XML is used as internal format, and the data sources and transformations are exposed through a custom scripting language.

[WH07] points out several issues that are important to consider when creating user interfaces for the transformation pipelines. The authors investigate the issue of usability of their mashup tool Marmite for end-users, based on the rather simple user interface of Apple Automator[1]. In a usability study of Automator, they identify difficulties in selecting operations and the lack of feedback during the creation of data flows as most severe problems for end-users. Their approach to solve the former is to unobtrusively suggest appropriate operations to the user, depending on the context such as the currently selected operation in the existing pipeline. The latter needs to be solved by enabling the user to execute test runs with output previews, preferably in an iterative manner that does not require the whole set of transformations to run when only single operations were updated. These findings were implemented in their mashup tool, which was found to work reasonably well apart from its shortage of operations (causing Marmite to fail for a few tasks when being used by less experienced users).

2.5 Further reading

In [HRO06], Halevy et. al digest the most significant activities in data integration research from 1996 to 2006, which provides a good starting point for diving deeper into the subject matter.

[1] Apple Inc., "Working with Automator": http://developer.apple.com/macosx/automator.html

Chapter 3

Technologies

3.1 Related software

Most papers mentioned above indicate the existence of an accompanying implementation, either as the actual motivation for the research work or as a proof of concept. The majority of these prototypes are not in wide usage though, some not being available to the public at all. In this section we investigate how a selection of related tools is approaching the problem space of data integration, in order to get a feel for how previous research has affected production software.

Schema matching tools are not covered in this section as they are hardly present outside of the research world – most advanced ETL and data integration tools include schema matching capabilities but remain with simple (mainly string-based) solutions as those are good enough for common use cases like database and XML mapping suggestions.

3.1.1 ETL and mashup tools

Driven by the need for high performance, production ETL software exhibits a strong focus on database operations. A common property of such tools is that the available transformations operate on a set of rows and columns like records found in traditional RDBMS, and that the user can specify the data flow with pipelines, often directly editable through a user interface that presents operations and flows on a canvas.

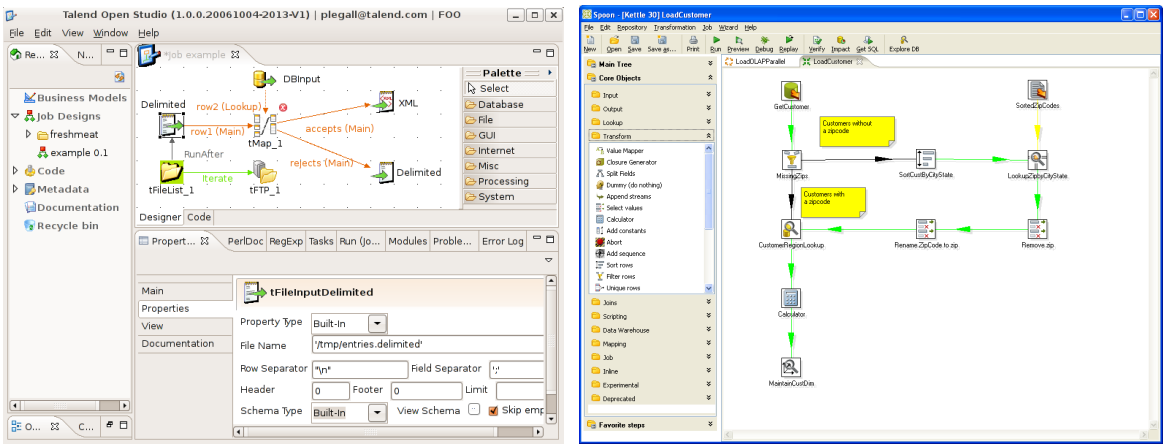
Talend Open Studio

Talend Open Studio[1] is an ETL tool written in Java, with an Eclipse-based transformation workflow designer. The system builds on a collection of ''jobs'', each job containing a set of components which represent transformations and other data operations. A large number of components is shipped, including various input/output components for files and databases, file transport over various protocols, custom code operations (written in Java or Perl), filters, joins and aggregation operations, schema mapping, control flow components like ''foreach'' loops and termination conditions, data replication and combination, and also a component that encapsulates other jobs.

The components are connected with arrows specifying the flow of the operation. The data flow is outlined by the ''Main'' set of connector arrows, more data can be added by intersecting this flow with ''Lookup'' sources. Schema information is being used for each component, although due to the record-based data model, schemas cannot contain nested elements. Nesting can be achieved by concatenation/split operations, but are then only viewed as raw data without related schema information. For execution of the jobs, Talend Open Studio generates Java or Perl code (depending on the project type which also affects the set of available components) that can either be run and debugged inside the Eclipse environment or exported to a self-sufficient executable file, deployment on servers being the main use case of the latter.

Internally, components are plugins that provide a set of properties specifying possible input and output connectors as well as its parameters. Also, a component consists of three code templates which are called in different phases, which in combination generates the code that is necessary to perform the transformation. The ''begin'' template contains code to initialize any required data if necessary. The ''main'' template contains code that will be executed each time a row is being processed. Finally, the ''end'' template cleans up its data, e.g. by closing file handles or database connections. Talend Open Studio will generate the code in a way that the ''begin'' code is called in reverse order of component execution, whereas the ''main'' and ''end'' parts are called in their normal execution order.

[1] Talend: http://www.talend.com, inspected version of Talend Open Studio: 3.0.0RC1 (http://www.talend.com/downloads/download.php?version=v300rc1a)



(a) Talend Open Studio (b) Pentaho Data Integration

Figure 3.1: ETL tools

Pentaho Data Integration

Pentaho Data Integration[2] (formerly Kettle), or PDI in short, is another high-profile ETL tool, with a lot of similarities to Talend Open Studio. Even though the operations are called ''steps'' instead of components and the connector arrows are ''hops'', the architecture of the two tools is very similar. (As is the large collection of available data operation plugins.)

There is important difference though: unlike the single ''job'' specification that is used in Talend Open Studio for all kinds of data flows, PDI makes a distinction between ''jobs'' and a more granular kind of operation, called ''transformations''. A job in PDI is used to specify the high-level flow by combining transformations and connecting those to their input sources and output destinations, like FTP transfers or sending emails according to transformation results. Jobs are also responsible for data transformations that do not happen on a row level, like XSL transformations. The ''transformation'' component on the other hand takes care of low-level row-based data operations like selecting, filtering or sorting the input rows. Transformations can also include other transformations (but not jobs), whereas loops are only supported in jobs but not in transformations. According to PDI developer Roland Bouman[3], this distinction was made

[2] Pentaho Data Integration: http://kettle.pentaho.org, inspected version: 3.1.0 RC1 (http://downloads.sourceforge.net/pentaho/pdi-open-3.1.0-RC1.zip)

[3] Roland Bouman on the distinction between jobs and transformations: http://rpbouman.blogspot.com/2006/06/pentaho-data-integration-kettle-turns.html#c1621331275290593029

primarily for performance reasons as the data and flow constraints that come with ''transformations'' enable asynchronous execution of steps.

Another architectural difference compared to Talend Open Studio is the execution of jobs: PDI does not generate code but executes jobs natively in its job executor engine. The workflow modeller exports job and transformation descriptions that the job executor is able to run on a server without user interface dependencies.

Yahoo! Pipes

Yahoo! Pipes [4][5] is an advanced tool for creating mashups and other data transformations. In fact, Pipes' approach is very similar to the ETL tools mentioned above, as it also features a library of components (here called ''modules'') with various input and output parameters, a canvas for assembling and configuring the modules, and a debugger that shows a real-time preview for the results of the module that is currently selected in the canvas. The most obvious difference to traditional ETL tools is the focus on web data sources as opposed to database sources, which certainly influences the target user segment a lot. (Despite the different focus, Pipes can import and process file data like CSV, XML or JSON just as well, but even those must be fetched from the web.)

Each input and output parameter is specified for one of the predefined types that Pipes provides[6]: ''items'', ''location'', ''number'', ''text'', ''datetime'' or ''url''. The ''items'' data type represents lists of structured data types, whereas all others are atomic variables that can be extracted from data of the ''items'' type or entered by means of a manual input field. It seems[7] that Pipes stores ''items'' data in a tree structure, so operator modules such as ''Sub-Element'' or ''Unique'' can operate on elements that are deeply nested in the data structure of the top-level elements. There is no indication of any usage of schemas related to these data structures, though.

DERI Galway has created open source software called Semantic Web Pipes [MPP+08], which is inspired by Yahoo! Pipes but drops the notion of multiple types in favor of an RDF-only approach, enabling operations like SPARQL queries instead of mashups for all the non-semantic data that Yahoo! Pipes supports.

[4] Yahoo! Pipes: http://pipes.yahoo.com
[5] Screenshot by Kiet Callies: http://www.flickr.com/photos/kietcallies/2039894398/, published under the CC BY-NC 2.0 license.
[6] Data type reference for Yahoo! Pipes: http://pipes.yahoo.com/pipes/docs?doc=data_types
[7] Being based on closed source code, one can only guess about the inner workings of Yahoo! Pipes.

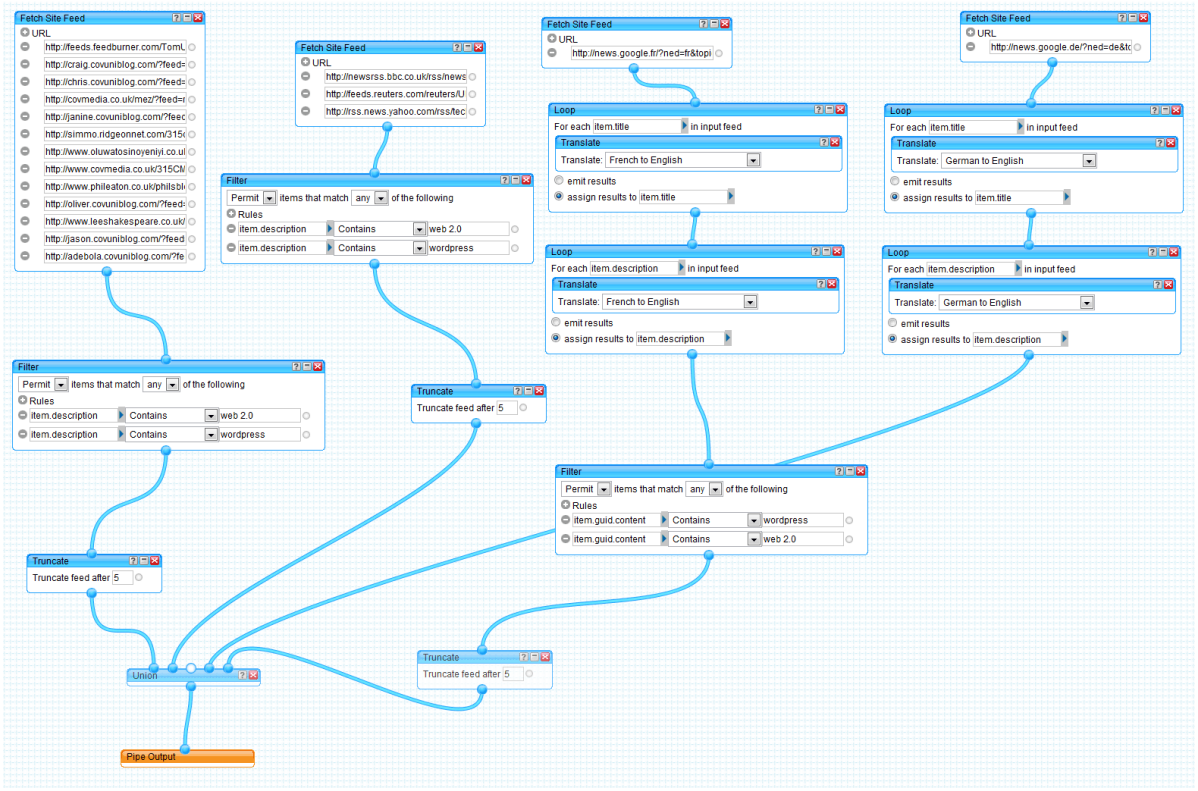


Figure 3.2: Yahoo! Pipes

3.1.2 Drupal modules

In order for the prototype to fit into its target software ecosystem, existing software in that area has to be considered.

Drupal[8] is a content management system and web application framework that is flexible enough to power a variety of web sites, such as blogs, news sites, social networks, and intranet / project management solutions. The base distribution, called Drupal core, contains only the most widely used features and is maintained using strict quality assurance practices. Through its hook system, Drupal core provides means for so-called modules to adapt and extend the existing functionality [Van08]. The *contrib* repository[9] on drupal.org hosts a vast number of such modules.

Though import/export functionality is recognized as one of Drupal's problem spaces in need of a better solution, there have been a number of attempts to create focused

[8] Drupal: `http://drupal.org`
[9] Drupal contributed modules: `http://drupal.org/project/Modules`

solutions for certain aspects of the data transformation and conversion issue. As of this writing, the 6.11 version[10] of Drupal core is the latest stable release and used as a basis for comparisons, except where indicated otherwise.

Drupal core

In addition to plain XHTML output produced by standard page callbacks, Drupal offers a way to render structured content by passing PHP arrays to the `drupal_render()` function, which is mainly used through Drupal's Forms API functionality. It allows to recursively render any given structure by calling the appropriate theme function for each element, which translates the structured element to plain XHTML. Starting with the leaf elements, `drupal_render()` will therefore flatten the nested structure to a simple output string.

A current shortcoming in the is the limited applicability of this functionality. Drupal's main content entities, ''nodes'', provide their contents in an unstructured form even before the rendering takes place, so extension modules with the intent to alter the rendering of node contents can not perform any transformations except for string operations. This will be fixed in Drupal 7 (the upcoming feature release, planned for late 2009) where the `hook_page_alter()` patch[11] was already applied; this patch defers the flattening of structured node contents to strings so that structural changes to rendered contents become possible.

Another downside of the current framework is that the final output will always appear in XHTML format[12]. The ''Abstracted Web Services API'' patch[13] promises to fix these issues by switching output renderers depending on the requested format. Once this patch gets included, Drupal will be able to output more data formats, which creates new opportunities for data export.

On the import side, Drupal core ships with the Blog API and Aggregator modules. Blog API offers XML-RPC interfaces for the Blogger, Movable Type and MetaWeblog APIs – which works well but naturally depends on the remote host to provide the data in the given form, so this module is not suited for importing other types of data than blog

[10] Drupal 6.11: `http://drupal.org/node/449114`
[11] drupal.org issue ''hook_page_alter()'': `http://drupal.org/node/351235`
[12] Actually, Drupal allows nodes to be rendered to RSS too, but that functionality is hardcoded so it doesn't allow modules to provide support for further output formats other than XHTML and RSS.
[13] drupal.org issue ''Abstracted Web Services API'': `http://drupal.org/node/145551`

posts. Blog API then stores the received data as a node with basic properties like title, body and author. Aggregator on the other hand stores the RSS feeds that it parses in an internal data store, which means that the resulting feed items won't be able to make use of the extensibility that Drupal provides for node content. Both Blog API and Aggregator are simple fixed-function data converters from an architecture point of view, and of limited interest for more generic data integration purposes.

CCK and "Fields in core"

Without further extensions, Drupal core only provides basic fields for its node content: there are node title, body, author and creation time that are rendered to XHTML, and some options for meta-information like node visibility, menu position or commenting features. A few modules in Drupal core, such as Upload or Taxonomy, allow to add more information to a node. Still, these additional fields are hardcoded and appear at a fixed position.

With the CCK module[14], it is possible to flexibly construct the structure of nodes out of the original node properties and additional fields that can be mixed and matched in any order, separately for each content type. Field functionality is provided by further modules and consists of components to enter (''widgets''), display (''formatters'') and manage the data (''fields''). Widgets provide appropriate form elements for the given field type and formatters render the field data to XHTML. The interesting point is that each of these components are independent from each other, a field can be assigned any compatible widget or formatter (those can also be provided by yet another set of modules).

Because of the large percentage of Drupal sites using this module, any tool for data integration with Drupal must be able to support CCK fields. Naturally, that involves dealing with potentially unknown field types so it must be possible to handle those with pluggable integration functionality. This is also where the limits of the Blog API and Aggregator modules in Drupal core become apparent, as neither of these knows about these or provides sufficient interfaces to extend their functionality for CCK fields.

CCK has traditionally lacked a usable public API so programmatical configuration of CCK fields and data exchange was hard to achieve in versions for Drupal 5 and earlier.

[14] CCK (Content Construction Kit): http://drupal.org/project/cck, inspected version: 6.x-2.0-rc7 (http://drupal.org/node/306274)

The Drupal 6 version has started to tackle this issue, whereas a complete API is worked on for Drupal 7 by means of the "Fields in core" initiative[15] which aims to move CCK field functionality to Drupal core. By converting hardcoded node fields to CCK's field format, it also bridges the gap between the different kinds of data that make up a Drupal node.

Views

According to Dries Buytaert[16], the Views module[17] is the most widely used of all contributed modules for Drupal. It provides listings for nodes and other entities, and can sort and filter these listings using conditions on an entity's fields. This functionality is built on top of an SQL query builder, and other modules can provide their own fields, filters and sorting mechanisms by specifying their database tables and operations to be executed on it.

The output format of the listing is determined by "output styles" that can be configured for each view[18]. While fields are rendered by the modules that specify the field's database relation, output styles are pluggable and can be provided by any module. CCK is integrated with Views and offers any of its formatters as renderer for the fields specified by CCK.

