Converging Web and Desktop Data with Konduit

Converging Web and Desktop Data with Konduit

Laura Drăgan[1], Knud Möller[1], Siegfried Handschuh[1], Oszkár Ambrus[1], and Sebastian Trüg[2]

[1] Digital Enterprise Research Institute, National University of Ireland, Galway
firstname.lastname@deri.org
[2] Mandriva S.A., France
strueg@mandriva.com

Abstract. In this paper we present Konduit, a desktop-based platform for visual scripting with RDF data. Based on the idea of the semantic desktop, non-technical users can create, manipulate and mash-up RDF data with Konduit, and thus generate simple applications or workflows, which are aimed to simplify their everyday work by automating repetitive tasks. The platform allows to combine data from both web and desktop and integrate it with existing desktop functionality, thus bringing us closer to a convergence of Web and desktop.

1 Introduction

With the Semantic Web gaining momentum, more and more structured data becomes available online. The majority of applications that use this data today are concerned with aspects like search and browsing. However, a greater benefit of structured data is its potential for reuse: being able to integrate existing web data in a workflow relieves users from the investment of creating this data themselves. On the other hand, when it comes to working with data, users still rely on desktop-based applications which are embedded in a familiar environment. Web-based applications either simply do not exist, or have shortcomings in terms of usability. They can only access web data, and do not integrate with data that users might already have on their own desktop, let alone with other applications. Even considering that it may be beneficial for users to publish some desktop data online, releasing all their data on the web may raise significant privacy issues. Instead, what is needed is a way of accessing structured web data from the desktop, integrate it with existing desktop-data and applications and work with both in a unified way.

The Semantic Desktop through projects such as Nepomuk now opens up new possibilities of solving this problem of integrating data and functionality from both web and desktop. On the Semantic Desktop, data is lifted from application-specific formats to a universal format (RDF) in such a way that it can be interlinked across application boundaries. This allows new ways of organizing data, but also new views on and uses of arbitrary desktop data. What is more, because desktop data is now available in a web format, it can also be interlinked and

processed together with genuine web data. While the unified data model makes this scenario easier than it previously was, implementing it would ordinarily still require an experienced developer, who would use a full-edged programming language to create applications that manipulate and visualize RDF data. With current tools, casual or naive users would not be able to perform such tasks.

In this paper, we present an approach for mashing up RDF data, which can originate from either the web or, through the Semantic Desktop, from arbitrary desktop applications. While the individual components that make up our approach are not new in themselves, we believe that the combination is new and opens up possibilities that have not been available before.

2 Background

Our work is based on and influenced by several existing technologies, such as the Semantic Desktop, Unix pipes, scripting languages, visual programming & scripting and dataflow programming. In the following we will describe these technologies.

Our approach assumes that we are working on a **semantic desktop** [1], rather than a conventional one. As discussed earlier, this means that data in application-specific formats has been lifted to a uniform data format, such as RDF or, in the case of the Nepomuk project [3], to an extension such as NRL[3] (in the remainder of this paper, we mean a semantic representation language when we say RDF). Representing desktop data in a uniform format means that it can be interlinked and processed in a uniform way across the desktop, but also that it can be interlinked and processed with web data in the same way.

For our application the implementation of choice of the Semantic Desktop is Nepomuk-KDE[4], developed during the Nepomuk project as part of the K desktop environment. However, also more mainstream products, such as Spotlight technology of Mac OS X are a step towards a unified view on all desktop data.

The concept of **pipes** has been a central part of UNIX and its derivatives since 1973, when it was introduced by M. Doug McIlroy. The basic idea of pipes is that individual processes or programs can be chained into a sequence by connecting them through the operating systems standard streams, so that the *stdout* of one process feeds into its successor's *stdin*. In this way, tasks which require the functionality from different applications or data from different sources can elegantly be combined into a single workflow.

Scripting languages such as Perl and Unix shell allow rapid application development and a higher level of programming. They represent a very different style of programming as compared to system programming languages like C or Java, mainly because they are designed for "gluing" applications [8]. The libraries provided by most scripting languages are highly extensible, new components being added as the need for them arises. Being weakly typed is another defining characteristic of scripting languages that Konduit employs.

