# Modular Web Queries—From Rules to Stores

Uwe Aßmann[2], Sacha Berger[1], François Bry[1], Tim Furche[1],
Jakob Henriksson[2], and Jendrik Johannes[2]

[1] Institut für Informatik, Ludwig-Maximilians-Universität München
{sacha.berger|francois.bry|tim.furche}@ifi.lmu.de
[2] Fakultät für Informatik, Technische Universität Dresden
{uwe.assmann|jakob.henriksson|jendrik.johannes}@tu-dresden.de

**Abstract.** Even with all the progress in Semantic technology, accessing Web data remains a challenging issue with new Web query languages and approaches appearing regularly. Yet most of these languages, including W3C approaches such as XQuery and SPARQL, do little to cope with the explosion of the data size and schemata diversity and richness on the Web. In this paper we propose a straightforward step toward the improvement of this situation that is simple to realize and yet effective: Advanced module systems that make partitioning of (a) the evaluation and (b) the conceptual design of complex Web queries possible. They provide the query programmer with a powerful, but easy to use high-level abstraction for packaging, encapsulating, and reusing conceptually related parts (in our case, rules) of a Web query. The proposed module system combines ease of use thanks to a simple core concept, the partitioning of rules and their consequences in flexible "stores", with ease of deployment thanks to a reduction semantics. We focus on extending the rule-based Semantic Web query language Xcerpt with such a module system though the same approach can be applied to other (rule-based) languages as well.

## 1 Introduction

As the amount and diversity of data available on the Web is constantly increasing, querying this great abundance of information is becoming more and more important. In fact, it is becoming less important to possess certain knowledge, but more important to know how to acquire it—know how to formulate a precise *query* to find the desired information. Query languages for different purposes are emerging in multitude. [2] surveys some existing query and transformation languages for Web and Semantic Web data, identifying 14 textual XML query languages and 24 for RDF metadata.

Yet, most of these languages provide very little support to the user to cope with the dramatic increase in information size and diversity. Increasing information diversity results in increase of query size and complexity, which can weigh down even experienced query programmers. It must be easy for users to partition (both conceptually and from an evaluation point of view) query programs and to make such partitioning flexible enough to allow for reuse in different contexts. This is not the case unless the query language provides some means to separate large and complex queries into smaller, properly isolated, and reusable fragments—*modules*. Such modules allow to "localize" the effect of the introduction of additional data sources or query tasks in query programs.

Thus, modules allow a *separation of concern* not just on the basis of single rules but on the basis of larger conceptual units of a query program. For example, one part of a Web application is often concerned with extracting data from a set of sources, such as a set of Web pages. At the next step, the data might have to be syndicated into a common view and format. From this syndicated data, some new implicit data could possibly be derived. Finally, the resulting data set should be displayed in an appropriate human-readable form, for example, by being displayed in a well-structured Web page (see Section 3 for an example). These different steps taken by the application have to do with different concerns of the overall realization, such as data extraction, data management and data display. Furthermore, each of the concerns deals with different schemata, but the knowledge of the schemata can be hidden and encapsulated within each concern – within each module. In contrast, exposing all these concerns in one monolithic query program not only becomes very hard to understand, but is also impossible to manage as a change in some part may affect any other part.

This work is based on ideas from [8, 1] where we propose a flexible approach (demonstrated along Datalog examples) for augmenting arbitrary languages with new levels of *abstractions*, and new constructs for authoring reusable entities. The only requirement we put on the newly introduced constructs is that their realization is already expressible in the original language, i.e., that they have a reduction semantics. In doing this we take advantage of existing software composition techniques[3] to realize the added reuse abstractions [8]. However, in this paper we do not focus on the details of composition systems, but show an application of the ideas to a concrete query language, viz. Xcerpt [11]. The semantics is derived from the formal semantics in [1].

For that language, we propose a module system that (a) demonstrates how Web query languages can profit from modules by partitioning the query program as well as its execution; (b) provides an easy, yet powerful module extension for Xcerpt that shows how well-suited rule-based languages are for component-based reuse; (c) is based on a single new concept, viz. “stores”; and (d) uses a reduction semantics exploiting the power of a language with views. This semantics enables the reuse of the existing query engine making the design of the module system easier and its deployment less time consuming.