By using fields with appropriate CCK formatters, a Views style could output its listings in a wholly different data format than the originally intended XHTML while remaining fully configurable by the site administrator – in fact, Views already provides an RSS output style, and the Views Datasource module[19] provides output styles for various data formats based on XML, JSON, RDF and XHTML[20]. However, the flat hierarchies of fields in a node is an obstacle that makes Views outputs unsuited for the flexible construction

[15] drupal.org issue "Fields in core": http://drupal.org/node/265604,
Field API tasks for Drupal 7: http://drupal.org/community-initiatives/drupal-core/fields
[16] Dries Buytaert, "State of Drupal" keynote at Drupalcon Szeged 2008, 27.08.2008 - slides available at
http://szeged2008.drupalcon.org/files/drupalcon-szeged-dries-buytaert.pdf
[17] Views: http://drupal.org/project/views,
inspected version: 6.x-2.0-rc3 (http://drupal.org/node/309397)
[18] "view" is the Views module's term for a listing and its configuration.
[19] Views Datasource: http://drupal.org/project/views_datasource,
inspected version: 6.x-1.0-alpha2 (http://drupal.org/node/280239)
[20] More specifically, Views Datasource 6.x-1.0-alpha2 supports raw XML, OPML, generic and Simile/Exhibit JSON, FOAF, and the hCard microformat.

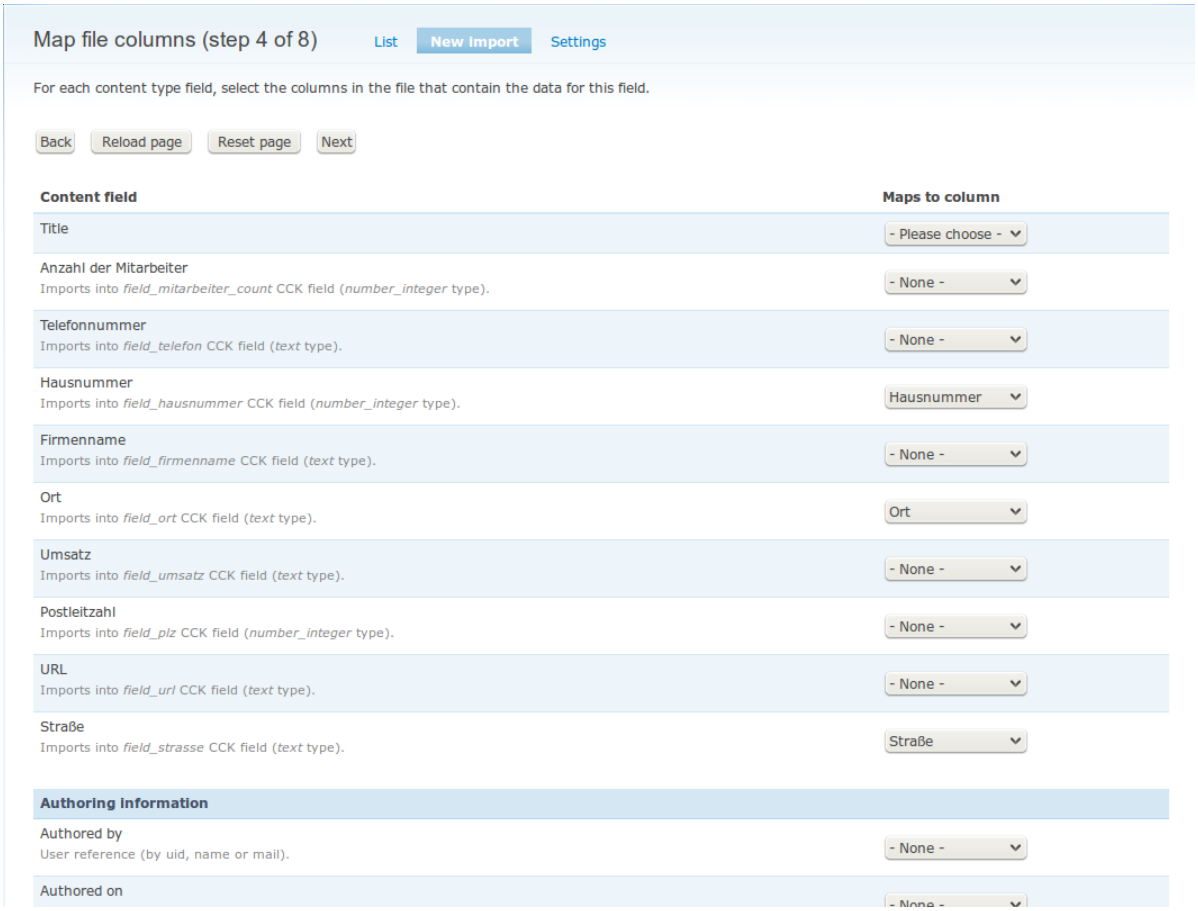


Figure 3.3: Mapping CSV columns to Drupal nodes with Node Import

of nested content, which is be a requirement for exporting more complex structured or semi-structured data formats.

Node Import

Node Import[21] is a conceptually simple module for creating a node out of each line in a Comma Separated Values (CSV) or Tab Separated Values (TSV) file. It relies on Drupal's hook system to determine supported content types and the fields of each content type.

When importing nodes, the user is required to (manually) upload the CSV/TSV file which is then intermediately stored on the server. The module then parses the first line which

[21] Node Import: http://drupal.org/project/node_import, inspected version: 6.x-1.0-rc4 (http://drupal.org/node/383992)

it interprets as column header containing the field titles for the imported data structure. Using a naïve matching algorithm purely based on string equivalence of the field names on each side, Node Import guesses a mapping between the fields in the imported file and those in the content type. After the user sets global and field options, each field is processed by the appropriate plugin that transforms the given string into its field representation in the Drupal node array, and each resulting node is stored after parsing all the fields this way. Plugins can even perform simple data cleaning by filtering out a node if the fields in the CVS/TSV file don't conform to a given condition.

Importing files and images (into the Upload and Image modules' internal structures, respectively) is possible, under the restriction that the referenced files are already stored in a predefined location on the server. (So Node Import does not perform any file transactions by itself.) Also, the module is only able to write a single data element for each row, so for multi-value CCK fields, Node Import will only fill in the first value and leave out possible other ones.

The Node Import module has not yet been ported to Drupal 6 and, at the time of writing, is only available in versions for Drupal 5 and earlier.

Feed aggregation, contd.

Due to the limitations of the Aggregator module in Drupal core[22], a plethora of replacement modules have been created for that purpose[23], each implementing certain aspects of the desired functionality. The "Pluggable architecture for aggregator.module" patch[24] that will be part of Drupal 7 obsoletes the duplication of functionality by making the Aggregator module extensible enough for further modules to provide their own features on top of it.

[22] The Aggregator module in Drupal 6 has shortcomings in the areas of performance and extensibility for input formats and transport methods. Further issues arise because Aggregator does not create feed items as nodes, which prevents support for more advanced functionality like integration with the Taxonomy and CCK modules, usage of comments, or fine-grained control of workflows.

[23] Aggregator replacements: Feedparser (`http://drupal.org/project/feedparser`), Planet (`http://drupal.org/project/planet`), Aggregator2 (`http://drupal.org/project/aggregator2`), SimpleFeed (`http://drupal.org/project/simplefeed`), Aggregation (`http://drupal.org/project/aggregation`), Leech (`http://drupal.org/project/leech`), and FeedAPI (`http://drupal.org/project/feedapi`).

[24] drupal.org issue "Aggregator rework": `http://drupal.org/node/303930`

An interesting point is that this patch separates feed import into separate ''fetcher'', ''parser'' and ''processor'' tasks: the fetcher is responsible for downloading the feed into a string variable, the parser transforms it into Aggregator's internal array format, and an arbitrary number of processors can afterwards act on the parsed data, e.g. by storing them as nodes or in a private database table. This architecture allows for flexible import facilities not only for regular feeds but also for CSV or XML imports.

However, the scope of this solution is limited by the common interface that is required to make fetchers, parsers and processors work together: storing all fetched data in a string variable poses memory limits to the size of the input data, and the number of possible imported fields is constrained to a set of pre-defined properties: ''title'', ''description'', ''timestamp'', ''author'', ''guid'', and ''link''. The Aggregator module is therefore still tailored to a certain kind of content, and will require hacks or workarounds to make it suitable for other use cases.

Import / Export API

The Import / Export API module[25], unmaintained since April 2007 and only available for Drupal 5, attempted to provide a generic solution for Drupal's lack of data conversion capabilities. It incorporates many elements commonly found in data integration tools: separation of schemas and data, support for nested structures, field definitions using types and references, (admittably hackish) schema mappings, and pluggable extensibility for input/output engines, field types and schema definitions. Input engines are responsible for transforming data from their internal sources to a PHP array structure given a schema and possible engine options, whereas output engines write the data back from that array structure to the target format.

The module is believed by the Drupal community[26] to have failed for its overreaching scope and complexity. In addition, maintenance was made more difficult by the fact that all data sources have been accumulated in a single place, so porting the module to a new Drupal version is one large undertaking instead of a collection of bite-size tasks.

[25] Import / Export API: http://drupal.org/project/importexportapi, inspected version: 5.x-1.1 (http://drupal.org/node/137925)

[26] Well-known Drupal developers on the futility of the Import / Export API: http://lists.drupal.org/pipermail/development/2008-January/028593.html, http://groups.drupal.org/node/10042#comment-31823

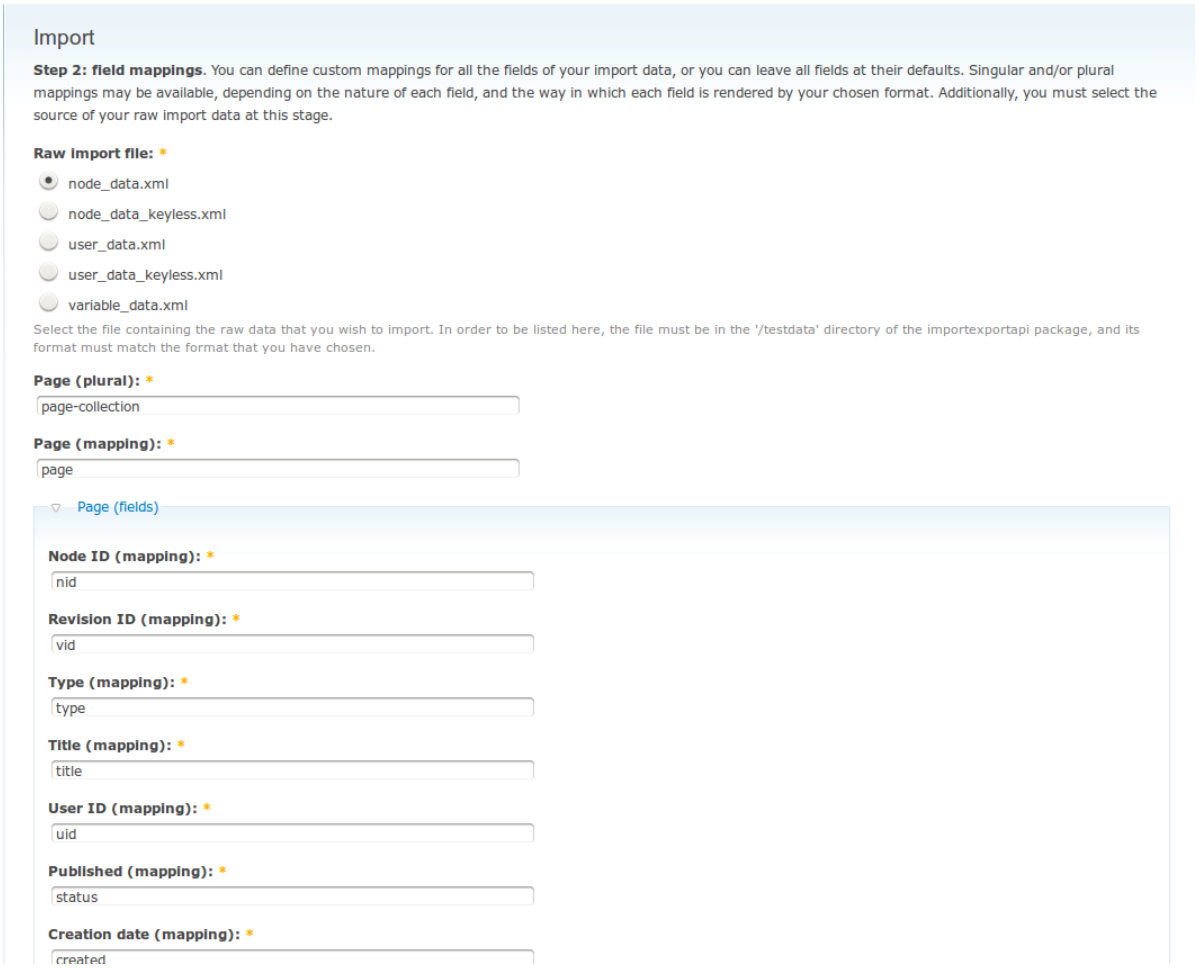


Figure 3.4: Mapping XML elements to Drupal nodes with Import/Export API

RDF modules

As RDF is slowly gaining acceptance as utilizable data sources such as DBpedia [ABL+08] or Open Calais[27] appear, the relatively new RDF[28] and SPARQL modules[29] gained quite a bit of momentum in the Drupal community, culminating in a prominent mention in the "State of Drupal" keynote at Drupalcon Boston in March 2008[30]. Since then, a community initiative has been working on bringing more RDF features to Drupal

[27] Open Calais by Reuters: `http://www.opencalais.com`
[28] RDF module: `http://drupal.org/project/rdf`, inspected version: 6.x-1.0-alpha7 `http://drupal.org/node/414010`
[29] SPARQL module: `http://drupal.org/project/sparql`, inspected version: 6.x-1.0-alpha1 `http://drupal.org/node/267443`
[30] Dries Buytaert, "State of Drupal" keynote at Drupalcon Boston 2008, 03.03.2008 - video available at `http://www.archive.org/details/DrupalconBoston2008-TheStateOfDrupal`

7 and beyond[31], starting with semantic annotations of Drupal content with RDFa but not yet incorporating full RDF repository capabilities into Drupal core.

The RDF module is a mere abstraction of the RDF storage layer (for RDF triplestores and textual exchange formats) with the associated API to access and query triplets. The module shows no serious shortcomings in itself, so its suitability for usage in data integration scenarios mostly depends on that of RDF itself. Shipping with a number of capabilities for import and RDFS-based export as well as for SPARQL queries, this set of modules might well be considered as a basis for further developments.

Configuration management

The modules mentioned in the above sections are all focused on managing content, however configuration management also plays an important role in import/export. With "install profiles", Drupal core offers a mechanism for setting up web sites when they are initially created. Basically, an install profile is just a PHP script that calls a number of functions to define default configurations and possibly create content too. The Install Profile API module[32] simplifies this task by providing a set of helper functions for reducing the amount of PHP code in an install profile and creating it in a more structured way. Still, a major shortcoming of install profiles remains as the lack of configuration export features makes it necessary to create install profiles manually.

The Deployment family of modules[33] attacks a different aspect of configuration management, namely the issue of staging, or how to move configuration from a test server to the production server and contents back to the test server from there. For that purpose, Deployment allows administrators to create multiple deployment plans specifying which data should be transferred to another Drupal installation. The set of supported data is extensible via modules providing a "Deployment implementer", and each implementer can provide options on which data exactly will be deployed in the respective deployment plan. When the items are actually deployed, the module sends all data from its implementer plugins to the target server over XML-RPC.

[31] groups.drupal.org group "Semantic Web": http://groups.drupal.org/semantic-web
[32] Install Profile API module: http://drupal.org/project/install_profile_api, inspected version: 6.x-1.2-beta1 (http://drupal.org/node/312592)
[33] Deployment modules: http://drupal.org/project/deploy, inspected version: deploy 5.x-1.0-beta2 (http://drupal.org/node/273143)

With the "port" module[34], an effort has been started to base both of these modules on a common foundation as they act on the same data and show partial overlaps in their feature sets. The central idea of port is that modules provide symmetric import and export functions for individual pieces of configuration and content, such as enabled modules, menu structure or nodes of a given content type. This data can be exported to text as serialized PHP, or to executable PHP code which on execution calls the import functions again. In general, port is suited well for a wide range of import/export tasks as long as no advanced transformation or schema handling capabilities are required; however, its development seems to have stalled since the original proposal was posted.

3.2 Data formats

In order to understand the traits of the data and schema models that the transformation framework should support, the following pages disseminate the concrete data formats together with their schema definition languages and a selection of sub-formats.

3.2.1 Delimiter Separated Values

Delimiter Separated Values (DSV) is the term for a family of two-dimensional array data formats, i.e. a DSV file can contain data for a single table with an arbitrary number of rows and columns. The format does not specify any semantics, and comes with only basic syntax rules. There are two delimiter characters – one to delimit fields in a row, and one to delimit the rows themselves – which can be assigned any character in theory. In practice, newline characters are used consistently as row delimiter, and the comma is the most common character for delimiting fields. The Comma Separated Values (CSV) format has been retroactively defined by the IETF as RFC 4180 [Sha05], though other delimiter characters such as tabs, semicolons or colons are in use as well. In a data integration scenario, the user should be able to specify the delimiter that is actually used. CSV is often used as term for the whole range of DSV formats.

DSV files do not have any schema associated. The first row is sometimes used to specify column names (in which case the actual data is starting from the second row on), but that practice is optional, thus any DSV parser must ask the user if such a "header row"

[34] Port module proposal: http://groups.drupal.org/node/14395

exists. A schema importer could take the same data file and import the first row into a flat schema, or at least determine the number of columns if no schema row is present.

3.2.2 XML

XML has been standardized[35] by the W3C and is a format that can contain any form of structured or semi-structured data. It is a foundation to create specialized formats, called XML applications, with clearly defined semantics and a subset of allowed elements.

An XML file consists of a tree structure of elements that can contain pure (text) data, child elements or both (''mixed data''). In a mixed data element, data and child elements are not separate, instead the child elements are embedded in the text. Further (meta-)data can be provided by element attributes, those are always given as flat data without nesting capabilities. Tag names are case sensitive.