[3] `http://www.semanticdesktop.org/ontologies/nrl/` (26/02/2009)
[4] `http://nepomuk.kde.org` (26/02/2009)

As a form of end-user programming, **visual programming** (VP) is targeted at non-experts who want to be able to automate simple processes and repetitive tasks, without having to learn the complexities of a full-fledged programming language. In visual programming users construct the program not by writing source code, but instead by arranging and linking visual representations of components such as data sources, filters, loops, etc. In other words, "a visual programming system is a computer system whose execution can be specified without scripting" [5] — "scripting" here in the traditional sense of writing lines of source code.

Recently, VP has gained some popularity in the form of Yahoo Pipes[5]. In allusion to UNIX Pipes, Yahoo Pipes allows the user to visually compose workflows (or pipes) from various ready-made components. Inputs and outputs are mostly news feed-like lists of items. Being a Web application, Yahoo Pipes is limited in that it operates on Web data only, in formats such as RSS or Atom. Another application that supports a wider range of (Semantic) Web data standards and also tightly integrates with the SPARQL query language is SparqlMotion[6]. Because of the simplicity and typical small-scale scope of software like Yahoo Pipes, SparqlMotion and also Konduit, they are often being tagged with the term *visual scripting* instead of VP.

Closely related to our approach are the Semantic Web Pipes [6], which apply the Yahoo Pipes look and feel and functionality directly to Semantic Web data. Also here, SPARQL is an integral component to define the functionality of the individual building blocks. A crucial difference between SparqlMotion and Semantic Web Pipes on the one hand and Konduit on the other is that they have a clear focus on Web data and do not integrate desktop data or application functionality.

The concept of designing workflows by chaining a set of components through their inputs and outputs is related to a form of programming called **dataflow programming** (e.g., [7]). Unlike the more conventional paradigm of imperative programming, a program in dataflow programming does not consist of a set of instructions which will essentially be performed in sequence, but instead of a number of interconnected "black boxes" with predefined inputs and outputs. The program runs by letting the data "flow" through the connections. As soon as all inputs of a particular component are valid, a component is executed.

2.1 Related Work

Apart from those mentioned above, there are a number of other systems which are related to Konduit. WebScripter [9] is an application that allows users to create reports in a spreadsheet-like environment from distributed data in the DAML (DARPA Agent Markup Language) format. Unlike our approach, WebScripter is based on the now more or less extinct DAML, and offers neither a

[5] `http://pipes.yahoo.com/` (26/02/2009)

[6] `http://composing-the-semantic-web.blogspot.com/2007/11/sparqlmotion-visual-semantic-web.html` (26/02/2009)

visual composition environment nor the option to connect to desktop functionality. Potluck [4] is a web-based platform for visually mixing structured data from different sources together, even if the data does not conform to the same vocabulary or formatting conventions. An important restriction is the fact that only data from sites which are hosted using the Exhibit[7] platform can be merged. Potluck is geared towards data integration, and therefore does not offer any of the workflow capabilities we implement in Konduit.

3 Konduit Components and Workflows

With Konduit we want to allow casual users to build simple programs in order to perform and automate everyday tasks on RDF data. Konduit provides a collection of useful components ready for immediate use. The components offer individual units of functionality and are represented visually as blocks. They are connected through input and output slots, and in this way the flow of the program is defined. In order to keep simple the task of connecting components, the only data that flows through the workflow is RDF. This condition insures that each component always fulfils the minimal requirement for dealing with its input. Obviously, components may be specialized with respect to the actual vocabulary on which they can operate and will decide at runtime if and how it deals with the incoming RDF. By neither allowing different kinds of data (e.g., text, numbers, lists, images, etc.), nor typing the RDF data with respect to the vocabularies they use, we stay very close to the original UNIX pipes concept, where data is always an untyped bytestream on the one of the standard streams *stdin* or *stdout*, and where it is up to each process or program how to handle it (see Fig. 1). Konduit is implemented as a desktop-based application for the