The rest of this paper is organized around these contributions: Following a brief introduction to Xcerpt we demonstrate the need for modules or similar reuse and partitioning mechanisms by a use case on integrating (Semantic and plain old) music data on the Web. Then we introduce the module extension for Xcerpt by implementing part of the aforementioned use case. We conclude with a discussion of the semantics and realization of the module extension.

## 2 Introducing Xcerpt

We choose to demonstrate our ideas using the rule-based, Web and Semantic Web query language Xcerpt [11], which has been co-developed by some of the authors and is particularly well-suited for reuse due to its rule-based nature. This chapter is not intended

[3] Developed within the Reuseware Composition Framework (`http://reuseware.org`).

as a full introduction to Xcerpt but merely recalls some of its most relevant features for this article. For a proper introduction please see [11].

An Xcerpt program consists of a finite set of Xcerpt *rules*. The rules of a program are used to derive new, or transform existing, XML data from existing data (i.e. the data being queried). *Construct rules* are used to produce intermediate results while *goal rules* form the output of programs.

While Xcerpt works directly on XML or RDF data, it has its own data format for modeling XML documents or RDF graphs, viz. Xcerpt *data terms*. For example, the XML snippet `<book><title>White Mughals</title></book>` corresponds to the data term `book [ title [ "White Mughals" ] ]`. The data term syntax makes it easy to reference XML document structures in queries and extends XML slightly, most notably by also allowing unordered data.

For instance, in the following query the construct rule defines data about books and their authors which is then queried by the goal. Intuitively, the rules can be read as deductive rules (like in, say, Datalog): if the body (after **FROM**) holds, then the head (following **CONSTRUCT** or **GOAL**) holds. A rule with an empty body is interpreted as a fact, i.e., the head always holds.

```
GOAL
  authors [ var X ]
FROM
  book [[ author [ var X ] ]]
END

CONSTRUCT book [ title [ "White Mughals" ], author [ "William Dalrymple" ] ] END
```

Xcerpt *query terms* are used for querying data terms and intuitively describe patterns of data terms. Query terms are used with a pattern matching technique[4] to match data terms. Query terms can be configured to take partiality and/or ordering of the underlying data terms into account during matching (indicated by different types of brackets).

Query terms may also contain logic variables. If so, successful matching with data terms results in variable bindings used by Xcerpt rules for deriving new data terms. Matching, for instance, the query term `book [ title [ var X ] ]` with the XML snippet above results in the variable binding `{X / "White Mughals" }`.

*Construct terms* are essentially data terms with variables. The variable binding produced via query terms in the body of a rule can be applied to the construct term in the head of the rule in order to derive new data terms. For the example above we obtain the data term `authors [ "William Dalrymple" ]` as result.

## 3 Use case: Music aggregation with the Web Music Library

The use case illustrated in Figure 1 presents a library (called MusicLibrary) of functionality useful for coping with music and information about music found on the (Semantic) Web. At an (arguably) lower layer, information is extracted from various established Web sites like `amazon.com` or `discogs.org`. The extraction has to be handled differently for every web site, but is valuable for many users and applications. For example,

[4] Called *simulation unification*. For details of this technique, please refer to [10].

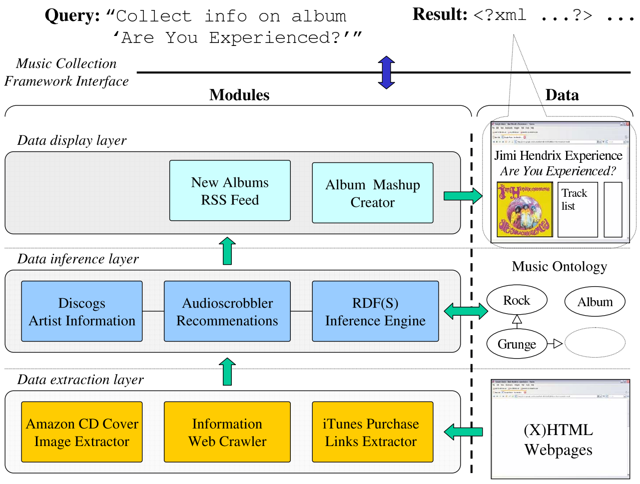


**Figure 1.** Many query languages only allow writing monolithic queries, while modular query development greatly increases reuse and ease of programming.