Order is an inherent property of the tree structure, whereas attributes in a given element are unsorted. Identifier attributes (''`<a id="&xyz1">`'') can be used in other elements as reference to a specific element (e.g. ''`<b aref="&xyz1">`''). XML data can be navigated and queried with XPath or XQuery, the former often being used inside an XML document itself for element references in attributes instead of id specifications.

Schemas

Several schema definition languages are available to define constraints for structure and content of XML data. If the data conforms to such a schema then it said to be *valid*, as opposed to the property *well-formed* which only confirms that the syntactical rules are being followed.

The original approach to schemas for XML was a DTD, or Document Type Definition. While being included in the XML 1.0 standard already, DTDs lack support for XML namespaces and are not specific enough to define certain constraints such as data types and numbers for minimum/maximum element counts. It is possible to define elements by specifying allowed child elements that can appear in single or arbitrary amounts, compulsory or optional. The ''`#PCDATA`'' token is used to specify (parsed) text data. DTDs also allow to define new entity types to be decoded on parsing, similar to the

[35] Links to various XML working groups and standards by the W3C: `http://www.w3.org/XML/`

pre-defined ‘‘`<`’’ or ‘‘`&`’’ entities. DTDs are supported by many XML parsers, including PHP’s SimpleXML.

W3C XML Schema fixes most of the shortcomings of DTDs, and allows for greater specificity. Support for XML namespaces makes it possible to place different constraints on elements with the same tag (but different namespace), also, XML Schema allows to specify cardinalities with an additional restriction for the number of allowed elements in an n, m range. Support for data types enables stricter control over the textual contents of elements and attributes, and the specification for XML Schema defines a wide range of predefined types such as strings, integers or even lists. New types can be created, even with the help of regular expressions for string types. XML Schema also allows for default attribute values, which allows more flexibility but on the other hand has the negative effect that parsers without XML Schema support will potentially read incomplete data. Both XML Schema and RELAX NG are modular enough to allow a schema to be split into multiple files.

RELAX NG [CM01] has been created for improved readability over XML Schema and offers a very similar feature set with few semantic differences. Most notably, with RELAX NG it is possible to specify element structure depending on the concrete value of an attribute. Additionally, the ability to parse non-deterministic content models is powerful but a challenge for the validating parser to implement as look-ahead parsing must be supported.

Schematron is a less popular XML schema definition language and asserts that a series of XPath statements evaluate to `true`. It is unwieldy for tree structure definitions and hard to import into traditional abstract schema models. That, in combination with the fact that it’s mostly being used in combination with RELAX NG, makes Schematron less interesting for the purposes of this work, except for the goal that external schema validators should be able to interface with the transformation framework.

Example format 2: RSS

RSS is an XML format for describing content sources called ‘‘channels’’ and a number of posts or teasers belonging to a channel. RSS is actually a family of formats, with two branches progressing into different directions. RSS 0.90 and the 1.0 [BDBD+00] are in fact XML serializations of RDF data, making use of namespaces, RDF resource attributes and other related concepts. RSS 0.91 to 0.94 as well as RSS 2.0 [Boa07] leave

out namespaces except for unspecified extensions and also does not incorporate other internet standards, for the sake of simplicity. According to syndic8.com[36], the versions in active use today are mostly 2.0, 1.0 and 0.91, with other versions totaling to less than 2%. RSS 2.0 is backwards compatible to 0.91 so in practice, two different versions need to be supported by an RSS feed reader.

When processing feeds, a transformation framework should be able to tell apart the different versions based on name and attributes of the document element (e.g. "`<rss>`" in RSS 2.0 and "`<rdf:RDF>`" in RSS 1.0, plus "version" or namespace attributes respectively). Channel elements include "`<title>`", "`<link>`" and "`<description>`" elements as well as a series of optional data like "`<language>`" or "`<image>`". The actual contents are contained in a series of "`<item>`" elements which RSS 1.0 references through RDF references whereas RSS 2.0 contains these elements directly as children of the "`<channel>`" element. There are further commonalities and differences like the above, such as the date of item creation which is specified as "`<pubDate>`" in RSS 2.0 but as Dublin Core [Dub03] metadata "`<dc:date>`" in RSS 1.0.

A transformation framework should allow to convert one version of the RSS format into another one after determining its version. In order to do that, known different elements need to be renamed and their namespace changed. Furthermore, elements unknown to the schema (which are allowed in both branches of RSS formats, e.g. further Dublin Core metadata) must be preserved during that translation. For repeated execution, duplicate filtering capabilities are required so that imported feeds can be compared with existing ones on the basis of their global unique identifier or, alternatively, on the combination of author and creation date of an item. The remaining feed items then need to be merged into the existing list, which might be a representation in the database schema or as composite XML file containing a collection of feed channels and items. It should be possible to merge only known, predefined elements or all elements including previously unknown ones.

When importing the data into an integrated data store, some elements will need to be transformed from their XML representation into the target format. For example, elements specifying dates and author names might be mapped to Unix timestamps and Drupal users when being imported as Drupal nodes, while files could be downloaded from the given URLs and imported into the local file directory with a corresponding database entry, associated to a CCK file field.

[36] syndic8.com RSS version statistics: `http://www.syndic8.com/stats.php?Section=rss`

A schema importer should be able to take a feed URL, determine the RSS version, and provide a schema template with both required fields and optional fields that exist in the feed data that is downloaded from the given URL.

Example format 3: HR-XML SEP & German Standard CV

HR-XML SEP (Staffing Exchange Protocol) [BC07] is a classical example for an industrial data exchange standard. Specified with exactly defined semantics that are available in machine-readable form as interrelated collection of XML Schema files, this XML format contains a deeply nested structure of mostly optional elements describing a candidate for a job position and his or her history, availability information and other data that is relevant to job applications.

The largest challenges for transforming SEP information into other data representations relate to mapping and manageability aspects of specifying the transformation. As many elements are being nested and reused in the source schema, it would be desirable for a transformation framework to support schemas and transformation rules to be composed out of smaller building blocks, thus reducing complexity and maintenance efforts especially when several different transformation rulesets are based on the same schema.

Another point is the large set of possible data, which is unlikely to correspond with an internal schema. When transforming from SEP, many elements will likely be ignored and dropped during the transformation. Parts of the XML data will refer to objects that are already present in the system, it must be possible to find such correspondences and update the rest of the data to reference internal objects instead of child elements. When transforming to SEP, required data elements might be missing, these should be able to be generated on the fly using assumptions that are inherent to the system but not present as actual data. (For example, a web site for Austrian job applicants can assume "`de-AT`" as default locale even if the user has not specified one.)

Broad specifications are often adapted to more specific needs – that is not only the case with XML itself but also with HR-XML SEP which is being incorporated into a standard targeted to German job candidate exchange needs. The resulting specification is called German Standard CV[37], or GSCV. It is compatible with SEP but imposes additional

[37] German Standard CV home page: `http://german-standard-cv.de`

restrictions on the contents such as a limited set of allowed elements or a strictly defined list of possible values for skills, industries the candidate might have worked in, or educational institutions to be attended. The Information Manifold system presented in [LRO96] handles this challenge by introducing derived schemas that are merely linked to the broader one and thus change whenever the original schema changes.

3.2.3 RDF

Like XML, RDF[38] is also a technology specified by the W3C. Unlike XML though, RDF is an abstract data model with a focus on semantics instead of syntax. RDF data can be represented in various data formats, XML being one of them.

The fundamental concept of RDF is to represent all data on the Web as a directed graph. The nodes in such a graph represent any entities and are being used as ''subject'' and/or ''object'' in an RDF statement, while the edges denote the relation (the verb, or ''predicate'') between subject and object of a single statement. Almost any kind of data can be represented with a set of RDF statements, including properties of an entity, entity containers such as ordered (''`rdf:Seq`'') and unordered (''`rdf:Bag`'') lists, and generally relations to any other entity. Subjects, verbs and objects are preferably referred to by a URI that might or might not exist as actual location on the Web. Nodes can also contain literal values instead of URI content, or they might not contain any contents at all – in that case, such a node represents an unnamed, intermediate pointer to other nodes and is called a blank node. Node content can be typed, and RDF reuses the datatypes defined by the XML Schema specification [BM04] except where these datatypes don't conform to RDF's requirements of a datatype[39].

It is important to note that, due to the data model, relations should be regarded as links. For example, this property manifests when an instance of a structured type refers to other instances of the same type - those are not contained in the former as actual attributes but as pointers. Another example of this characteristic are lists, not being represented as actual collection of items next to each other but as linked lists in a recursively-nesting structure. For a transformation framework, the point to take away is that the meaning of data will not necessarily match its physical representation, and

[38] Links to various RDF resources and specifications: `http://www.w3.org/RDF/`
[39] RDF datatype requirements and supported XML Schema datatypes: `http://www.w3.org/TR/rdf-mt/#DTYPEINTERP`

similar representations of data can convey different meanings. Also, it must be possible to break apart schemas into modular parts so that one part can contain not only other elements but also references to objects of the same type.

Whether a URI represents a link to other information or an abstract concept is not inherent to the data model, the correct interpretation depends on the software's knowledge of the intended semantics. In order to clarify and unify the representation of similar data, the RDF community is busy defining schemas and ontologies for well-defined subsets of the infinite scope of recordable knowledge. The use of unified ontologies also enables queries involving multiple participants on the web, each one providing their own bits of knowledge from RDF data stores (called "triplestores") exposed to SPARQL *endpoints*. RDF is subsequently an ideal way to perform retrieval of and reasoning over data that is clustered all over the web. SPARQL, the query language for RDF data, enables this kind of data processing by asking for variables that are related to other nodes in a certain kind of way. The peculiarity of this approach is that the creator of a SPARQL query does not have to specify joins or other means to get to the desired data, instead the query engine performs the required reasoning to retrieve appropriate triples from local and remote triplestores by itself (assuming the graphs are known to the query engine). Of course, that only works in practice if common ontologies are in wide use by participants on the Web.

Using a subject/predicate/object triple model, it is possible to represent any kind of knowledge and relations between nodes, but it is not possible to represent knowledge about the statements themselves. In order to cope with this shortcoming, RDF can be made subject to *reification* which causes statements to be transformed into nodes, thus making them referenceable by URI or as blank nodes. Subject, verb and object are all turned into objects, with the statement node being the new subject and the RDF reification vocabulary ("`rdf:subject`", "`rdf:predicate`", "`rdf:object`") being used as predicate relating to the elements of the original triple.

Lastly, it should be mentioned that even though RDF/XML is specified as official data exchange format for RDF data, this format should not be treated like common XML data normally is. Not only is RDF/XML not designed for XML validity (only well-formedness) and thus does not conform to a predefined DTD or XML Schema definition, but the conception of the format also does not guarantee similar XML structures for the same data. Processing RDF/XML as pure XML will therefore support one or many of the possible XML representations but might fail to support others, so data processing should

only happen after unserializing the XML data into proper triplestores.

RDF Schema

RDF Schema is a schema definition language that is conceptually very simple. Essentially, all it does is to define classes that can be used as types for nodes in the RDF graph, and properties declaring the existence of predicates. Both of these can be derived from other classes or properties respectively, inheriting all characteristics of their parents. Without further definitions, classes and properties are purely informational and don't have an immediate effect on the semantics of the RDF graph.

There is one way to add semantic information though, and that is to define domains ("`rdfs:domain`") and ranges ("`rdfs:range`") for properties. The domain specifies the subject classes that the predicate will apply to, and the range does the same for objects. If multiple domains or ranges are given, the corresponding node must be an instance of all these classes at once. Properties can be defined with any number of domains and/or ranges, including zero.

The above summary already describes the full scope of RDF Schema, there is not really more to it. RDF Schema does not provide possibilities to specify the cardinality of properties or even whether a property is required for a given class instance or not, and there is also no way to declare whether a class instance may contain other properties as well. These are semantics that need to be provided by the concrete application. Furthermore, RDF Schema does not define datatypes, i.e. the allowed string contents of a node – although properties can declare nodes to conform to a datatype that is defined in an external specification such as XML Schema. Another consequence of the "global graph" concept is that no root node or fixed set of allowed types can be specified. In RDF, semantic information is assumed to be incomplete and extensible through further specifications, therefore every aspect of RDF Schema only annotates existing information with context but does restrict its use in any way. If a domain or range of a property doesn't apply to the classes of a statement's subject or object respectively, the graph is not regarded as invalid but instead the property is just assumed not to apply to the given statement.

As a consequence of this approach, no fixed hierarchies or mandatory elements can be assumed for a node of a given type in the RDF graph, which poses a challenge for a transformation framework that relies on such information. This lack of context could

theoretically be worked around by extracting the necessary information from SPARQL queries, however a more feasible approach might be to just provide authors of such a query with the possibility to specify the schema definition with an exact set of result elements by themselves.

RDF Schema definitions are expressed in RDF itself, and are merely a few more triples in the overall graph. Nodes can be declared as instances of a class by including a statement with "`rdf:type`" as predicate and the class as object. Properties are instantiated automatically by using the same URI that was specified in the property description. (Classes and properties are themselves instances of predefined RDF classes, more specifically "`rdfs:Class`" and "`rdf:Property`".)

Example vocabulary 1: FOAF

The FOAF ("Friend of a Friend") vocabulary [BM07] is, while not yet finalized or recommended by the W3C, a well-known use of RDF, and aims for describing people (instances of "`foaf:Person`") and relationships among them. Each person can be described by a set of properties, including obvious ones such as "`foaf:name`", "`foaf:nick`", "`foaf:mbox`", "`foaf:weblog`" or "`foaf:depiction`", or more obscure ones like "`foaf:myersBriggs`" and "`foaf:dnaChecksum`"[40]. A person (or its superclass, "`foaf:Agent`") can be associated to online identities with "`foaf:holdsAccount`", "`foaf:openid`" and various instant messaging identifier specifications. In a similar fashion, interests as well as current and past projects can be listed, birthday and geospatial location can be specified, and people can be said to belong to groups. Through "`foaf:knows`", FOAF provides an intentionally vague mechanism to relate a person to another one.

FOAF is meant to be extensible through other RDF vocabularies, e.g. Dublin Core for denoting metadata of documents that are related to a user, DOAC (Description of a Career)[41] for work history and skills, or DOAP (Description of a Project)[42] for projects that a person works on.

A transformation framework could export information that is available in a person/friend profile, and likewise import that information in order to gather more data about its own

[40] Yes, even the FOAF specification states that "foaf:dnaChecksum" is meant as a joke.
[41] DOAC web site: `http://ramonantonio.net/doac/`
[42] DOAP web site: `http://doapspace.org/`

users.

Example vocabulary 2: SKOS

SKOS (Simple Knowledge Organization System) is an RDF vocubulary that allows to describe knowledge organization systems ''such as thesauri, taxonomies, classification schemes and subject heading systems'' [MB08]. It would also be suited to represent taxonomies of Drupal sites, which makes it a promising candidate for support in a Drupal-based transformation framework – although not all elements of the SKOS vocabulary can be represented in a Drupal taxonomy, so an import of external SKOS data might be lossy.

SKOS defines a class named ''`skos:Concept`'' as central type, such a concept can encompass labels (terms) as ''`skos:prefLabel`'', ''`skos:altLabel`'' or ''`skos:hiddenLabel`''. There can be multiple labels in different languages, although the specification requires that there shall be only one preferred label per concept and language. Labels are denoted as strings.[43] Relations between SKOS concepts are described by using the ''`skos:broader`'', ''`skos:narrower`'' and ''`skos:related`'' properties. Several more demanding needs are also accommodated by SKOS with specifications for informal label descriptions, or collections of SKOS concepts (known as ''vocabularies'' in Drupal) via the ''`skos:ConceptScheme`'' class and related properties.

In the context of a transformation framework, the challenge would be a correct mapping of SKOS concepts and relations to internal data storage. For Drupal taxonomies, the ''broader'' and ''narrower'' properties map well to the parent/child relationship while ''related'' is a good match for term synonyms that Drupal also supports. In order to properly map Drupal terms to SKOS concepts, the latter need to be augmented with internal term ids so that duplicates can be avoided and associations can be stored in the database.

3.2.4 Relational databases

A useful property of relational databases (RDBMS) is that they always come with a predefined schema, namely the one that makes up the table definitions and can be extracted

[43] For less common and more complex scenarios, SKOS also provides an alternative mechanism for specifying RDF resources as labels, in the form of the SKOS-XL extension vocabulary.

from the database itself. On top of that, many frameworks offer an additional abstraction layer that might contain more detailed information, such as semantic annotations that are usually not available in a machine-readable form. The schema provided by RDBMS themselves is a collection of interrelated subschemas, one for each table, without nesting of attributes.

The challenge in mapping and transforming relational data is to determine foreign keys (which often cannot be determined from the database schema itself) and to use identifiers as pointers in the way that they are intended. The same data can be represented differently depending on the level of database schema normalization. A number of cases has to be considered:

- An item property can be part of a table row in its actual format.
- An item property can be part of a table row as foreign key linking to another table (or the same one), if that property needs to be shared among different items in the database.
- An item property does not necessarily be part of the table row that represents the actual item, it might just as well be linked from the table that represents the property. The relation might also be stored in a separate n:m association table, optionally with additional properties referring to the relation itself.
- Finally, item properties in a table row do not necessarily need to be direct children of the row item, their original representation might put them into nested objects or arrays and only combine the row in this way for ease of maintenance or performance.

Also, RDBMS schemas do not normally provide the cardinality of relations; while it is mostly implicit in the design of the database schema, there is no safe and easy way to determine it automatically. Thus, the user needs to be asked for cardinality constraints, or they have to be constructed using external information.