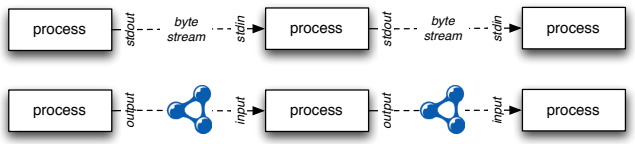

Fig. 1: Unix pipes and RDF pipes

Linux desktop environment KDE4, and is based on Plasma[8]. The architecture is plugin-based, so that each component is realised as a plugin into the Konduit platform. Technically, Konduit plugins are also so-called "Plasma applets". Therefore designing and implementing new ones is quite straightforward (from the point of view of a KDE developer); and although all existing Konduit plugins have been written in Qt/C++, to write new ones can be done using the Ruby,

[7] `http://simile.mit.edu/exhibit/`
[8] `http://plasma.kde.org/` (26/02/2009)

Python or Java bindings of Qt. We expect that new plugins will be developed by external power users, as the need for them arises. As Plasma applets, the Konduit plugins can be loaded and used as independent applications directly on the desktop, without being restricted to the Konduit workspace. The workspace is not unlike a drawing canvas, on which the components can be dropped from a lateral toolbar. On this "drawing area" the user can connect the input slots to output slots of different components, move the blocks around, set their parameters and in this way build small applications.

Konduit makes use of the semantic desktop features that come as part of Nepomuk implementation in KDE4, especially the Soprano RDF framework[9]. Soprano is also used to store the saved workflows and black boxes as RDF in a repository (with the given connections and values for configuration parameters).

3.1 Components

Formally, a component is defined by the following parameters: (i) a set of RDF input slots I, (ii) a set of RDF output slots O, (iii) a set of parameters P which allow for user input in the workflow, (iv) a unit of functionality F, which works on the input I and generates the output O. The parameters P influence the behaviour of F.

Definition 1. *Component = (I, O, P, F)*

The number of input and output slots is not fixed and can be 0 or more. Depending on the number of slots, components can be grouped in three categories: sources, sinks, and ordinary components. *Sources* are components that do not have any inputs. They supply the workflow with data. Because the data graphs can be merged, there can be more than one source for any workflow. Typical examples of sources are connectors to RDF stores, file (URL) input components, or converters from other, non-RDF formats. *Sinks* are components that do not have any outputs. They represent the final point(s) of any workflow. Examples of sink components are application adaptors, serializers (file output components) and visualizers. Unlike in dataflow programming where a component is run as soon as all inputs are valid, the Konduit workflows are activated from a sink component, usually by clicking on an activation button.

Ordinary components, can be further classified according to the kind of functionality F they contain.

- **Merger** - combines the input graphs into a single output graph
- **Duplexer** - duplicates the input graph to two outputs.
- **Transformer** - applies a transformation on the input graph and outputs the resulting graph.

An important aspect of our approach is directly tied to the fact that all inputs and outputs are RDF graphs. As a result, any workflow can itself become

[9] `http://soprano.sourceforge.net/` (26/02/2009)

a component, meaning that workflows can be built recursively. In this way, it is possible to create a library of specialised components (which we call *blackboxes*), based on the combination of basic components. We will pick this idea up again in Sect. 3.2.

Sources. Sources are a special type of components that do not have any input slots. There is always at least a source at the start of any workflow.