many of the currently established desktop music players exploit the album or CD images of Amazon to display cover art while playing back music. Encapsulating reusable queries dealing with a particular information source allow for flexible maintenance and propagation to a larger user base. The legacy information as found on external Web sites is then converted to an internal representation loosely based on the Music Ontology [7]. Music Ontology is an RDFS-based standard, hence knowledge inference and reasoning on—possibly incomplete—Music Ontology data can be achieved using an RDFS reasoner. Since such a reasoner is usable in many different fields of applications, it is implemented and provided as an Xcerpt module and included in the main library, hence allowing for its reuse. Perhaps more interesting to the end user, various modules providing pleasant visualizations of gathered information or predefined query skeletons can be provided in the library. Such modules can also be provided by third parties or, last but not least, as part of an application using the Web Music Library. We show only small extracts of the actual modules for space and presentation reasons.

### 3.1 Realizing Musical Modules in Xcerpt

How can we today realize this application in Xcerpt? In the absence of modules we have to carefully craft a single query program with a considerable number of rules (well over three dozens if we follow the basic design presented below) at each step taking great care that the rules do not, by chance, interfere with each other. Furthermore, we have to update the whole query program as soon as any information source changes, since this information is hard-coded in the program.

In the presence of a module extension, the task becomes a lot less daunting: Let us start from the top with a user program that gathers information about Jimi Hendrix from all the sources described in Figure 1. For that, it relies on a module called MusicLibrary (discussed above). The library is not a mere database, it is an interface to various ways of reasoning about musical information available on the Web. To the user the complexity remains hidden. The user just poses his query to the module without caring whether the data is extensional or intensional and how it is obtained. The module system ensures that, regardless of the actual rules and their distribution between modules, there is no chance for interference by rules of different sub-modules used within MusicLibrary.

```
IMPORT "MusicLibrary"

GOAL
  html [ body [
    h1 [ "Records by Jimi Hendrix" ],
    table [ tr [ td [ "Record" ], td [ "Year" ] ],
            all tr [ td [ var R ], td [ var Y ] ] ]
  ] ]
FROM
  in "MusicLibrary" (
    desc record { artist [ "Jimi Hendrix" ],
                  title [ var R ], year[ var Y ] } )
END
```

The MusicLibrary module itself is integrating data and knowledge of other modules the same way as the user program. It has to provide the information, and only the desired information, to the user of the module. Some rules may be necessary internally in the module to achieve the task, but should not be directly visible to the user of the module. The visible parts of the module are hence *public*, the others (implicitly) *private*.

Apart from using knowledge of other modules, modules may also receive data provided by importing modules. MusicLibrary accesses data extracted by a module gathering MusicBrainz metadata, feeds it to a module for converting that data to Music Ontology knowledge (Musicbrainz2MOFacts), and finally injects that knowledge to an RDFS reasoner (using the MO-Ontology-Reasoner module). It also accesses `discogs.org` directly and feeds the acquired data into another instance of the MO-Ontology-Reasoner. To distinguish multiple instances of the reasoner, each instance is given an alias (using the `@` construct), which can be used the same way as the module identifier when querying, or sending data to, a module. In this way, modules also give rise to *scoped* reasoning where consequences only apply in a certain scope (or module), but are not (automatically) propagated outside of that scope. In particular, knowledge in different scopes may, if considered globally, be inconsistent, but within each scope be consistent.

```
MODULE "MusicLibrary"
IMPORT "MusicBrainz"
IMPORT "Musicbrainz2MOFacts"
IMPORT "MO-Ontology-Reasoner" @ "reasoner-for-musicBrains"
IMPORT "MO-Ontology-Reasoner" @ "reasoner-for-discogs"

CONSTRUCT public var KNOWLEDGE
FROM in "reasoner-for-musicBrains" ( var KNOWLEDGE ) END

CONSTRUCT public var KNOWLEDGE
FROM in "reasoner-for-discogs" ( var KNOWLEDGE ) END

CONSTRUCT to "reasoner-for-musicBrains" ( var FACTS )
FROM in "Musicbrainz2MOFacts" ( var FACTS ) END

CONSTRUCT to "Musicbrainz2MOFacts" ( var METADATA )
FROM in "MusicBrainz"( metadata [[ var METADATA ]] ) END
...
CONSTRUCT discogs-document-for-crawler[ all HREF ]
FROM in document(iri="http://www.discogs.org") ( desc a [[ href [ var HREF ] ]] ) END
```