3.2.5 PHP arrays and objects

As native constructs, the structured types built into the programming language are in general the least problematic of data types to represent in an accurate, information-preserving way. On the other hand, these structures do not normally come with an

associated schema, and even if they do (in case of property definitions in a class) it is hard to retrieve that schema programmatically. Also, many structured variables lack a predictable schema at all, especially with such flexible type systems found in scripting languages like PHP.

With arrays and objects being defined by (probably even related) code, it is easy to let additional code hook into the transformation framework though, so that additional required data is provided by plugins. Arrays and objects should consequently be treated as separate types according to the context, although it might still make sense to provide the raw data in a generic way too if sufficient information is available.

Example format 1: CCK fields

As mentioned in section 3.1.2, CCK fields provide a way to extend Drupal nodes with different types of content. From a data storage point of view, field modules declare one or more database fields which are then created as actual fields in the database by the CCK module. The database schema is thus created on the fly, details of the table structure depend on certain properties of the field instance. The field module does not directly interact with the data storage, this is abstracted by CCK which provides the data as a standard PHP array containing the properties that were specified in the field declaration. In addition to the properties fetched from the database, field modules can load further properties into the array by implementing the "`hook_field($op='load')`" hook function. Due to the denormalized database schema that CCK uses, empty fields might still be loaded. Field modules are able to declare such fields as empty by implementing the "`hook_content_is_empty()`" callback.

CCK provides functions to retrieve the tables and columns that were created for a given field, that information can be used to generate a schema for any field, which should be sufficient for most field types. Some fields don't store their data directly in the CCK-managed properties but use these properties to link to other resources such as users, nodes or file objects. A transformation framework must offer capabilities to amend the imported schema with pointers to these other entities so that the schema matches the array structure that will be loaded by CCK field data sources.

Example format 2: Drupal node object

A Drupal node is represented by an object that is an instance of the `stdClass` class predefined by PHP. Drupal core uses a number of fields that exist in every node object, but extension modules can load any other properties into that object as well. For example, CCK extends the node object with one property for each field that is present in the respective content type, whereas the Upload module adds an array of files to the node object as "`$node->files`". A sizeable number of modules load additional data into the node object this way, and in general there is no way to automatically constuct a schema for this data. A node object schema importer could provide the fixed properties as predefined schema elements and extend that base schema by inferring further properties from an example node object, or by providing a hook for other modules to specify the schema for their node extensions.

Another case to consider are structures consisting of multiple nodes, such as provided by the Book module in Drupal core or by Content Profile[44]. The structures in use by these modules contain lists or trees of node objects, with similar or different content types (= schemas) for the different objects depending on the use case of the module. The structures deserve to be represented in a separate schema, but the node objects in that schema should still be provided by the original node object schema importer.

[44] Content Profile: http://drupal.org/project/content_profile, inspected version: 6.x-1.0-beta2 (http://drupal.org/node/272864)

Chapter 4

Realization

4.1 Requirements

The creation of a general data transformation framework with a focus on data preservation is not a sufficiently precise specification for a prototype. In order to provide direction for the implementation of the prototype, its requirements ought to be defined in more detail. This section covers a set of use cases that should be possible to implement on top of the framework, and lists the resulting constraints that guided architectural considerations.

The general premise for the realization of the prototype is the choice of its target audience, which, apart from the general aim of integrating ETL with complex data types and their schemas, greatly influences priorities and design decisions. For this prototype, the Drupal open source community has been chosen as target audience, which translates to Drupal/PHP for the programming language and set of APIs, and also provides constraints and possibilities that come with a focus on the web instead of traditional desktop applications.

4.1.1 Use cases

Import/export of Drupal nodes

With nodes being the primary type of content in Drupal websites, functionality to import and export nodes is clearly the feature that has the greatest importance among users of

the framework.

The least complex use case in this area is conversions from and to CSV files, which are easy to map because they commonly only represent strictly two-dimensional information without nested elements. If an imported file includes a row with column headers or not, it should be able to use that first row for an interface where the single fields can be mapped to fields in a Drupal node, also for the actual data import the column header row needs to be skipped. For the corresponding export, it should be possible to write the same data with or without the leading column header row.

Conversions from and to XML files are more demanding because of their nested data trees. Some elements might translate to a list of imported items whereas others map directly to field contents, and one cannot tell which one is appropriate just by looking at the level of nesting – this information must be provided either by the user or by an existing XML schema, and it is likely that fields of a Drupal node will map to XML elements in different nesting levels. Moreover, some fields might map to multiple XML elements, for example a date that is given as a set of year, month and day tags. Lastly, it is also necessary that not only an element's text value can be processed but also its set of attributes. In combination, those factors make XML a much more difficult format to map and process than CSV data, and require a significant amount of flexibility.

In addition to considerations about converting the data itself, it should be mentioned that imported data files do not necessarily originate from traditional file uploads but could also be present on a different location on the internet, including dynamically generated files such as RSS feeds and web service communication data such as JSON.

Another vital aspect of data import and export is object identity, as mentioned in section 2.2: based on references, unavailable data might need to be loaded from external sources and combined with other data entities. This requirement includes both designated references (XML ''id'' attributes or Drupal node id values) but also indirect references such as node titles that might or might not already exist in a database. In such cases, it should be possible to distinguish between branches like ''exists'' and ''does not exist'' and take further actions accordingly.

Data-model evolution

As a website evolves, it will sooner or later face changes in the way that its data is structured. For example, it might be necessary to combine fields for first and last names into a single field or split the combined field into two parts. More complex data-model evolution tasks involve references and object identity just like for import/export tasks, although in this case it's fine to include internal identifiers that would otherwise be considered "legacy references".

There are also situations where an item needs to be split up into several related entities. For Drupal content, this often means creating different nodes that each contain a portion of the original data, and relating them via mechanisms such as CCK's "Node reference" field or the "Node Family" module's[1] child page model.

Of course, data-model evolution does not need to be restricted to page content but could just as well be applied to other data like XML files - the requirements for the framework are the same in both cases.

Format bridge

Given the capability of performing import/export conversions between internal website content and external data, it seems obvious that it should be possible to use the framework for converting between two external formats, such as transforming an XML file to RDF triples according to rules laid out earlier. Data transformations that use nodes or other internal formats as intermediary step would be prone to unnecessary data loss though, so it must be possible to transform directly between these formats.

The framework should provide means to perform format conversions both by taking direct user input (e.g. an uploaded file) or perform such conversions automatically when called as a web service. Naturally, it needs to be possible to weave in data from other sources as well, which creates the opportunity to create mashups.

As a final consideration, the framework should not prohibit functionality to automatically map data elements through integration of external schema matching software.

[1] Node Family: http://drupal.org/project/nodefamily

4.1.2 Objectives

Taking the use cases and surrounding conditions as a premise, the following objectives were adopted as objectives that are critical to the realization of the prototype:

- Internal state of loaded data (e.g. Drupal nodes) can be preserved, so that the framework is able to write it back to its permanent data store without full knowledge of the elements that constitute the data.
- The framework shall enable both programmatical usage as well as configuration of transformations through one or more user interfaces. It is possible to create a ''native'' user interface that exposes most of the framework's functionality in a sufficiently usable way.
- Time and memory consumption of transformations is limited to O(n) complexity – n denoting the size of the input data – as long as all operations in a transformation pipeline also conform to this behavior.
- Development of the prototype within the bounds of an open source project requires special emphasis on maintainability of the resulting code, and adherance to community values like the Drupal coding styleguide or integrating with existing infrastructure instead of creating new solutions for problems that have already been tackled. As a consequence of maintainability requirements and the fact that open source development primarily happens in a decentralized way, it is imperative that extensions can be created with minimal effort and that their code can reside outside the framework.
- Schema information can already be utilized at design time of a transformation pipeline, so that data mapping can be performed upfront without having the actual input data at hand. (Of course, input data might include its own schema information inline, in which case it either needs to be processed twice or there's no way around dealing with it only at execution time).
- Given a suitable hierarchy-based schema, the framework shall enable navigation and data extraction from any data source that adheres to structured, semi-structured, relational or graph-based/object-oriented data models; in particular, it should be possible to support CSV, XML and RDF data as well as application-internal array and object formats.

On the other hand, a number of potential objectives was explicitely decided not to be critical architectural constraints:

- Performance is not as important as to constrain transformations only to operations that can be implemented by database queries – execution of PHP code is sufficiently performant for our purposes, and enables native handling of nested data structures.
- When there is a conflict between theoretical correctness and one of the listed objectives (e.g. a conceptual issue that would make it impossible to create a usable interface) then the former may be compromised in favor of the latter.
- Rule-based schema definitions like Schematron are complex to present for mapping purposes and at the same time are not as widely used as hierarchy-based schema definitions like XML Schema. For the design of this transformation framework, such schema definition approaches need not be considered.

4.2 Architecture

In order to fit into the Drupal ecosystem, the prototype was written as a family of Drupal modules. The name "Transformations"[2] has been adopted for the main module that implements the framework itself, taking an example from names of other modules with a similarly broad scope such as Views[3] or Rules[4].

The main architectural patterns used by Transformations are pipes and filters [HW03], separation of front-end (UI) and back-end (API), as well as plugin-based extensibility. Contrary to common Drupal coding practices which currently center around the procedural programming paradigm, the prototype was designed in a mostly object-oriented way in order to facilitate extensibility and maintainability.

At the core of Transformations, there are four fundamental types of entities that constitute the bulk of the API and, in conjunction, enable the assembly and execution of transformations:

[2] Transformations has been published at `http://drupal.org/project/transformations`, with the last released version being 6.x-1.0-alpha3 (`http://drupal.org/node/429944`) at the time of writing.
[3] Views has already been covered in section 3.1.2.
[4] Rules: `http://drupal.org/project/rules`

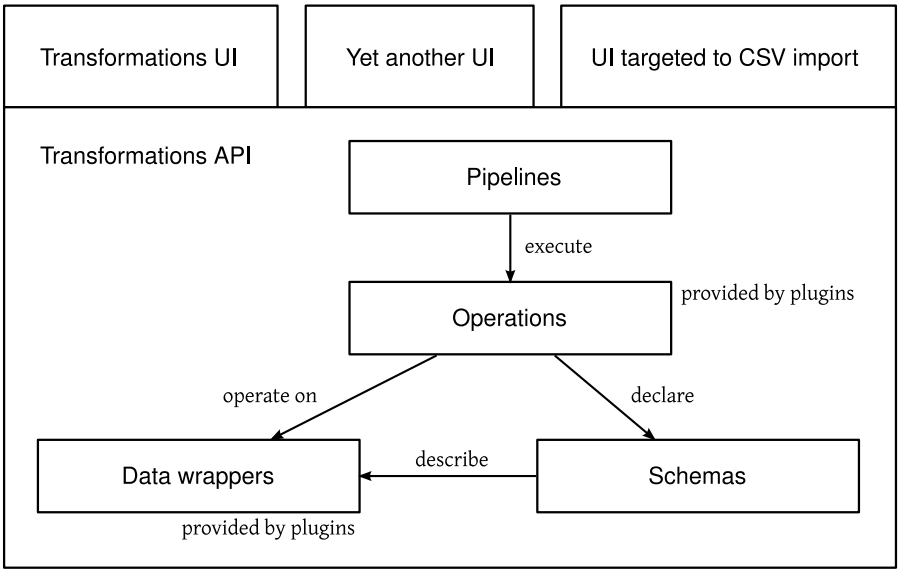


Figure 4.1: Architecture of the prototype

Data wrappers provide a way to access data independent of their actual internal storage format. This enables lazy-loading of data that is not readily available yet – for example, text that is read from a file, or SQL query results that need to be retrieved row by row – and as a consequence, makes it possible to process large amounts of input data given a limited amount of memory.[5]

As a second important trait, data wrappers provide means to traverse data structures in a generic way, so that it is possible to create operations that work on any kind of list or structure. When interfacing with other APIs it might be necessary to use a specific data type (such as arrays or objects) but as long as there are ways to convert data to those internal formats, the abstract tree structure offered by data wrappers is a significant advantage that avoids duplication of operations for lots of different data types.

Schemas provide a way to specify the expected structure and format of data that is supplied as a data wrapper. The usage of schemas makes it possible to determine the compatibility of input/output connections in a transformation pipeline's set of operations, and can also be used for mapping and parsing purposes. The root of a schema, a schema element, corresponds to a data wrapper and specifies the data format that the wrapper provides, as well as its set of children.

[5] Web applications like Drupal are commonly facing severe memory limits, which makes it all the more important to conserve memory wherever possible.

Operations constitute the actual functionality that can be incorporated into a transformation pipeline. They take any number of data wrappers as input, perform any kind of operation on them, and return any number of output values that can be passed to other operations or back to the user. An operation's input and output slots are described by a number of properties including the expected schema. The fact that schemas are known prior to actual execution of the operation enables the determination of format compatibility at design time and dynamic refinement of operation outputs.

Pipelines are configurations of how operation inputs and outputs are connected. In addition, they also allow to specify dynamic pipeline parameters and fixed data to be assigned to operation inputs, and operation outputs to be mapped to global pipeline output. The data flow in pipelines must not be circular, i.e. each operation in a given pipeline will only be executed once.

It is important to note that operations can encapsulate other operations, which among other things makes it possible to embed pipeline operations in other pipelines. This facilitates separation of concerns into a reusable library of pipelines; also, through the use of foreach operations, it is possible to get around the non-circularity constraint of pipelines by embedding a pipeline operation in another operation that executes the pipeline multiple times in a single run.

Extension modules are expected to provide additional operations (extending the set of basic operations that ships with Transformations itself) and can also provide additional data wrappers or schema proxies for their operation inputs and outputs, whereas pipeline configuration and execution is centralized in the API module.

Other Drupal modules can provide user interfaces for all or part of the API's functionality, or they can use it in a direct, programmatic way. The prototype ships with a generic user interface called ''Transformations UI'', exposing most capabilities to the user.

4.3 Framework

This section describes the challenges and results of designing the back-end part of the transformation framework, subsequently referred to as ''API module'' or simply ''Transformations''.

4.3.1 General aspects

On integration with Drupal

When reading that the prototype is specifically targeted to the Drupal community and even the core API depends on Drupal infrastructure, the question might arise why it is not implemented in generic PHP (especially since the landscape of open source generic data transformation tools in PHP is pretty much non-existent at the moment) – or in some other programming language altogether, like Java. The answer to this question is both a technical and a social one:

- The technical reason is that Drupal offers an excellent platform for rapid (web) application development, including its module system that allows simple extensibility via ''hooks'' and the Form API that enables the creation and processing of HTML forms in just a few lines of code. In order to make the framework independent from Drupal, it would be necessary to introduce another layer of indirection or to re-implement existing infrastructure, none of which lies within the scope of this work.
- The social reason is that the existence of a generic data transformation framework fills an actual need in the Drupal community even if only a minor part of its potential is realized. Implementing the prototype as a ''genuine'' Drupal module promises interest and, as a consequence, support and contributions from at least a few community members, which in the eyes of the author is preferable to an even more generic piece of software that is not in active usage.

It is certainly possible to overcome these issues and change the framework into something that is Drupal-agnostic and still nicely integrated, however to achieve such a state takes significantly more effort for the same impact, so this aim was chosen not to be pursued at this time.

The plugin system

Transformations can be extended by two types of plugins: operations and data wrappers. In order to stifle external contributions, it is important that plugins can be created with minimal effort and without the need of changing upstream code. On the other hand,

a benchmark by Larry Garfield[6] has shown that unnecessarily loaded PHP code can cause significant performance issues in the two-digit percentage range, therefore plugin code should be split out of module files that are loaded each time that Drupal is invoked, i.e. on every page request.

As a trade-off between these two factors, Transformations uses a registry-style plugin system that encourages outsourcing code into separate files which are only loaded on demand when the plugin is actually accessed. The plugin system is designed to resemble that of the Views module[7] which also uses on-demand loading of classes in separate files. The implementation, however, is a different one that reuses code from the Chaos Tool Suite module[8].

Loading class-based plugins in PHP comes with a special twist: When a class in a newly loaded file is loaded, all of its successors in the class hierarchy (that is, parent classes and interfaces, the parents of those, etc.) must have been loaded beforehand, otherwise PHP will abort program execution with an error. As a consequence, this characteristic forces the plugin host to ensure that all required files are loaded in the correct order – never include a class as long as its superclass is yet unknown.

In practice, all of these considerations result in a mechanism that requires the plugin author to provide information in three places:

1. The Chaos Tool Suite's auto-discovery plugin mechanism requires that the plugin directory is specified by a hook function that is called for each module when plugins are searched for. For a module with "`parser_csv`" as internal module name, such a function will look like this:

```
/**
 * Implementation of hook_ctools_plugin_api().
 */
function parser_csv_ctools_plugin_api($module, $api) {
  if ($module == 'transformations' && $api == 'transformations') {
    $api_version = 1;
    return array(
      'path' => drupal_get_path('module', 'parser_csv')
                . '/plugins/transformations',
      'version' => $api_version,
    );
  }
}
```

[6] "Benchmarking page loading" by Larry Garfield, accessed 14.04.2009: http://www.garfieldtech.com/blog/benchmark-page-split

[7] Again, see section 3.1.2 for more details on Views.

[8] Chaos Tool Suite: http://drupal.org/project/ctools, inspected version: 6.x-1.0-alpha3 (http://drupal.org/node/418638)

The Chaos Tool Suite will then use this information to look for files in the given directory, named "`[modulename].transformations.inc`" for Transformation plugins. In the given example, this would result in "`parser_csv.transformations.inc`" as name of the include file.

2. The file mentioned above contains information which the actual plugins that are actually available in which include file, and their parent class in case the plugin classes don't directly derive from the root classes that are provided by Transformations and included in any case. Again, this "registry" information is supplied by implementing a hook function:

```
/**
 * Implementation of hook_transformations_operation_info():
 * Return a registry of all the operations provided by this module.
 */
function parser_csv_transformations_operation_info() {
  $operations = array();

  $operations['TfRecordsFromCSVText'] = array(
    'file' => 'operations.csv.inc',
  );
  $operations['TfRecordsFromCSVTextExampleChild'] = array(
    'file' => 'operations.csv.example.inc',
    'parent' => 'TfRecordsFromCSVText',
  );
  return $operations;
}
```

After having gathered this information from all modules, the include files containing the plugin classes can be loaded in the correct order according to the class hierarchy.