There is a dedicated source component for reading data from the local Nepomuk RDF repository. This source extracts the desktop data according to a SPARQL construct query given as parameter. The Nepomuk source element has a variant that is meant to help the user create the SPARQL query in a friendlier way, by the means of a smart wizard, with autocompletion and suggestions. Another basic source component is the file input source, which takes

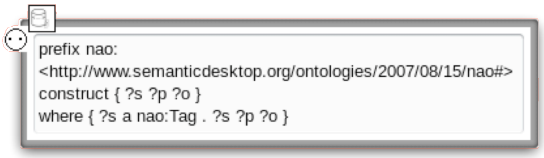

Fig. 2: Konduit Nepomuk Source that finds all the data about the tags from the local Nepomuk repository.

a URL as a parameter. The URL can point to a file (network is transparent so the path can be local or remote) or to a SPARQL endpoint (see Fig. 3). This component takes as parameter the expected serialization of the graph. For parsing it uses the parsers made available by the Soprano library. There are several

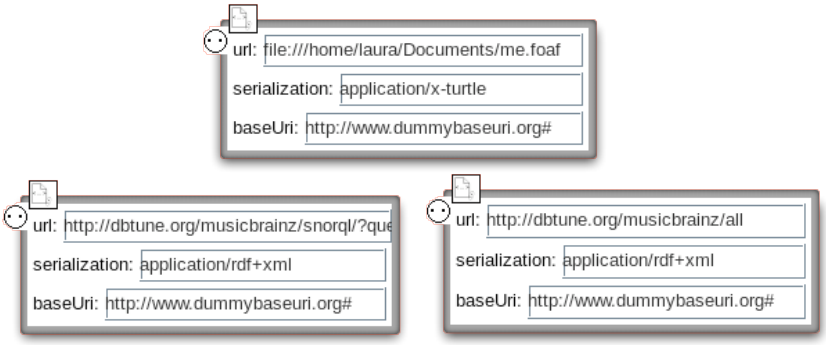

Fig. 3: The three uses of the Konduit File Input Source.

components that transform non-RDF data to RDF. The literal input takes any text given as parameter and transforms it to a RDF graph containing exactly one triple:

```
<http://www.konduit.org/elements/LiteralValue/data>
    <http://www.w3.org/2000/01/rdf-schema#comment>
        "string data"^^<http://www.w3.org/2001/XMLSchema#string>
```

The literal file input creates the same kind of triple, using as the string value the content of the file given as parameter.

Transformers. The most basic and simple transformer component is the filter element. It changes the input graph according to a SPARQL construct query given as parameter. The filter element can be saved with fixed queries and thus create specialized converters from one vocabulary to another. Another useful transformer is the duplicate remover component, which as the name suggests, outputs each unique triple from the input graph exactly once and discards all the duplicates.

Visualizers. The visualizer components display the RDF data received as input in various forms. So far there are only two components of this type: the data dump sink which shows the graph as quadruples in a separate window; and the data table sink which creates tables for each class of resource found in the input graph, each table having on each row one data for one instance in the graph. The columns are given by the properties of the class shown in each table.

Application adaptors. Application adaptors call the functionality of an external application or protocol with the data given in the input graph.

One such adaptor is the mailer element. It takes as input several graphs of data: one of foaf:Persons with mbox and name, one with the list of files to attach to the sent emails, a template for the message and a subject.

Another adaptor is the scripter element which passes the input RDF graph as input to a script available on the desktop. There is no restriction regarding the nature of the script or the language in which it is written, as long as it is executable, it takes RDF as input and it outputs RDF as well. The serialization for the input and output must be the same and it can be specified as a parameter.

3.2 Workflows

A workflow is defined by specifying (i) a set of components C, (ii) a function f defined from the set of all the inputs of the components of C to the set of all the outputs of the components of C and the *nil* output. The function f shows how the components of C are connected. The inputs that are not connected have a *nil* value of f; the outputs that do not represent a value of f are not connected.

Definition 2. *Workflow = (C, f) where f: inputs(C) → outputs(C) ∪ { nil }*

Workflows can be saved and reused. Saving a workflow implies saving all the components that have at least one connection to the workflow, as well as their existing connections, parameters and layout. There is no restriction that the components should be completely connected, so there can be input or output slots that remain open. A saved workflow can be reopened and modified by adding to it or removing components, or changing connection or parameters and thus obtaining different workflows with minimum effort.