Finally, let us glance at the MO-Ontology-Reasoner module which is one of the modules that not only extracts data but is injected with data to operate on. Hence, one of the queries is adorned with the **public** keyword, indicating that chaining is to be performed against the rules of the importing module that pass input data to the reasoner. Those facts, together with the ontology definition (and any domain dependent reasoning we would like to perform on the music ontology data) are sent to an RDFS reasoner module, whose consequences are then made publicly available. This RDFS reasoner is an example of a highly reusable module that can be shared among many different modules. It implements the RDF semantic in the (graph-based) query language Xcerpt (cf. [5] for details).

```
MODULE "MO-Ontology-Reasoner"
IMPORT "RDFS-Reasoner"

CONSTRUCT public var KNOWLEDGE
FROM in "RDFS-Reasoner" ( var KNOWLEDGE ) END

CONSTRUCT to "RDFS-Reasoner" ( var FACTS )
FROM public var FACTS END

CONSTRUCT to "RDFS-Reasoner" ( var MO )
FROM in document(type="xmlrdf" iri="http://purl.org/ontology/mo/") ( var MO ) END
```

## 4 Modular Xcerpt—Requirements and Constructs

We have seen that modules can greatly ease the development of complex Web queries (as observed increasingly) and how to apply them in examples. Before we discuss the principles of the semantics in Section 5, let us first summarize the core concepts and constructs introduced. We divide the presentation of the concepts in two parts: from the perspective of the module programmer and of the module user.

**Module programmers** need constructs for defining sets of rules and ways of declaring appropriate access to the module—interfaces for proper encapsulation. To allow module authors to encapsulate modules, *visibility constructs* are employed. For each rule of the module, the construct term and the query term (if present) is associated with a visibility concept: *public* or *private*. Only public visibility is specifically specified, otherwise the default visibility *private* is used to encourage encapsulation.

**Module declaration:** We can group sets of rules into modules and give such a set an identifier. This module can than be imported into other modules or programs.
$\langle$*module*$\rangle$ ::= ‘MODULE’ $\langle$*module-id*$\rangle$ $\langle$*import*$\rangle$* $\langle$*rules*$\rangle$*

**Module interfaces:** We can declare allowed access points to a module to facilitate encapsulation and proper interfaces. Any construct term can be annotated with **public** to indicate that it can be queried by importing modules (see below).
$\langle$*interface-out*$\rangle$ ::= ‘public’ $\langle$*construct-term*$\rangle$
Conversely, importing modules may provision data to an imported module (see ‘module provision’ below). This data is exclusively queried by query terms marked with **public** in the imported module.
$\langle$*interface-in*$\rangle$ ::= ‘public’ $\langle$*query-term*$\rangle$

In other words, a module programmer defines the name and the in- and output interfaces of a module. The input of a module is accessed or queried by public query terms, the output of a module is formed by public construct terms. A module should also be complemented by documentation for the user describing its task and interfaces.

**Module users** need to be able to (a) declare which modules they want to use in a program, to (b) query the public interfaces of such modules, and to (c) provide data to such modules.

**Module importation:** We can import modules into other modules or programs. The only effect of a module is that the module identifier (or its alias, if an alias is used) becomes available for use in module querying or provision statements. In practice, module identifiers are often rather long and complex URIs which makes the use of (short and easy to read) aliases advisable in most cases.
⟨*import*⟩ ::= '`IMPORT`' ⟨*module-id*⟩ ('`@`' ⟨*alias-id*⟩)?

**Module querying:** We can query the consequences of the public construct terms of a module. The given query term is matched only against the results from *public* rules of the given module but neither against those from that module's *private* rules nor against other rules from the current module.
⟨*module-access*⟩ ::= '`in`' ⟨*module-id*⟩ '`(`' ⟨*query-term*⟩ '`)`'

**Module provision:** We can feed or provision data to the public query terms of a module. The result of a rule with such a construct term is only considered for *public* query terms in the given module, not for query terms in the current module or for query terms from the given module that are not marked *public*.
⟨*module-provision*⟩ ::= '`to`' ⟨*module-id*⟩ '`(`' ⟨*construct-term*⟩ '`)`'

With only these three operations, a module user can flexibly compose modules (even multiple instances of the same module) while all the encapsulation is taken care of by the module system without further user intervention.