3. In the actual plugin file, the respective plugin classes are accompanied by one more callback function which provides all information about the plugin that is not directly related to include order:

```
/**
 * Implementation of [module]_operation_[class]().
 */
function transformations_csv_operation_TfRecordsFromCSVText() {
  return array(
    'label' => t('CSV text to list of records'),
    'description' => t('Transforms plaintext CSV (comma separated values)
          data into a list of records, each record consisting of the
         fields in the respective text line.'),
    'category' => t('CSV'),
  );
}

class TfRecordsFromCSVText extends TfOperation {
  // plugin implementation goes here
}
```

Transformations merges these properties with the registry information fetched beforehand[9], and caches them in the database so that subsequent creation of plugin objects can skip these information retrieval steps and proceed directly with including only the required plugin files.

When all information about plugins has been retrieved (mostly by fetching the results of the above information discovery from the cache), file inclusion and creation of plugin objects is achieved through a factory method of the root class of the respective plugin type.

4.3.2 Data wrappers

Purpose and scope

One of the aims of a generic data transformation framework is to provide unified access to data that serves the same purpose so that said data can also be processed in a generic way, avoiding the duplication of logic for each different internal data format. As such data often needs to be processed or fetched from external sources in order to be accessible in the desired format, there are two ways to handle this requirement: either the framework transforms all ''foreign'' data to a format that it can handle natively (and also very efficiently), or the original data is kept, with an access/compatibility layer wrapped on top of it.

For the prototype, the latter option was chosen in order to enable processing data in its original format if the operation knows how to access it. Among other things, this makes it possible to edit existing data without needing to decompose and reassemble all of it – in fact, this would also require complete knowledge of all concerned data and how to reassemble it in the first place. While this characteristic is less important for traditional applications of ETL in the data warehousing domain where high performance is deemed more important than keeping data structures intact, it is crucial for this work as exact preservation of data is among its prime goals.

[9] Speaking from a technical point of view, this third information retrieval function is unnessecary, and all the data could just as well be returned by the registry function. However, having another plugin-specific function puts the data closer to the actual plugin and thus avoids plugins and plugin descriptions getting out of sync.

As a consequence, the purpose of data wrappers in Transformations is to provide unified access to data for generic operations while still enabling to perform operations with specific knowledge about a certain kind of data. The different kinds of data that should be appropriately represented by the generic wrapper interface are primitive data types such as strings, numbers and boolean values, as well as complex types whose data model conforms to one of the previously mentioned formats, namely CSV, XML, RDF, relational database tables, and also PHP arrays and objects.

Data wrappers achieve this goal by exposing two aspects of the underlying data: knowledge about the data format if data can be accessed directly, and a way to traverse its structure in case the data can (or should) be represented in a structured way. Both of these capabilities are optional to implement for data wrappers, however for a wrapper to be useful at least one of them should be provided.

Creation

A data wrapper is created by calling the factory method `TfDataWrapper::create()`, specifying the data that is to be wrapped as argument. By implementing another hook function (`hook_transformations_data_wrapper_priority()`), plugins can volunteer when they want to wrap the given data, and subsequently the most appropriate wrapper class is instantiated with the data as only source of information.

All wrapper classes inherit from the abstract root class `TfDataWrapper` which defines a set of behaviors that data wrappers must adhere to.

Formats

If a wrapper provides direct access to data (not necessarily to the internal format in which the wrapped data is stored), it implements the `data()` method that will be used to retrieve the data in question – for example, an array wrapper might return the wrapped array as is, while a mapper for an SQL query result handler might get converted to an array before being returned. Transformations introduces the notion of ''formats'' to describe both the data type and additional properties that are known about the data that will be exposed by the `data()` method.

The Transformations API defines a set of standard formats for common data types, such as "`php:type:string`", "`php:pseudo-type:number`" or "`php:class:TfDataWrapper`". These are provided by the data wrapper and can be automatically determined if the concrete data already exists, otherwise the wrapper must use its knowledge to specify the type prior to fetching the actual data. It is also possible to add "custom" formats to a wrapper object, which can be utilized by operations to further add knowledge that the wrapper class does not know about, for instance an operation that loads a Drupal node object (which is a plain instance of PHP's stdClass) can add a format like "`drupal:node:object`" to specify that all properties of a node object apply to the given data.

The existence of formats paves the way for two further capabilities: on the one hand, they present a way for schemas to describe operation inputs and outputs in advance, and on the other hand formats can be directly used during execution of an operation, e.g. to filter lists of data by their formats, perform operations depending on whether or not a given format exists, or to add new formats in a validation operation.

Structure and traversal

In order to make use of schemas, it is not only necessary to define the concrete data format but also offer a way to reach the elements that are specified by the schema. Different data formats offer different ways to navigate to their child elements (or other elements that are directly related to the data, such as RDF triples that are reachable from a given triple), so data wrappers need to provide unified access not only to the data itself but also to its children.

The papers mentioned in section 2.1 cover most properties that are important for at least one of the data models that we aim to support. Specifically, desired capabilities are (1) preservation of element order, (2) string or number labels that are not necessarily included in the element value itself, e.g. array keys and object property names, (3) multiple occurrence of such labels in a single collection of elements, and (4) object identity that allows an element to be referenced from other places.

Section 2.2 has shown that maintaining strong object identity cannot be done with incomplete sets of data. Given that the nature of RDF data and other data sources on the web is incomplete by design, object identity is dropped as a requirement for data wrappers. Instead, the resolving of object references is outsourced to operations that are

knowledgeable enough to work on the specific kind of data, for example, given an XML element and "id" attribute, a suitable operation could retrieve the referenced element by utilizing the original element's knowledge of document structure.

It was discovered that PHP iterators support all of the remaining requirements, so data wrappers wanting to expose their element structure can implement a method that returns an iterator which provides access to all child elements. For each accessed element, the iterator supplies the string or number label as key and the actual element data as value. Additionally, the use of iterators opens up possibilities to defer fetching the data, which can greatly reduce memory usage.

A challenge that arises when working with iterators is how to return its output elements to the calling client – if the functionality of data wrapping (including the traversal capabilities) should be preserved throughout more than one level of nesting, the returned elements again need to be wrapped. On the other hand, the developer experience is somewhat diminished if even for a simple iteration of string fields it is necessary to call `$iteratedWrapper->data()` in order to access the actual value. The `TfDataWrapper::children()` method therefore lets the client choose between three different modes that return slightly different iterators, only one of which needs to be implemented by the actual wrapper plugin (the other ones are provided by the `TfDataWrapper` class itself).

In the simplest mode – which is provided by the plugin – the iterator is free to provide the data either wrapped or "raw", depending on which is easier to do. For the client, this mode is only interesting for use in a proxy iterator that does not care about the data itself. The modes that are actually interesting for general use are one that always wraps the data, therefore presenting a predictable interface to access, and another one which unwraps elements that are not lists/collections. In combination with the fact that the `TfDataWrapper` class implements PHP's `IteratorAggregate` interface using the latter mode, syntactic sugar like this is possible:

```
foreach ($wrapper as $label => $stringValue) {
  $combinedString = $label . ": " . $stringValue;
}
```

Of course, such usage can only be done if the format of the children is known in advance. Note that there is no iterator available that delivers the `data()` result for collections as well, because collection wrappers are not required to provide any concrete data at all.

Immutability

Due to the design of transformation pipelines, it is possible that the same data wrapper can be accessed by different operations on different, independent data paths. Thus, data wrappers impose an immutability constraint that prohibits modification of the wrapped data once the wrapper is instantiated. As a downside, this constraint requires operations to clone the data before modifying it, which might result in unnessecarily high consumption of resources. A combination of copy-on-write and reference counting mechanisms might be able to alleviate the problem for the common case when only a single data path exists for an edited element, an implementation of this idea has not yet been attempted though.

Serialization

Pipelines need to be are able to store operation inputs as predefined options, so that fixed values do not need to be passed as dynamic parameters for each pipeline execution and the pipeline can be turned into a black-box that can make further assumptions with exact knowledge of input values. The fact that operations can take any kind of data wrapper as input makes it necessary to enable serialization of data wrappers.

For the common case, this is not a problem, as most inputs assigned by end users are likely to only require primitive types like strings, numbers or booleans. For programmatical purposes though, it can be necessary to pre-assign values that have been retrieved by means of executing other operations, including execution of pipelines (which is actually a use case that can also be interesting to the end user, and has in fact been added to the prototype's UI as well).

In short, it must be possible to serialize wrapped data in order to provide the desired flexibility. If the wrapped data is volatile, as is the case e.g. for SQL query result handlers, this poses a further challenge. Transformations thus implements serialization of data wrappers using a distinction of three cases:

1. The data wrapper can choose to declare itself as serializable, in which case it provides the serializable data as result of the `serializableData()` method that it implements. When unserializing, the same wrapper class is simply created with the given data.

2. Volatile data that is not declared to be a collection, such as a wrapper with a file handle as underlying data, is retrieved via its `data()` method and directly serialized this way. When unserializing, an appropriate wrapper class is determined by the generic, plugin-based wrapping mechanism of `TfDataWrapper::create()`.
3. Volatile data that is declared to be a collection is serialized by recursively serializing all of the key / value pairs of its child elements. On unserializing, a data wrapper is created that provides these key/value pairs by means of a suitable iterator (which is different from a standard array iterator, because that one would not allow duplicate keys).

As the reader might notice, this mechanism will fall down if a data wrapper both lacks serialization capabilities and provides circular navigation (e.g. backreferences in RDF data) as this will lead to infinite recursion loops, the hope is that such cases are rare enough not to occur in practice.

Excursus: Approaches that were not adopted

During the development of data wrapper concepts, some ideas that might sound tempting to realize were not adopted for reasons which are listed below to raise awareness and help to avoid misdirection in future projects.

- While being related conceptually, data wrappers do not include their corresponding schema as a member variable of the wrapper object. Not only would this disproportionately bloat the size that is required for serializing data wrappers but it would also cause significant effort when child elements are accessed: for every child, the corresponding child schema element needs to be retrieved in order to maintain correct wrapper/schema associations, which (as is shown later) is a parsing problem and therefore not only expensive but also unreliable.

  Instead of storing the schema together with the data wrapper, it is made to be part of the specification of operations, and as such does not need to parsed each time a wrapper's children are accessed.
- If data wrappers are able to represent data in a form that is different from their internal storage format, the thought suggests itself to present that data in different formats depending on the needs of the user – for example, a wrapper for a date might provide the date as either ISO date string, timestamp or DateTime object.

Further considerations, however, show that this is not a good idea because of the following reasons.

Firstly, multiple access methods would necessitate separate implementations of all format and traversal functionality, i.e. the data wrapper itself – better have two distinct wrappers instead of trying to handle everything in one place. Secondly, such usage would encourage the transformation of data inside the wrapper; however, transformation of data and describing its expected results is exactly the scope of operations, so enabling data wrappers to do the same would blur the line between the two and cause conceptual confusion. As a result, all transformation duties except for a single access method are assigned to operations as well[10].

4.3.3 Schemas

Purpose and scope

When assigning input data to an operation, it is important that only data is assigned that the operation expects and is able to handle. Generally, operation output and input slots can be tested for compatibility by comparing their expected input and output types, the operation's job at runtime is that the actual type conforms to the expected one. Common ETL and mashup tools have no difficulties matching slots as they work on simple types, for these tools the number of input/output types can be reduced to a small set of strings like ''number'', ''location'' or ''items''[11]. Transformations on the other hand asks for a more elaborate approach as nesting of complex types is supported by data wrappers. The main purpose of schemas in Transformations is therefore to preserve information about trees (and graphs) or data throughout operations, especially those that deal with combining and extracting data, in order to improve the accuracy of input/output compatibility checks. As such, schemas take the role that the above ''simple'' type specifications hold in other frameworks.

A second purpose is to enable extraction of elements from data structures if no type-specific query methods are provided by other operations, while keeping schema information accurate for use in subsequently chained operations. In order to do that,

[10] Of course, it is possible to create operations which just create a different data wrapper for existing internal data, which has the same effect but less complex conceptually.

[11] As mentioned in section 3.1.1, the ''items'' type in Yahoo! Pipes does not seem to convey any information about the items that it contains, so conceptually it's just a data type like any other primitive type

we require generic query logic that is by design less efficient and capable as queries specifically tailored to a given type of data, but able to process all structured data that can be accessed through data wrappers.

Finally, schemas are designed to enable schema matching for two existing data sources whose schemas are known, as well as to provide mapping target slots for a data structure that is to be generated from scratch (using other inputs as data sources).

For all of these usages, it is important to note that schemas are supposed to correspond directly to the format and structure of data wrappers – one could say that a schema represents the static type of the corresponding data wrapper.

The schema tree

Given their relation to data wrappers, it's obvious that schemas must be able to describe the same data models that we previously defined as requirements for the wrappers. The schema node properties given in [MZ98] (as described in section 2.1) provide a sensible set of requirements for the desctiptive capabilities of schema trees.

However, a schema for data structures is in fact little more than a formal grammar as is used by parsers, and extracting elements from a data structure is, by implication, a parsing problem. It was found that the schema structure proposed in that paper does not work sufficiently well when being used as parse tree in a ''schema parser''. (Parsing techniques are covered in [ASU86].)

Thus, a different schema structure was developed that shows a higher resemblance to the structure of common parse trees and maps better to data wrapper concepts, while still keeping the same level of expressiveness. The following list describes the elements that this schema tree structure can consist of.

Schema elements denote concrete data and as such directly correspond to data wrappers themselves. They contain the same list of formats that data wrappers also provide, and information about the child elements that result from traversing the wrapper's child iterator. In addition, schema elements also store information about the label that is provided as iterator key – strictly speaking, this is not a property of the data wrapper itself and thus does not perfectly fit into a schema element, but putting the label into other places comes with other disadvantages so this is is adopted as the most sensible compromise. (In particular, the schema

tree would grow more complex and user-facing visualization of schema elements would suffer.)

While the element label is usually irrelevant for list collections, dictionary-style (associative) collections commonly use the label as primary way to identify values. For that matter, a schema element can specify whether its label should only be used as abstract identifier with no further meaning (for lists) or if it denotes the actual key returned by iterators. When the latter is the case, this information can be used for parsing the data structure.

In formal grammars, a schema element would correspond to both a terminal symbol for its label and format parts, and to a non-terminal symbol for any child elements that it contains. Child elements are specified by a single ''child collection'', which is an instance of one of the collection types mentioned below.

Schema collections, like rules in a formal grammar, can contain both schema elements and further collections that denote elements returned by the same iterator. In other words, all descendants of a collection work on the same level of nesting until a schema element is encountered, which may include a different set of children on an even deeper nested level.

This is the actual difference of schemas to traditional syntactic analysis: while the latter operates on a single stream of tokens, nested data structures can be said to rather provide a tree of token streams. However, apart from some additional complexity, this trait does not introduce any conceptual challenges.

A collection can be one of four possible types: a union (also known as choice), a sequence, a set, or a subset (which is a set for which only a fragment of its members are known). While the first three are well-established entities in schema and grammar descriptions, subset specifications are less common. Their purpose is to capture the nature of data with labelled child elements, such as RDF data or dictionary arrays/collections, where a partial set of elements is known but additional unknown elements are also possible and allowed. Naturally, this distinction from the standard set – which describes the exact set of its members – causes a completely different parsing strategy, which is why subsets are a different collection type. Also, the use of subsets inside of other collections would disrupt their parsing capabilities as it matches any kind of (unknown) data, therefore subsets are only allowed to occur as collection roots.

Schema cardinalities are inserted into the schema tree whenever an item may occur

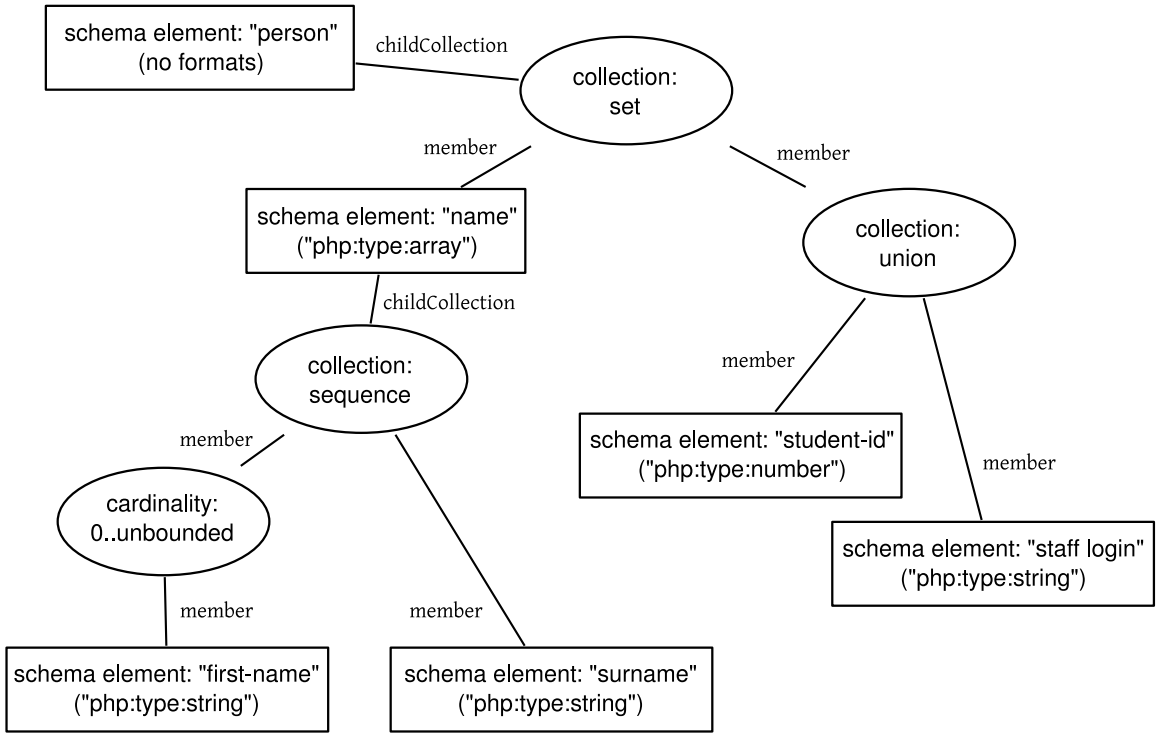


Figure 4.2: Example of a schema tree

less or more than once, which is applicable to both schema elements and collections. As the way that cardinalities are used in parsing depends on the containing collection type, they are modelled as separate entity that contains information but does not provide any parsing functionality on its own.