Even the simple workflows can have numerous components, the more complex ones having tens of components can become too big to manage in the workspace provided by the application. To aid the user with handling large and complex workflows, we added modularization to Konduit. Workflows can thus be saved as reusable components, which we call *blackboxes* and which are added to the library of available elements. Blackboxes can be used afterwards in more complex workflows. This can be done recursively as more and more complexity is added. The inputs and outputs of blackboxes must be marked in the original workflow by special input and output components (as illustrated in Fig. 4).

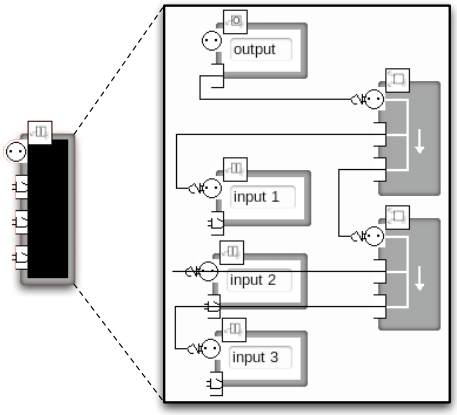

Fig. 4: Looking inside a tri-merger black-box created by concatenating two merger elements.

4 Use Case

The following example illustrates what Konduit can do for the user of a *semantic desktop*.

John is a music enthusiast. He enjoys having his music collection organized, and if he likes an artist he will try to find that artist's entire discography. Whenever he discovers a new singer he creates a file with that singer's discography and

marks which albums or songs he owns and which he does not. This task requires usually many searches - on the web as well as on John's own computer. Some of the common problems he runs into are: on the web the information he needs is spread across several web pages which need to be found; on his computer the music files are spread over several folders, and he would have to manually check each file to mark it as owned.

This example highlights a number of important aspects that our approach addresses, and illustrates how a tool such a Konduit can be used:

- Accessing and processing desktop data: John uses the semantic desktop offered by Nepomuk on KDE4 so he has his music library metadata stored in the Nepomuk repository and can therefore be processed by Konduit.
- Accessing and processing web data: Services[10] expose their data as RDF, which means that our system can use it.
- Merging desktop and web data: Since both kinds of data sources use a unified data model, Konduit can simply mash both together.
- Using desktop functionality: Since our approach is desktop-based, we can easily access and integrate the functionality of arbitrary desktop applications or run local scripts that are normally executable on the desktop (with the restriction of taking as input and outputting RDF).

Three main parts of the workflow stand out: preparation of the data found online, preparation of the data found locally on John's desktop and the generation of the file. For the first two parts we create sub-workflows which we save as blackboxes and use them in the final workflow. Both blackboxes take as input the name of the artist and output album and track data, one from the desktop and the other from the web.

Desktop data. To access the local music metadata we need a Nepomuk source component. It will return the graph of all songs found in the Nepomuk repository, with title, artist, album, track number and the URL of the file storing the song. This graph needs to be filtered so that only the songs by the specified author remain. For it we use a filter element.

Web data. We use the SPARQL endpoint provided by the musicbrainz service[11] to retrieve music information. To connect to it we need a file input source with a query that takes data about artists, albums and tracks. The graph returned by the source has to be filtered by the artist name. This is done with a filter component that has also the function of a vocabulary converter, as it takes in data described using the Music Ontology [12] and creates triples containing the same data described with the Xesam ontology [13].

[10] such as `http://dbtune.org/musicbrainz/`
[11] `http://dbtune.org/musicbrainz/sparql`
[12] `http://purl.org/ontology/mo/`
[13] `http://xesam.org/main/XesamOntology`