So far, all module access is always explicitly scoped with the module identifier. In a language with views such as Xcerpt, this suffices as we always can add a bridging rule (such as the first rule in the MusicLibrary module from Section 3) that makes all data obtained from the public interface of an imported module available to other rules in the importing module (without need for qualification). We provide two additional variants of module import for convenience that cover this case. They only differ in the way they affect module cascading: '`import public`' ⟨*module-id*⟩ makes all data provided by the public interface of module *module-id* available to all unqualified rules in the importing module and also adds it to the public interface of that module whereas '`import private`' ⟨*module-id*⟩ only makes it available to the unqualified rules.

## 5 Reducing Xcerpt Modules—Stores

The dual objectives of our approach are to (a) keep the module system simple and easy to use and to (b) allow the reuse of existing language tools and engines without modification. These two objectives actually go hand in hand, as a reduction semantics for modules (i.e., a semantics that is based on the semantics of the module-free language) proves to be elegant and easy to understand and naturally fulfills the second objective.

To allow users to truly think in terms of modules and make use of this abstraction, it is important to ensure proper and valid module interactivity *statically* before applying the module-unaware query engine to the involved rules. Thus, only the intended rule dependencies must be present in the merged rules—we have no way of enforcing rule separations during rule execution.

For the Xcerpt module system we ensure proper rule dependencies using the notion of **stores**. Intuitively, a store is a designated data area where data and queries are appropriately redirected to adhere to the proper access of rules as specified by the module programmer. A store is associated with an identifier and consists of a *private*, *in* and *out* part. Intuitively, the *private* part is intended for data access internal to the module only and the *in* and *out* parts for input and output data of that module. That is, data to be processed by the module will be injected into the *in* part of the store and data constructed by the module—upon request from another module—will reside in the *out* part of the store and can be queried by an importing module.

Stores can already be simulated using the existing Xcerpt mechanisms. Let us first assume that for each module we have one associated store that is identified by the same (unique) identifier. The construct terms and query terms of each rule in an imported module as well as rules using **`in`** or **`to`** for module access or provision in an importing module are modified such that the appropriate store is referenced:

```
in <module-id> ( <query> )          ⟶   store [ id [ <module-id> ], access [ "out"], <query> ]
to <module-id> ( <construct> )      ⟶   store [ id [ <module-id> ], access [ "in"], <construct> ]
CONSTRUCT <c> FROM <q> END          ⟶   CONSTRUCT store [ id [<module-id>], access["private"], <c>]
                                        FROM store [ id [<module-id>], access["private"], <q> ] END
```

Some rules in the imported module are exempted from this transformation, viz. construct terms in goals (producing results for the end user), query terms specifically referencing an external resource (such as an XML document or other module) rather than the internal module store. Also, if the query term is a complex query it might be necessary to propagate the store specification inside the query (e.g., over disjunctions, negation, etc.). However, these details are omitted here for space reasons.[5]

### 5.1 Refining *Stores:* Instance Stores

The store concept described above ensures basic encapsulation capabilities for Xcerpt modules and is attractive for its simplicity. However, there are certain situations where associating one store per module is not sufficient. Consider the situation where two modules (`A,B`) imports a third one (`C`) and both `A` and `B` injects data into the store associated with `C`. In such a case, after module `C` has processed the data, module `A` *may* receive data initially injected by module `B`. As such, modules `A` and `B` are not kept separate violating one of the core premises of our desire for modules. This is not a limit of the store approach, but due to the assumption of the existence of one store per module.

To address this problem, we associate stores not with a module but with a module *import*. This can be seen as instantiating a store for each module import with the identifier of the importing module. We thus end up with two stores C<A> and C<B>, due to two import operators. A similar case where this is needed is when we use the same module

[5] But available with examples at `http://www.reuseware.org/modularxcerptexample`.

but with different "feeds" using aliases. This is the case in the Music Library module presented in Section 3 where aliases (using `@`) were used to force such separations.

**Implementation.** Not only is it an advantage to reuse the query engine in executing the transformed and merged rules, it is also beneficial if existing technology can be used to realize the above-described transformations. To achieve this, we realize the module system via composition in the Reuseware Composition Framework [8]. The composition framework allows for the development of a light-weight composition system responsible for handling the augmented constructs related to modules. The composition framework allows both to extend the Xcerpt language with the additional syntactic constructs and to handle the transformation and merging of the involved rules in the manner described above to enforce encapsulation. The details of this implementation are left out for space reasons, but are available at `http://www.reuseware.org/modularxcerptexample`.