For a graphical example of how schema trees can be structured, have a look at figure 4.2. Note that schema trees can also contain cyclic references, for example to describe RDF data about a person which (possibly by way of several intermediate elements) contains information about another person with the same set of possible properties.[12]

The schema tree is best traversed using the Visitor pattern [GHJV95], which helps with testing for mutual compatibility of two schema trees and with schema-based parsing/extraction of schema or data elements. For visitors whose execution is not limited by an input parameter, special precautions (object comparison / nesting limit) are required to prevent infinite loops that would otherwise result from the possibility of schema trees with cyclic references.

[12] In fact, FOAF (as a high-profile example) sports this very use case as its core principle.

Schema proxies

The prototype also provides proxy classes encapsulating the aforementioned standard parts of the schema tree. There are two cases where proxies will be required:

- When a schema element is being used as part of another schema, that existing element is added to a schema collection as one of its members. Often though, the label of the original schema element will not be the same as the one required for embedding that element. (For example, there might be a schema where the same ''person'' structure is required to be named ''author'' in one place and ''reviewer'' in another one – it might even be the case that one includes the other as child element.)

  As the label is coupled to the same schema element that describes the actual data format and children, it is necessary in this case to alter the label properties but fetch all other information from the original element. A proxy class offers the means to implement this requirement. Of course this would not be necessary if the label was a distinct schema tree node in the first place, so this issue is home-made to a certain amount.

- A second use case is deferred building of schema trees, which can avoid unnecessary schema tree generation when that process would be expensive. This can be particularly useful when another complex schema tree needs to be included in the current one but is not yet built, which for example applies to pipeline operations that need to analyze all contained operations before being able to accurately determine their output schema.

- The third, most demanding use case is for specifying elements as query target. When parsing the data with the goal to extract one or more elements from the data structure, there needs to be a way for the client to specify which elements should actually be retrieved. Transformations does not introduce a separate query language. Instead, schema nodes are repurposed to describe which data is queried for. Using schema nodes as description of a query requires additional data to be stored for a node, including the two flags ''is query target'' and ''contains children that are query targets'' as well as a backreference to the parent node that makes it possible to propagate the latter up the query schema tree.

  It is not sufficient to add these properties to the original elements because of circular references; for example, marking the ''author'' element from the example

above would also mark all other ''person'' elements such as ''reviewer'' or other occurrences of ''author'', which is probably not intended. The prototype deals with this problem by wrapping a proxy class above all schema nodes that are traversed, each of which can contain its own set of distinct properties even if the same original schema node is wrapped multiple times.

Shortcomings of the current approach

Although the current implementation for schemas opens up possibilities that would not be available otherwise, it also has its drawbacks. First of all, the reliance on operations to transform data instead of a self contained data store makes it necessary to also rely on operations to provide accurate schema trees for their input and output slots. It is possible for operations to provide empty or undetailed schemas, however in that case type safety after element extraction cannot be guaranteed.

Also, requiring operations to provide schemas places additional burden on plugin developers. Especially for cases where external logic is employed – for example other query languages like XPath or SPARQL – it might be hard or unfeasible for an operation to accurately predict the resulting schema. If it can be done, though, the advantages gained from this requirement are probably worth the trouble and will result in better mapping capabilities and type safety.

Another issue is the complexity that arises from the tree structure when the characteristics of cyclic references, traversal and (query) proxies are combined. This complexity is detrimental to maintainability, and might be able to be reduced given further research and development.

4.3.4 Operations

Purpose and scope

Being able to split data transformations into distinct pieces of functionality comes with great benefits for flexibility and modularity, and indeed is a main differentiator between ETL and traditional mapping approaches. The latter are largely concerned with establishing correspondence and try to infer data transformation rules automatically

from that information, while the former place a stronger focus on the user to design transformations that cannot be inferred by determining correspondence.

While the existence of schema information brings a number of new possibilities from the mapping domain, the prototype clearly places its focus on the ETL approach, enabling the user to define custom rules for data flows that consist of a number of smaller building blocks. In Transformations, these building blocks are called operations.

Operations are provided by plugins and can provide any kind of functionality on the set of input values that it is assigned. The responsibility of an operation can essentially be divided into two separate tasks: the design time aspect that describes which kind of input data is required and which kind of output data will be provided, and the execution time aspect that actually performs the operation and ensures that the supplied assertions are accurate.

Which kind of functionality an operation provides is completely up to the plugin itself – it could load or save data from external sources, generate new data, modify/restructure existing data that is given as input, or wrap other operations with changes in behavior.

Input and output slots

Each operation defines a fixed number of operation inputs[13] and a dynamic number of operation outputs. Inputs are assigned to the operation object prior to execution, whereas outputs are assigned to a designated output object during execution of the operation.

Input and output slots have a few properties in common: all of them are identified by a (string) key, are described by a user-facing label and optionally a more detailed description, and provide a schema that describes the expected schema for the data wrappers that will be assigned. Inputs can additionally specify if they are optional and provide a default value if this is the case. Also, clients are able to specify input schemas for input slots in advance without assigning the actual input data yet. This can be accessed as further property (mostly intended for use by operations themselves); if

[13] Originally, a split between ''inputs'' and ''options'' was considered, the former receiving input from other operation outputs and the latter allowing direct user input. This idea has been discarded though, because the framework is more flexible when allowing either user input or data connections for each input slot, and the distinction between options and inputs is in fact just a presentational one that should be handled by the user interface. (The prototype still doesn't quite do without some kind of fixed options, as the section about type options will show later.)

no specific input schema has been assigned to a given input then the expected schema will be used as fallback for this property.

Design time vs. execution time

Building schema trees is potentially expensive (as previously mentioned in section 4.3.3) but most operations don't actually make use of schemas during execution – testing for input/output slot compatibility as most important use of schemas is only useful before an operation is actually run, i.e. at design time. Transformations thus takes special care that operations are only required to build schema trees when they are really used.

Primarily, this means that the listing of input/output keys is separated from the retrieval of their corresponding properties. Instead of returning all information about all their slots at once, each property needs to be specifically requested with a call to the `inputProperty()` or `outputProperty()` method of an operation.

Schema propagation

After a bit of consideration, it is obvious that operations must not be constrained to a fixed set of outputs. This is due to the usage of schema information; depending on the input data, a data extraction operation wants to provide different outputs with different schemas for all of the extracted elements. (For example, for a row in a CSV file there should be one output for each string field that it contains.) Changes in output keys or schemas can depend both on the input schema and on the concrete data that may already be assigned to an input.

As a consequence, one operation's outputs might depend on another operation's outputs, and if assigned input schemas or data change for one then it might also affect connected other ones. Therefore it is necessary to propagate output information across operations.

Transformations achieves that by providing an output schema listener mechanism, which is an implementation of the Observer pattern from [GHJV95]. Altered operations will signal the schema change by invoking `TfOperation::setOutputSchemaDirty()` while listeners react to these changes by implementing the `outputSchemaChanged()` method of the `TfOutputSchemaListener` interface (after having registered themselves using

`TfOperation::registerOutputSchemaListener()`). If done correctly, this will cause schemas to be recursively propagated to all affected subsequent operations. This mechanism will work both for operations wired together on the same level and for those that are wrapped inside another operation.

Type options

One could argue that the same kind of schema changes should be possible for input slots as well, and indeed there is a need for variable operation inputs. Operations that implement list creation, string combination or number addition functionality are well suited to take an arbitrary number of inputs. One could work around these by simply exposing a fixed set of, say, five inputs[14], and combine the results with other suited operations. For certain operations though, workarounds won't be possible, like operations executing pipelines or wrapping other operations – or more generally, in cases where not all inputs have similar meaning.

Changes in the input schema are generally harder to handle than for output schemas. If inputs depend on other inputs then the input data needs to be assigned in a given order which must be communicated to the client in addition to the (variable) set of input slots. Also, changes in input schemas will likely cause assigned input data to be unset/invalidated while keeping other input data intact, including the one that causes the change. This is no issue for outputs as their values are only retrieved after the operation has been readily set up.

These problems are solvable on both the API and user interfaces level, given enough thought and effort. At this time however, the prototype does not allow input keys and schemas to change once an operation has been instantiated. Instead, the solution has (for now) been implemented as mechanism that happens before the creation of an operation object, and has been titled as ''type options''. API users knowledgeable about the required options can just pass those to the newly created operation like this:

```
$pipelineOperation = TfOperation::load('TfPipelineOperation', array(
  'pipeline' => $pipeline_id,
));
```

User interfaces on the other hand cannot know which values may be provided for all possible operations, and therefore need a possibility to discover possible values. As

[14] This is the Yahoo! Pipes approach, by the way.

some of the type options may depend on others, an iterative algorithm has been devised that repeatedly calls a function[15] until all values are assigned. Using the above example of the pipeline operation, an implementation of this callback might look as the following code:

```
/**
 * Implementation of [module]_operation_[class]_type_options().
 */
function transformations_operation_TfPipelineOperation_type_options(
      &$typeOptionsInfo, $prefix) {
  $typeOptionsInfo[$prefix . 'pipeline']['label'] = t('Pipeline');

  $query = 'SELECT pipeline_id, name FROM {transformations_pipeline}';
  $params = array();

  if (isset($typeOptionsInfo[$prefix . 'pipeline']['value'])) {
    $query .= ' WHERE pipeline_id = %d';
    $params[] = $typeOptionsInfo[$prefix . 'pipeline']['value'];
  }
  $result = db_query($query, $params);
  $pipelines = array();

  while ($pipeline = db_fetch_object($result)) {
    $pipelines[$pipeline->pipeline_id] = check_plain($pipeline->name);
  }
  $typeOptionsInfo[$prefix . 'pipeline']['options'] = $pipelines;
}
```

As can be seen, the callback returns a label and a set of options that can be selected by the user. When the function is called again then assigned values are provided by the framework as a "value" element, which in this case is used to constrain the list of options. (The single option still needs to be added in order to provide the description.) When no more than a single option is available for selection, the algorithm has gathered all required values and stops calling the function.

In practice, this mechanism enables creating an operation with workflows like the one shown in figure 4.3. Note that the second and fourth options are provided by the "for-each loop" callback while the third option is provided by the "pipeline" callback.

What the type option mechanism does not provide is the capability to retroactively choose different options than the ones that have been chosen initially. Also, type options is currently only able to provide for the selection of choices with integer and string values as options, although this constraint could be removed without changing the general algorithm.

[15] Actually, the type options callback could be much nicer implemented using PHP's "late static binding" feature, but that is only available in PHP 5.3 and later, which at the time of writing is neither released nor widely used so as to be included as a requirement for this module.

Figure 4.3: Selection of type options (using callback function values)

Once they have been entered, type options are not anymore required to stick with integers or strings. An operation's `typeOptions()` method is responsible for returning a set of type options that is equivalent to the ones that have been passed, the only condition being that they must be serializable and that the operation must be able to take them as constructor arguments. For example, this would make it possible for an operation to copy a pipeline's content instead of referencing it by its identifier, thus embedding the whole pipeline configuration as a type option.

Execution and output

Execution and computation of actual output results takes place in an each plugin's implementation of the `TfOperation::execute()` method. Clients can access this method by calling `$operation->output()` which wraps additional logic such as input checks and caching around the former.

The operation gets passed a `TfOutput` object that needs to be assigned all the output values which have been specified as output slots. This object not only has the advantage that results can be returned to the client as a single variable enabling key-based access to the output results, but it also adds some safety measures and provides error logging facilities.

Here is a simple example for an implementation of the `execute()` method, borrowed from the `TfWholeStringAtOnceFromFile` operation:

```
protected function execute(TfOutput $output) {
  $filename = trim($this->input("filepath")->data());
  $contents = @file_get_contents($filename);

  if ($contents === FALSE) {
    $output->setErrorMessage(t('Unable to open file "!filename".', array(
      "!filename" => $filename,
    )));
    return;
  }
  $output->set("contents", $contents);
}
```

On the client side, the same operation would be accessed like this:

```
$fileReadOperation = TfOperation::load("TfWholeStringAtOnceFromFile");

// Input data will be automatically wrapped in TfDataWrapper objects.
$fileReadOperation->setInput("filepath", "/home/jakob/test.txt");
$output = $fileReadOperation->output();

if ($output->isValid()) {
  fwrite(STDOUT, $output->at("contents")->data());
  // Also available: $output->keys() to determine which keys can be accessed.
}
else {
  fwrite(STDERR, $output->errorMessage() . "\n");
}
```

As a further measure, operations that cause side effects (such as saving to a file or to a database, printing debug output, etc.) can specify this property so that they don't get optimized away when they would otherwise be irrelevant for pipeline execution.

4.3.5 Pipelines

Purpose and scope

Given a bit of thought, it would be possible to have operations register subsequent operations that are connected[16] to their output slots. Execution flow would be spawned automatically by just executing any connected operation which would recursively invoke other ones according to the connections and dependencies of each operation.

When taking a closer look, this proves to be a bad idea not only because it increases object coupling but also because of configuration concerns. When input and output configurations are managed by the operations themselves, they are harder to access

[16] "Connections" and "mappings" can be used interchangably in this context. For consistency, this document sticks to the term "connections" when used in relation to pipelines in Transformations.

as they're hidden somewhere inside the dependency tree, and the line between pre-defined options and dynamically passed inputs is blurred.

Transformations therefore separates two responsibilities: execution is performed by operations, while configuration is handled by pipelines. In order to execute a pipeline, there needs to be a pipeline operation that interprets the pipeline configuration and propagates operation outputs from outside. Operations do not know about each other; when a change in one operation affects another one then more abstract mechanisms such as the Observer pattern have to be employed.

Consequently, pipelines are intended for serialization while operations are not. Given the fact that pre-assigned inputs and type options for an operation also constitute part of the pipeline configuration, these need to be serialized as well. Section 4.3.2 already covered how data wrappers can be serialized and restored, while operations take responsibility that their type options are serializable. Likewise, all internal properties of pipelines are defined in a way that is suitable for serialization.

Summing up, the purpose of pipelines is to manage and store a set of operations along with related properties and connections.

Operation specifications and properties

When an operation is added to a pipeline, it is not retained as an object but instead transformed to a specification that allows the operation to be re-instantiated. That specification consists of the operation class and its type options (if necessary), and is identified through a newly generated operation identifier. After the operation has been added, throughout the `TfPipeline` API it will be only accessed using this identifier.

Each operation can additionally be assigned any number of custom properties that enable client code to store additional information such as user-defined operation names or the placement of an operation in a user interface. For custom properties that are not specific to an operation but apply to the whole pipeline, the client can also define pipeline properties. By itself, a pipeline only defines its name and numeric database identifier.

Connections

Pipelines distinguish between four types of entities that can be connected: pipeline parameters, fixed data, operations, and pipeline outputs. Operations can be used both as source and target of a connection, with an operation's set of output keys as possible sources and its input keys as possible targets. Pipeline parameters and fixed data are only available as connection sources, the former representing dynamically passed data and the latter denoting data wrappers that act as pre-assigned operation inputs. Pipeline outputs are only available as connection targets, and only for operation outputs (so it's not possible to connect pipeline parameters or fixed data directly to a pipeline output).

So, less formally, operation inputs and outputs can be connected to one another while inputs can also take their data from pipeline parameters or fixed data and operation outputs can also deliver their data to pipeline outputs. Operation inputs that are optional do not need to be connected, and the user is free to leave any of the operation outputs unused.

By relying on the immutability constraint of data wrappers, it is possible to assign any connection source to multiple connection targets. For obvious reasons the opposite is not possible, so connections in a pipeline are a set of 1:n relations.