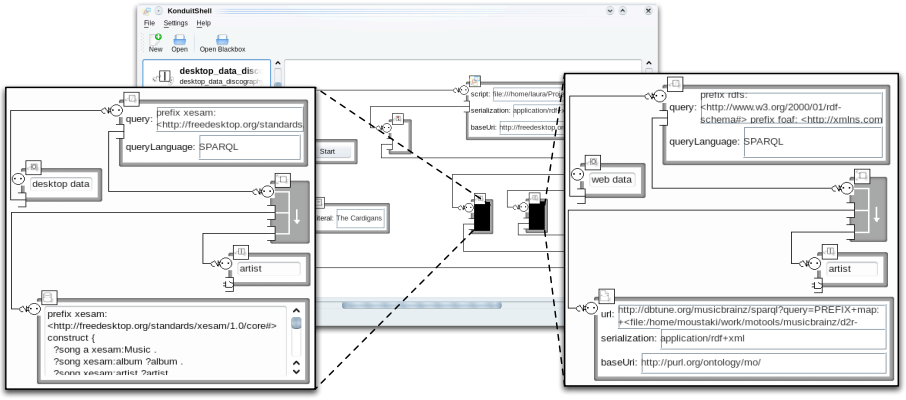

Fig. 5: The entire discography generator workflow.

Running the script. The scripter will take as input a graph constructed from the two subgraphs: one containing the data about the artist extracted from the web and the other the data about the artist available on the desktop. Both graphs contain xesam data. The merged outputs are first passed through a duplicate remover component to eliminate the redundant triples. The script takes the resulting graph of albums and tracks for the given artist and generates a file containing the discography. The RDF output of the script contains the path to the generated file, and is used by a File Open component to display the discography in the system default browser. The final workflow is depicted in Fig. 5 and the generated discography file in Fig. 6. A more detailed description of the workflow, including the SPARQL queries that are used and the script can be found at [2]

5 Discussion and Future Work

In this section, we will discuss a number of issues related to our conceptual approach in general, as well as to our Konduit implementation in particular.

We have argued that we restrict the kind of data that can flow within a workflow to be only RDF. By only allowing one kind of data, we keep the model simple and elegant. However, in reality we will often want to deal with other kinds of data (text, URLs, images, etc). At the moment, we handle these cases through component parameters, but this solution often feels rather awkward. We plan to study further whether adding support for other types than RDF will justify the increase in complexity.

Currently we do not have support for control flow components (loops, boolean gates, etc). On the one hand, including such features would certainly make our approach much more versatile and powerful and may be an interesting line of development for the future.

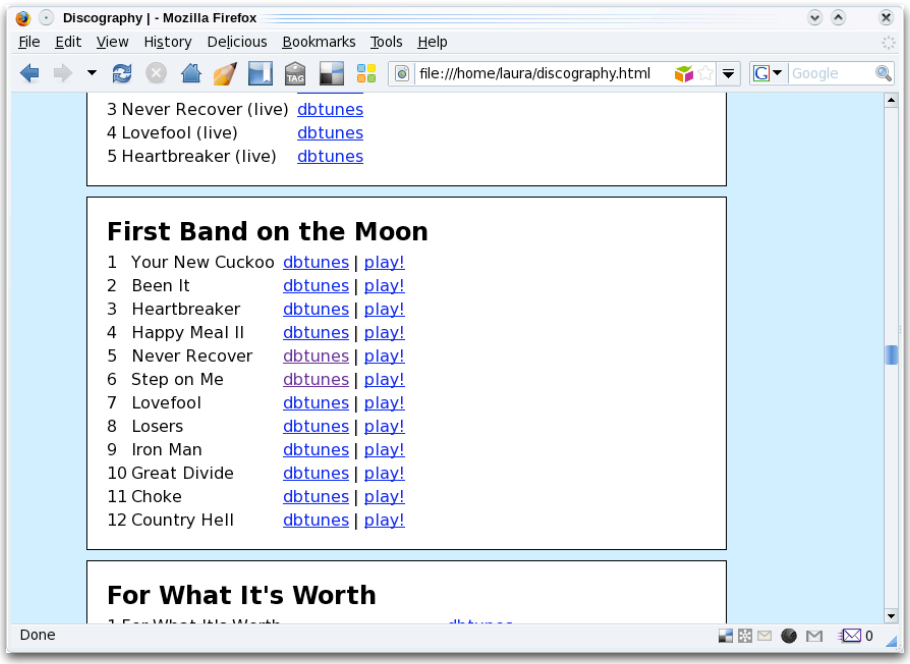

Fig. 6: The generated discography page for *The Cardigans*.