A single method takes care of creating connections between all possible connection sources and targets:

```
// Assign the "dynamicPath" parameter (connection source)
// to the "filepath" input (connection target) of the operation
// that is referenced by $fileReadOperationId.
$pipeline->connect(TfPipeline::Parameter, "dynamicPath",
                   $fileReadOperationId, "filepath");

// Drop the previous connection (because of the 1:n model) and instead
// assign a fixed value to that operation input.
$pipeline->connect(TfPipeline::Data, "/home/jakob/test.txt",
                   $fileReadOperationId, "filepath");

// Pipe the file contents into a debug/dump operation to print them on screen.
$pipeline->connect($fileReadOperationId, "contents",
                   $dumpOperationId, "data");

// Also provide the file contents to the client as pipeline output.
// Find a new name for the pipeline output key, preferably "contents"
// if that one doesn't already exist.
$pipeline->connect($fileReadOperationId, "contents",
                   TfPipeline::Output, TfPipeline::NewOutboundConnection);
```

Storage of connections can be achieved without redundancy by listing of the source for each connection target in an array. The same is not true for the opposite (listing all

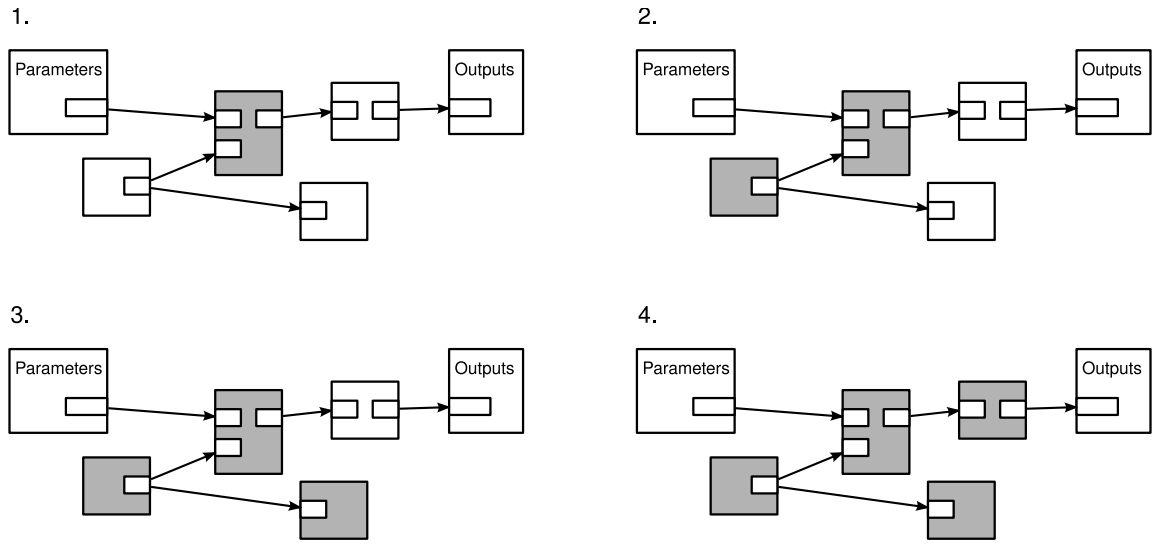


Figure 4.4: Graph coloring to determine pipeline execution order

targets for each connection) in the prototype because fixed data sources are not stored separately but as part of the definition of the connection source. The target listing can be reconstructed from a given source listing, which is useful as the API wants to offer the client ways to make use of both depending on the client's current needs.

Execution

For successful execution of a pipeline, all of its operations must be executed in the correct order that accounts for the configuration of connections and the resulting input dependencies of the executed operations. Determining that order is not trivial because certain configurations require a path-finding algorithm to traverse connections either in the direction from source to target or in the opposite one, changing directions as necessary. An example of such a configuration is shown in figure 4.4, demonstrating the graph-coloring algorithm that Transformations uses in order to determine the correct execution order.

The algorithm follows the steps listed below:

1. Determine a list of operations that need to be executed. Simply using the full list of operations in the pipeline makes a good default, unless one wants to exclude certain operations that are yet in a drafting state. Let this list be known as ''remaining operations'', and create a yet empty list of ''executed operations''.

2. Select the first operation from the list of remaining operations, and remove it from that list. If the operation already exist in the list of executed operations, discard it and try with another one.
3. If any of the selected operation's dependencies (sources connected to the operation input) do not yet exist in the list of executed operations, add those at the front of the list of remaining operations. Also re-add the selected operation directly afterwards when dependencies are left, so that the same procedure can be repeated once all dependencies have been satisfied. Then start over at step 2.
4. If no dependencies are left to be executed, the selected operation itself can be run. Add it to list of executed operations, and schedule all connection targets by inserting them at the front of the list of remaining operations.
5. Repeat from step 2 until no operations remain, in which case all of them have been added to the list of executed operations in the correct order.

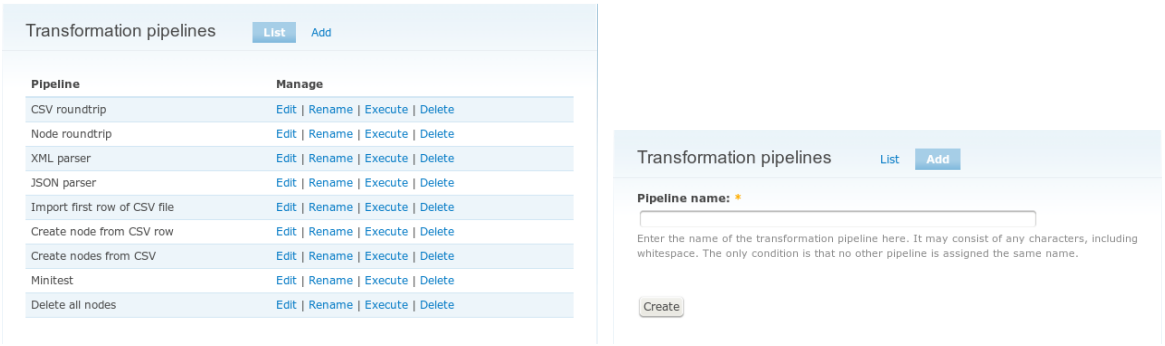
All of this can be done without actually instantiating the operation objects, and is still in scope for the `TfPipeline` class. Actual creation of operation objects and execution of pipelines is subject to a separate pipeline operation.

When operation objects are created, all connection sources providing fixed data must be assigned to operation inputs before querying for output keys or output properties. This is because the assigned data might induce schema propagation, which in turn can cause changes in the operations' output schemas. If input schemas were also affected by schema propagation, the input data would need to be assigned in order of execution (as determined above) because otherwise input keys might not yet exist and thus yield errors when trying to assign values to them.

The actual execution of the pipeline is straightforward, and only requires the pipeline operation to assign pipeline parameters, then execute all operations in the previously determined order and assign their output results to the respective connection targets.

4.4 A native user interface

Developers can use the Transformations API and prior knowledge about input/output formats of a limited set of operations in order to create user interfaces that are targeted to a specific use case. For example, one can use knowledge about the input/output



(a) List pipelines

(b) Add pipeline

Figure 4.5: Pipeline ''List'' and ''Add'' forms in Transformations UI

formats of the CSV[17] and Drupal data[18] extension modules for Transformations that provide CSV parser and Drupal node manipulation functionality, and combine these to build a clone of the wizard-style Node Import module which was mentioned earlier in section 3.1.2.

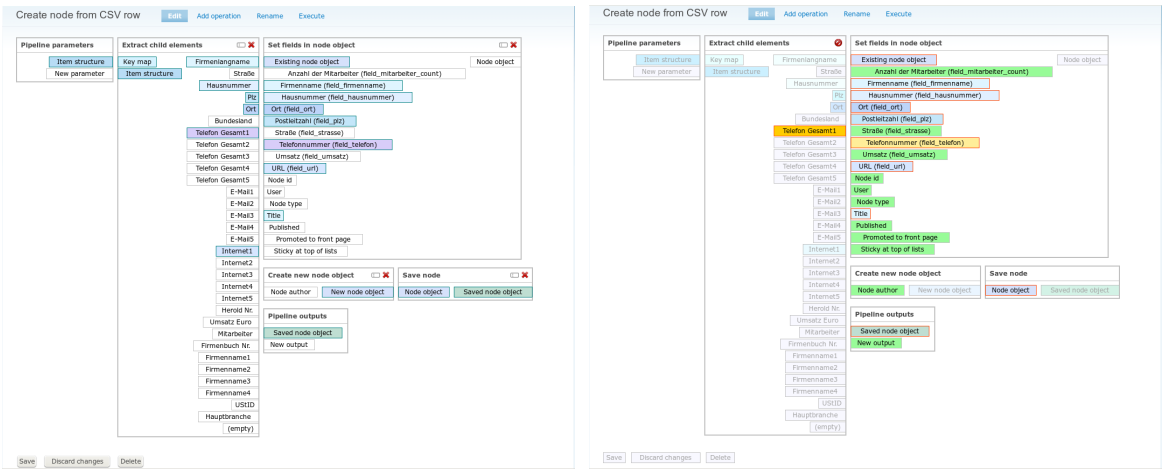
However, in order to provide the end user with the actual power and flexibility that the API offers to developers, it is necessary to create a more generic user interface that more closely exposes the internal concepts of the API. For the prototype, such a user interface has been created and assigned the name ''Transformations UI''.

Transformations UI provides a set of forms (using Drupal's Form API) to manage, edit and execute user-defined sets of pipelines. As the editing workflow in Drupal is web-based and thus relies on a series of subsequent page requests, the module uses the ''permanent object cache'' functionality of the Chaos Tool Suite to enable the editing of pipelines across multiple forms without storing them in the actual pipeline table yet. Thus, users are able to modify pipelines without being constrained to the lifetime of a form and its HTTP POST values. Only when the user presses the ''Save'' button, the pipeline is transferred from the temporary editing cache to its permanent storage.

The forms for basic pipeline management (overview, add, rename, delete) are only thin wrappers around the respective functionality in the API and do not require further mention. The ''Edit'' form is the most complex part of the UI, and presents operations as well as pipeline parameters and pipeline outputs as blocks, as is shown in figure 4.6.

[17] Transformations – CSV: `http://drupal.org/project/transformations_csv`, current version: 6.x-1.0-alpha1 (`http://drupal.org/node/415036`)

[18] Transformations – Drupal data: `http://drupal.org/project/transformations_drupal`, current version: 6.x-1.0-alpha2 (`http://drupal.org/node/430718`)



(a) Standard view (b) Connecting sources and targets

Figure 4.6: Pipeline ''Edit'' form in Transformations UI

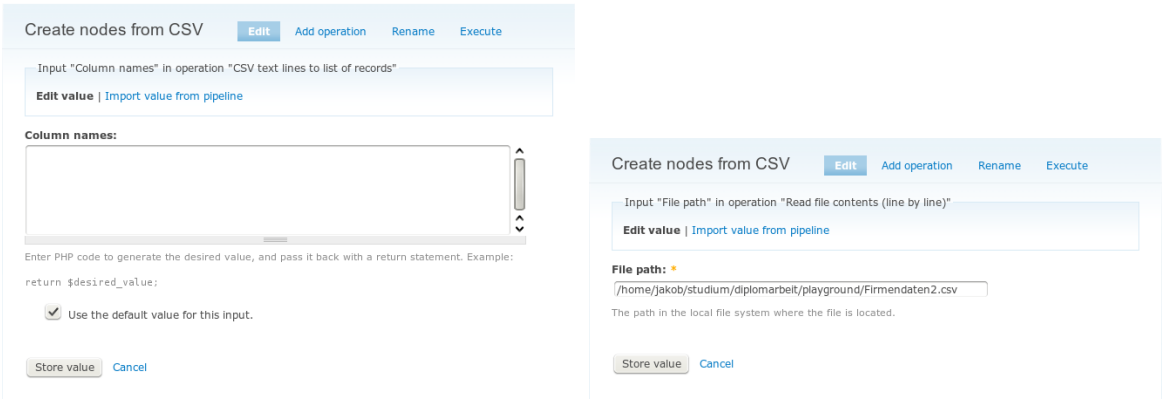
Given limited development resources, an AJAX/canvas-based interface like the one provided by Yahoo! Pipes was not feasible to implement. The basic notion of visualizing operations as blocks was retained though, and instead of drawing to a canvas, blocks are visualized as plain HTML/CSS with styled submit buttons representing connection sources and targets. Color coding and JavaScript highlighting is employed to mitigate the lack of connection lines.

Data widgets

For the interface to be usable for non-programmers, it is important that the user can enter input data through a suitable input element. This is required every time that user-defined data needs to be assigned to operation inputs, which is the case for manual pipeline execution (requiring pipeline parameters) and when operation input data is assigned as pre-defined options.

As operation inputs are data wrappers that can come in any format, is was necessary to provide one more pluggable mechanism[19]. Given the expected schema of the operation input and optionally a data wrapper if input data is already assigned, modules can specify whether they are able to provide a suitable form element that handles the data correctly. These form elements are addressed as ''data widgets''. In addition to

[19] Actually, the data widget and format hint functionalities are both implemented in the API, however being user interface issues they are still covered here in this section.



(a) Generic (PHP code) widget (b) Text widget

Figure 4.7: Data widgets for different schemas / operation inputs

standard Drupal form element functionality like validation or embedding of other form elements, data widgets also specify their priority for the given input schema and data; the widget with the highest priority will be used by Transformations. If no suitable widget is provided by any module, a PHP code widget will be used as a fallback that allows entering any data given sufficient programming knowledge.

When assembling data using existing Drupal form elements, it is rarely the case that the result delivered by the form element matches the one required to create an equivalent data wrapper. On top of that, the Form API in Drupal 6 has a serious shortcoming that requires form elements to use array structures as result values if they embed other elements – as a result, it is difficult to implement a widget with an object or primitive type as result. Thirdly, for optional inputs a capability is needed to remove any existing data so that the default input value defined by the respective operation is used.

The prototype solves all of these problems by wrapping the module-supplied form element into a custom form element that comes with logic to take care of all the details. If an operation input is optional, this form element embeds an additional checkbox element to specify whether or not to use the default value for that input. The other two issues are tackled by providing the form element's result value as an object specifically tailored to cope with the given situation. The underlying class (`TfDataWidgetValue`) takes care of three responsibilities:

- It provides a uniform interface for both the data widget and the form element user to access the element data either as data wrapper or in the element's internal storage format.

- It uses callbacks defined by the data widget to convert between those two formats. Conversion is performed automatically as necessary.
- It tricks Form API by disguising as an array, which can be done by implementing PHP's `ArrayAccess` interface.

Using this approach, objects and primitive types as form element results are possible in spite of the aforementioned difficulties, and data widget plugins are offered an easy way to provide them.

The prototype also offers an alternative to direct input: another pipeline can be executed and one of its pipeline outputs assigned to the operation input. Most importantly, this makes it possible to import schema information in advance so that it can be used to change an operation's connections and thus provide mapping targets that are tailored to the input data.

4.5 Format support

For the prototype, several operations have been implemented to support different data formats. Operations that are required for a broad range of use cases are shipped with the Transformations module itself. Other ones that integrate with specific formats or require higher maintenance effort make use of the framework's extensibility and have been split out into separate extension modules.

Transformations ships with operations for file access as well as XML and JSON parsers that provide their input data as data wrappers with navigation facilities. CSV support is implemented in the "Transformations – CSV"[20] module, and Drupal nodes can be accessed using the "Transformations – Drupal data"[21] module.

[20] Transformations – CSV: `http://drupal.org/project/transformations_csv`, with the last released version being 6.x-1.0-alpha1 (`http://drupal.org/node/415036`) at the time of writing.

[21] Transformations – Drupal data: `http://drupal.org/project/transformations_drupal`, with the last released version being 6.x-1.0-alpha2 (`http://drupal.org/node/430718`) at the time of writing. The bulk of field-specific code has been split into the Field Tool module which is an interesting development on its own, but explaining this module is not really in scope for this thesis.

Chapter 5

Analysis

5.1 Characteristics of the architecture

The usage of data wrappers instead of internal storage and detailed schemas in a framework that otherwise resembles existing ETL tools has not been attempted before and implies most of the prototype's characteristics. A distinction has to be made between implications that are caused by the proposed architecture and limitations of the current prototype that can be fixed within the bounds that the architecture prescribes. This section only deals with the former, whereas implementation specifics are covered in section 5.2.

In general, the architecture enables the same functionality that is offered by traditional ETL tools, that is, flexible combination of smaller operations to a larger set of rules which can again be used as an atomic operation. The architecture allows standard functionality such as simple modification, aggregation, filters, loops and restructuring of data. Mapping can be done using connection sources and targets, additionally it is possible to provide operations that perform more advanced mapping as internal action (probably affected by a parameter or input schema information).

5.1.1 Implications of data wrapper usage

Data wrappers in Transformations enable storage of unknown data across operations in their native form while retaining a well-defined interface to access the data. The direct implication of this capability is that operations do not have to import all required data at

once, which eliminates the need for an internal data store that persists all loaded data in a fixed, pre-defined format. Usage of such a data store is possible within the framework, however it would just constitute another data format represented by data wrappers, its performance benefits only available to operations that can make use of the internal data access methods. The level of optimization and parallelization that is possible with fixed internal storage is not generally possible with data wrappers.

Availability of data in its native format makes it possible to interface with external APIs without prior conversion. By preserving internal state across operations, pipelines can be created not only for traditional import/transform/export workflows but also for load/merge/save approaches. In order to preserve existing data, it is not necessary to reconstruct all of it with complete knowledge of its corresponding schema and the preservation of all internal data in an internal data store. Instead, known elements can be accessed and modified while unknown elements are simply left untouched. Knowledge of schema information does not have to be complete to transform modified data back to its native data store.

As a downside, this advantage only applies when APIs and loaded data are available in the environment and programming language of the framework. For example, being implemented in PHP, the prototype cannot make direct use of Java objects and APIs, and would likely use traditional import/export methods to handle data that can more easily be accessed by Java APIs. (Alternatively, operations that work on Java objects could interface with a Java server via IPC mechanisms, and use a handle or serialized object as internal data type. While a bit complicated, this solution would actually retain the benefits listed above.)

Another implication of data wrappers is that generic operations can take less assumptions about capabilities for navigating the data. In particular, data structure parsers (and therefore element extraction and mapping) for generic data wrappers can only be implemented with parsing algorithms that do not require lookahead > 1 or backtracking, because a lack of two-way navigation capabilities in iterators (`$iterator->back()`, `$iterator->at($index)`) which cannot be expected from all possible data sources[1]. As a result, data might get incorrectly analyzed for certain schemas even if it correctly matches the given schema. This won't be an problem in most cases though, because

[1] It would of course be possible to get around an iterator's lack of a back() method by caching the iterated data, but given the potential size of all iterated data that's really not a good idea and would defeat the purpose of data wrappers in the first place.

schemas for data structures tend to be relatively simple compared to traditional language parsing. Further improvements can be achieved by specializations of the parsing/query algorithm for specific types of data wrappers.

5.1.2 Implications of schema usage

Specifying and propagating schemas means that operations can offer appropriate input and output slots for data that is loaded (and originally specified) by previous operations. Schema information can be used by operations to affect the performed actions. Due to the nested structure of schema elements, data can be described in more detail than a single ''type'' (or even a list of formats) would allow, which offers more possibilities to operations in terms of schema propagation than simply copying the list of slots from previous operations.

Type safety is improved by not only declaring a nested ''item'' type but also defining the set of items that such data may contain. With sufficient schema information, it is possible to determine incompatibile input and output slots if they use the same base type (e.g. XML element) but the structure of child elements does not match. However, it is not feasible to require an exact match of a source schema to the expected schema of an operation, because it cannot be assumed that schema information is complete – it could be missing from the start when no schema has been imported, or it could disappear during schema propagation across operations. Accurate comparisons might also be unfeasible when more complex schemas are involved, because comparison of schemas is (analogous to schemas being formal grammars) a language comparison problem. Input/output connection compatibility can therefore be split into three possible states: ''compatible'', ''maybe compatible'' and ''incompatible''.

The requirement of handling schema information correctly demands more effort to be put into operation plugins that work on structured data, so that schema details are preserved for subsequent operations that might make use of this information. When this effort is omitted, schemas fall back to represent a simple type system.

Furthermore, embedded pipelines can only make use of the parent pipeline's propagated schema information at design time if they are specific to a single parent pipeline. If a pipeline is an independent entity and (potentially) embedded by multiple pipelines, input schemas cannot be relied on at design time. Thus, in order to enable schema

propagation into embedded pipelines it is necessary to ''attach'' these exclusively to the pipeline that contains them.

5.2 Limitations of the prototype and future work

The accompanying prototype that has been created validates the general feasibility of implementing the proposed architecture, but its current implemetation poses a number of limitations that prevent the realization of several features that the architecture would potentially allow. This section lists the most notable shortcomings of the prototype that should be addressed in future work.

5.2.1 Variable input schemas

Type options, as described in section 4.3.4, are unchangeable after the initial creation of an operation and are unable to make use of the more powerful data wrappers for data storage. Despite challenges in the correct implementation of data flows, it is an important task to replace type options by regular operation inputs.

To achieve this, it might be necessary to introduce new input properties such as mutual input slot dependencies or a UI requirement to pre-assign input data in advance (inhibiting dynamic connections to other operation outputs). Also, data widgets will have to be defined as replacement for the current selection form elements, with the input schema instead of the type option callback specifying which kind of values is allowed.

Once implemented, the elimination of type options in favor of variable input schemas will enable similar flexibility for input slots as is already available for operation outputs: the use of pre-assigned values and schema information to affect the number and schema of input slots, which is especially important for providing capable element mappings.

5.2.2 Conditional execution

The current implementation is able to abort execution gracefully when an error occurs, along with proper use of error messages. However, for more flexible pipeline creation it would be desirable to enable conditional execution of operations depending on whether

an operation succeeds or not. This task involves defining ''on-error'' execution paths and a way to merge the results of both paths back into a single path. Merging will be the harder problem because pipelines are not currently able to specify sources from two different operation outputs (i.e. paths) and also operations are not currently prepared to handle different input schemas for the same input slot. Further research is necessary to solve this problem in a proper way.

As a performance improvement, executed operations should be told by the executing pipeline operation which output slots are connected to other connection targets so that unnecessary calculations can be avoided.

5.2.3 Reducing data copies

While the prototype does transport the original form of data objects across operations, the immutability constraint of data wrappers requires operations to clone those objects when any changes occur on the wrapped data. This is unavoidable in the general case, but many times it is not necessary to preserve the original form because no further operation in the execution path needs to access the original data anymore.

By introducing reference counters and copy-on-write semantics for data wrappers, the framework could be optimized for this specific case so that copies only need to be created when more than one operation requires access to a single data entity. Special care has to be taken with reference counting for child and parent elements that are wrapped in different data wrappers but are part of the same object.

Alternatively, the immutability constraint including the capability to create 1:n connections could be dropped and replaced by an explicit operation that simply duplicates its input data. This solution is easier to implement, but also requires more work and understanding of data flows from the user.

5.2.4 Schema propagation for embedded pipelines

Transformations UI can edit a pipeline only as an independent entity with no assumptions towards its surroundings; its pipeline parameters only relay the minimal requirements of the connected operation inputs. This also means that any schema information from outside cannot be used in the pipeline. If a schema is provided by an operation in

a higher-level pipeline, the edited pipeline is not able to make use of that schema information and needs to fetch the same schema information by other means.

This characteristic is intended for reusable pipelines, but pipelines that are designed to work as specialized part of a single other one should be able to make use of the schema. In order to make this work, the user interface needs to distinguish between reusable and embedded pipelines (the latter not being stored in the database directly but only as part of its containing pipeline) and offer a way to edit embedded pipelines as well. A proper implementation of this feature depends on variable input schemas (see 5.2.1). Furthermore, the user interface needs to ensure that the containing pipeline is known when editing the embedded one, in order to be able to perform the necessary schema propagation.

5.2.5 User interface improvements

Unrelated to actual API capabilities, the user interfacing part of the prototype presents much potential for improvement. Its main shortcoming is the static presentation of operation blocks, usage of JavaScript/AJAX for moving and interacting with these blocks would greatly benefit usability. Other areas that can improve the user experience include a larger set of data widgets (especially for list/array/object data) and a tool to import and export pipelines. Additional use cases can be made accessible by offering different methods of pipeline invocation, such as pipelines executing on schedule or when a certain web page is accessed in Drupal.

5.2.6 Operations

Naturally, there is no limit to the amount and capabilities of operation plugins. Third-party modules can provide operations that work on their own private data, while the set of base operations can still be improved. General functionality that would be worthwhile to implement as operations include XML schema parsing, RDF integration, support for direct SQL queries, and date/time conversion facilities. On the Drupal side, support for more CCK field types and other kinds of Drupal data (such as users or taxonomies) is desirable.

Chapter 6

Conclusion and Outlook

This work describes considerations and challenges for implementing a data transformation framework that incorporates schema information and data wrappers into an architecture that otherwise resembles existing ETL tools. The aim of these concepts is to maintain the generality and flexibility of the ETL approach while enabling more accurate preservation of input data throughout the design and execution of a transformation pipeline. A secondary goal was to provide the Drupal open source community with a flexible framework that is able to reduce the amount of modules that re-implement import/export functionality in a way that is not reusable by other modules.

In order to create a viable prototype, existing research (chapter 2) and related software as well as targeted data models (chapter 3) have been investigated. The resulting prototype is a Drupal module that is extensible by plugins and consists of data wrappers, schemas, operations and pipelines as building blocks of the architecture. Operations and pipelines are common elements in ETL software, whereas schema information is not normally present in this domain apart from schemas that denote a list of simple types which maps well to database tables. The prototype's notion of data wrappers is not found in other frameworks mainly because of performance considerations; wrappers in traditional data integration provide abstracted access only for describing and fetching data but do not cover internal storage.

As its main achievement, this work demonstrates that the usage of data wrappers as data exchange format between operations allows to create pipelines which implement load/modify/save workflows, rather than having to re-assemble exported data from scratch. The implied schema information helps improving type safety and provides

knowledge about the internal structure of nested data, which can be used by some operations and needs to be propagated throughout the pipeline.

Chapter 5 shows that full support of schema propagation is hard, and introduces significant complexity into the framework and operation plugins. While remaining issues related to data wrappers can be solved with a reasonable amount of effort, it is not clear whether full support of schema capabilities is reconcilable with the goal of simplicity and low-effort extensibility. Future work might either involve further improvements in schema support, or it might limit the use of schema information in favor of operation-specific replacements for the intended functionality.

The prototype is a working system that can be used in real-world scenarios, and has been published on drupal.org to invite further adoption and contributions. It has been successfully deployed to a Drupal installation of the initial key user, Austrian company Pro.Karriere, who recognize the framework's potential and are committed to supporting further progress. Enabling both generic and targeted user interfaces, the module is well-positioned to consolidate the scattered range of import/export modules for Drupal, so that code sharing is increased by building on the same API.

To the best of my knowledge, Transformations is the first open source PHP-based tool that implements a format-agnostic data transformation framework, and provides new possibilities to the Drupal community. As development continues, the module is destined to cover a larger number of use cases and improve flexibility and usability for its end users. Now that the framework is in place, it will be interesting to see where the software is headed to, in terms of development as well as in terms of usage and uptake in the community.

Appendix A

Acronyms

AJAX Asynchronous JavaScript And XML (reloading only parts of a web page)

CCK Content Construction Kit (Drupal module, allows flexible node structures)

CMS Content Management System (general-purpose dynamic web application)

CSS Cascading Style Sheets (format describing the looks of an (X)HTML document)

CSV Comma Separated Values (common exchange format for table-based data)

DTD Document Type Definition (original approach to XML schema definitions)

ETL Extract, transform, load (important process in data warehousing)

FTP File Transfer Protocol (mostly used for file management and downloads)

HTTP Hypertext Transfer Protocol (mostly used for transferring web pages)

IETF Internet Engineering Task Force (standards organization)

JSON JavaScript Object Notation (lightweight data exchange format)

PHP PHP Hypertext Processor (dynamically typed programming language)

RFC Request For Comments (internet standards if published by the IETF)

RDF Resource Description Framework (subject/predicate/object-based data model)

RDFS RDF Schema (class-based type system for RDF)

RDBMS Relational Database Management System (often just called "database")

RSS XML/RDF-based data format for news feeds, acronym depending on the version (2.0: Really Simple Syndication, 1.0: RDF Site Summary, 0.9x: Rich Site Summary)

SPARQL Simple Protocol And RDF Query Language (with an SQL-inspired syntax)

SQL Structured Query Language (to select and manipulate data in RDBMS)

TSV Tab Separated Values (less common exchange format for table-based data)

W3C World Wide Web Consortium (standards organization)

XML Extensible Markup Language (you knew that one, didn't you)

XML-RPC Remote Procedure Calling protocol over XML (used by web service APIs)

XHTML HTML reformulated as an XML format

Appendix B

Bibliography

[ABL+08] Sören Auer, Christian Bizer, Jens Lehmann, Georgi Kobilarov, Richard Cyganiak, and Zachary Ives. DBpedia: A nucleus for a web of open data. In *Aberer et al. (Eds.): The Semantic Web, 6th International Semantic Web Conference, 2nd Asian Semantic Web Conference, ISWC 2007 + ASWC 2007*, volume 4825/2008 of *Lecture Notes in Computer Science*, pages 722–735, Berlin / Heidelberg, 2008. Springer.

[ACM97] Serge Abiteboul, Sophie Cluet, and Tova Milo. Correspondence and translation for heterogeneous data. In Foto N. Afrati and Phokion G. Kolaitis, editors, *Database Theory - ICDT '97, 6th International Conference, Delphi, Greece, January 8-10, 1997, Proceedings*, volume 1186 of *Lecture Notes in Computer Science*, pages 351–363. Springer, 1997.

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers, Principles, Techniques, and Tools*. Addison-Wesley, Boston, MA, USA, 1986.

[BC07] Kim Bartkus and HR-XML Consortium. Staffing Exchange Protocol 2.5, April 2007. Online, accessed 07.10.2008, `http://ns.hr-xml.org/2_5/HR-XML-2_5/SEP/StaffingExchangeProtocol.html`.

[BDBD+00] Gabe Beged-Dov, Dan Brickley, Rael Dornfest, Ian Davis, Leigh Dodds, Jonathan Eisenzopf, David Galbraith, R.V. Guha, Ken MacLeod, Eric Miller, Aaron Swartz, and Eric van der Vlist. RSS 1.0 specification, December 2000. Online, accessed 07.10.2008, `http://purl.org/rss/1.0/spec`.

[BM04] Paul V. Biron and Ashok Malhotra. XML Schema part 2: Datatypes second edition, October 2004. Online, accessed 13.10.2008, `http://www.w3.org/TR/xmlschema-2/`.

[BM07] Dan Brickley and Libby Miller. FOAF vocabulary specification 0.91, November 2007. Online, accessed 14.10.2008, http://xmlns.com/foaf/spec/20071002.html.

[Boa07] RSS Advisory Board. RSS 2.0 specification (version 2.0.10), October 2007. Online, accessed 07.10.2008, http://www.rssboard.org/rss-2-0-10.

[BP96] Nacer Boudjlida and Olivier Perrin. An open approach for data integration. In *OOIS '95, 1995 International Conference on Object Oriented Information Systems, 18-20 December 1995, Dublin, Ireland, Proceedings*, pages 94–98. Springer, 1996.

[CHS+95] M. J. Carey, L. M. Haas, P. M. Schwarz, M. Arya, W. F. Cody, R. Fagin, M. Flickner, A. W. Luniewski, W. Niblack, D. Petkovic, J. Thomas, J. H. Williams, and E. L. Wimmers. Towards heterogeneous multimedia information systems: the garlic approach. In *RIDE '95: Proceedings of the 5th International Workshop on Research Issues in Data Engineering-Distributed Object Management (RIDE-DOM'95)*, pages 124–131, Washington, DC, USA, 1995. IEEE Computer Society.

[CM01] James Clark and Murata Makoto. RELAX NG specification, December 2001. Online, accessed 06.10.2008, http://www.oasis-open.org/committees/relax-ng/spec-20011203.html.

[Dub03] ISO 15836:2003(E) – Information and documentation - The Dublin Core metadata element set. International standard, published, International Organization for Standardization (ISO), 2003.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns*. Addison-Wesley Professional, Boston, MA, USA, 1995.

[HMN+99] Laura M. Haas, Renée J. Miller, B. Niswonger, Mary Tork Roth, Peter M. Schwarz, and Edward L. Wimmers. Transforming heterogeneous data with database middleware: Beyond integration. *IEEE Data Engineering Bulletin*, 22(1):31–36, 1999.

[HRO06] Alon Halevy, Anand Rajaraman, and Joann Ordille. Data integration: The teenage years. In *VLDB '06: Proceedings of the 32nd International Conference on Very Large Data Bases*, pages 9–16. VLDB Endowment, 2006.

[HW03] Gregor Hohpe and Bobby Woolf. *Enterprise Integration Patterns: Designing,*

Building, and Deploying Messaging Solutions. Addison-Wesley Professional, Boston, MA, USA, October 2003.

[Len02] Maurizio Lenzerini. Data integration: a theoretical perspective. In *PODS '02: Proceedings of the twenty-first ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 233–246, New York, NY, USA, 2002. ACM.

[LRO96] Alon Y. Levy, Anand Rajaraman, and Joann J. Ordille. Querying heterogeneous information sources using source descriptions. In *VLDB '96: Proceedings of the 22th International Conference on Very Large Data Bases*, pages 251–262, San Francisco, CA, USA, 1996. Morgan Kaufmann Publishers Inc.

[MB08] Alistair Miles and Sean Bechhofer. SKOS Simple Knowledge Organization System reference, working draft, August 2008. Online, accessed 14.10.2008, http://www.w3.org/TR/skos-reference/.

[MBF+90] George A. Miller, Richard Beckwith, Christiane Fellbaum, Derek Gross, and Katherine Miller. Introduction to WordNet: An on-line lexical database. *International Journal of Lexicography*, 3(4):235–244, 1990.

[MPP+08] Christian Morbidoni, Danh Le Phuoc, Axel Polleres, Matthias Samwald, and Giovanni Tummarello. Previewing semantic web pipes. In *The Semantic Web: Research and Applications*, volume 5021/2008 of *Lecture Notes in Computer Science*, pages 843–848. Springer Berlin / Heidelberg, 2008.

[MZ98] Tova Milo and Sagit Zohar. Using schema matching to simplify heterogeneous data translation. In *VLDB '98: Proceedings of the 24rd International Conference on Very Large Data Bases*, pages 122–133, San Francisco, CA, USA, 1998. Morgan Kaufmann Publishers Inc.

[RB01] Erhard Rahm and Philip A. Bernstein. A survey of approaches to automatic schema matching. *The VLDB Journal*, 10(4):334–350, December 2001.

[RD00] Erhard Rahm and Hong Hai Do. Data cleaning: Problems and current approaches. *IEEE Data Engineering Bulletin*, 23(4):3–13, 2000.

[SE05] Pavel Shvaiko and Jérôme Euzenat. A survey of schema-based matching approaches. In *Journal on Data Semantics IV*, volume 3730/2005 of *Lecture Notes in Computer Science*, pages 146–171. Springer Berlin / Heidelberg, 2005.

[Sha05] Yakov Shafranovich. RFC 4180: Common format and MIME type for comma-separated values (CSV) files, October 2005. Online, accessed 06.10.2008, http://www.ietf.org/rfc/rfc4180.txt.

[TAR07] Andreas Thor, David Aumueller, and Erhard Rahm. Data integration support for mashups, July 2007. Sixth International Workshop on Information Integration on the Web (IIWeb-07). Vancouver, Canada.

[Van08] John K. VanDyk. *Pro Drupal Development, Second Edition*. Apress, Berkeley, CA, USA, August 2008.

[VSGT03] Panos Vassiliadis, Alkis Simitsis, Panos Georgantas, and Manolis Terrovitis. A framework for the design of ETL scenarios. In Johann Eder and Michele Missikoff, editors, *Advanced Information Systems Engineering, 15th International Conference, CAiSE 2003, Klagenfurt, Austria, June 16-18, 2003, Proceedings*, volume 2681 of *Lecture Notes in Computer Science*, pages 520–535. Springer, 2003.

[WH07] Jeffrey Wong and Jason I. Hong. Making mashups with Marmite: towards end-user programming for the web. In *CHI '07: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 1435–1444, New York, NY, USA, 2007. ACM.