Some of the basic components available for Konduit require previous knowledge of writing SPARQL queries. Since the queries given as parameters to the source and filter elements can influence the performance of the entire workflow, we recognize the need for a smart query editor that is suitable for naive users. Our solution to support end users in creating queries is based on autocompletion, however, in order to make the system more accessible, we think it will be necessary to introduce a different kind of interface, which would abstract away from the actual syntax altogether and model the query on a higher level. Such an interface would possibly still be of a graphical nature, but without simply replicating the SPARQL syntax visually. Alternatively or additionally, a natural language interface would be promising direction for further research.

6 Conclusion

We have presented an approach for enabling casual, non-technical users to build simple applications and workflows from structured data. To simplify the building process, we have chosen a visual scripting approach, which is inspired by software such as Yahoo Pipes. We expect that users will benefit mostly from our approach if they operate in a Semantic Desktop-like environment, where they will have access to the data and functionality they are used to and have to work with on

a daily basis. However, our approach and implementation also enable users to integrate data and functionality from their desktops with data from the Web, thus representing a step towards the convergence of those two domains.

Acknowledgements

The work presented in this paper was supported (in part) by the Líon project supported by Science Foundation Ireland under Grant No. SFI/02/CE1/I131 and (in part) by the European project NEPOMUK No FP6-027705.

References

1. S. Decker and M. R. Frank. The networked semantic desktop. In C. Bussler, S. Decker, D. Schwabe, and O. Pastor, editors, *WWW Workshop on Application Design, Development and Implementation Issues in the Semantic Web*, May 2004.
2. L. Dragan and K. Möller. Creating discographies with Konduit, 2009. `http://smile.deri.ie/konduit/discography`.
3. T. Groza, S. Handschuh, K. Möller, G. Grimnes, L. Sauermann, E. Minack, C. Mesnage, M. Jazayeri, G. Reif, and R. Gudjonsdottir. The NEPOMUK project — on the way to the social semantic desktop. In T. Pellegrini and S. Schaffert, editors, *Proceedings of I-Semantics' 07*, pages pp. 201–211. JUCS, 2007.
4. D. F. Huynh, R. C. Miller, and D. R. Karger. Potluck: Semi-ontology alignment for casual users. In K. Aberer, K.-S. Choi, N. Noy, D. Allemang, K.-I. Lee, L. Nixon, J. Golbeck, P. Mika, D. Maynard, R. Mizoguchi, G. Schreiber, and P. Cudré-Maroux, editors, *6th International Semantic Web Conference and 2nd Asian Semantic Web Conference, ISWC+ASWC2007, Busan, Korea*, volume 4825 of *LNCS*, pages 903–910, Heidelberg, November 2007. Springer.
5. T. Menzies. Visual programming, knowledge engineering, and software engineering. In *Proc. 8th Int. Conf. Software Engineering and Knowledge Engineering, SEKE*. ACM Press, 1996.
6. C. Morbidoni, D. L. Phuoc, A. Polleres, and G. Tummarello. Previewing semantic web pipes. In S. Bechhofer, editor, *Proceedings of the 5th European Semantic Web Conference (ESWC2008), Tenerife, Spain*, volume 5021 of *LNCS*, pages 843–848. Springer, June 2008.
7. L. Orman. A multilevel design architecture for decision support systems. *SIGMIS Database*, 15(3):3–10, 1984.
8. J. K. Ousterhout. Scripting: Higher Level Programming for the 21st Century. In *IEEE Computer Magazine*, March 1998. `http://home.pacbell.net/ouster/scripting.html`.
9. B. Yan, M. R. Frank, P. Szekely, R. Neches, and J. Lopez. WebScripter: Grass-roots ontology alignment via end-user report creation. In D. Fensel, K. Sycara, and J. Mylopoulos, editors, *2nd International Semantic Web Conference, ISWC2003, Sanibel Island, FL, USA*, volume 2870 of *LNCS*, pages 676–689, Heidelberg, November 2003. Springer.