## 6 Related work

Practical Web *query* languages need to provide support for some form of reuse and modules as evidenced by (though somewhat limited) module support in languages such as XSLT and XQuery. *Rule* languages for the Web, on the other hand, show an apparent lack of module support, despite considerable research on module extensions for classical logic programming. One of the reasons that modules are still not in the "standard repertoire" of rule languages may be the complexity of many previous approaches.

Representative and, arguably, the most comprehensive treatment of modules in logic programming is presented in [4]. It is far more expressive than our approach but at the price of a complex semantics and several operations with, in our opinion, little practical use (such as module intersection or renaming). We believe that a single well-designed union operation with clear interfaces together with a strong reliance on views as a core feature of rule languages is not only easier to grasp but also easier to realize.

Though many rule languages for the Web fail to provide modules, this is not true for the two preeminent Web query languages, XSLT and XQuery. XSLT [6] can be considered a rule language, however using precedence rather than union semantics for multiple applicable rules. Rule precedence is also the dominating issue for XSLT's module system which provides intricate mechanisms for determining the precedence of rules from different modules. Nevertheless, the resulting module system is considerably less powerful (no scoped import, limited parameterization: apply-imports) yet needs a more complex semantics than module-free XSLT, quite in contrast to our approach.

It is worth mentioning that XQuery [3] also provides a module system, however without parameterization, but as a function programming language requires explicit flow control in all cases. Thus, issues such as private or public import (or the difference between import and include in XSLT) do not apply for XQuery. SPARQL [9], finally, the recently proposed RDF query language, has no concept of user defined program units (such as rules, functions, procedures, etc.) and thus no use for a module concept in the sense of our approach. However, rule-based extensions for SPARQL (in the spirit of Datalog) could certainly profit from the module system illustrated here using Xcerpt.

## 7 Conclusions and Outlook

We argue that one ingredient to cope with size and diversity of information on the (Semantic) Web is *modular* query authoring and execution. We show advantages along a concrete use case dealing with music information aggregation on the Web. Furthermore, we demonstrate how it is possible to augment existing query languages—here focused on the language Xcerpt—with new constructs while reusing already developed semantics and query engines thanks to a reduction semantics approach. The proposed module system is simple to use (in contrast to many approaches from logic programming) yet provides better encapsulation and more advanced features (such as scoping and paramterization) than module systems for XSLT or XQuery.

The proposed module system has been formalized [1] and implemented using the Reuseware Composition Framework. Integration with upcoming revisions of Xcerpt is planned. Furthermore, we would like to exploit existing techniques and tools such as Xcerpt's type system [12] for improving module composition. We are also investigating how similar techniques can be applied to add or improve module systems for other (non-rule based) query languages (for example, the module system of XSLT).

**Acknowledgement.** This research has been co-funded by the European Commission and by the Swiss Federal Office for Education and Science within the 6th Framework Programme project REWERSE number 506779 (cf. `http://rewerse.net`).

## References

1. U. Aßmann, S. Berger, F. Bry, T. Furche, J. Henriksson, and P.-L. Patranjan. A generic module system for web rule languages: Divide and rule. In *Proc. Int'l. RuleML Symp. on Rule Interchange and Applications*, 2007.
2. J. Bailey, F. Bry, T. Furche, and S. Schaffert. Web and Semantic Web Query Languages: A Survey. In *Tutorial Lect. Int'l. Summer School 'Reasoning Web'*, LNCS 3564. 2005.
3. S. Boag, D. Chamberlin, M. F. Fernández, D. Florescu, J. Robie, and J. Siméon. XQuery 1.0: An XML Query Language. Working draft, W3C, 2005.
4. A. Brogi, P. Mancarella, D. Pedreschi, and F. Turini. Modular logic programming. *ACM Trans. Program. Lang. Syst.*, 16(4):1361–1398, 1994.
5. F. Bry-Haußer, T. Furche, and B. Linse. Data Model and Query Constructs for Versatile Web Query Languages: State-of-the-Art and Challenges for Xcerpt. In *Proc. Int'l. Workshop on Principles and Practice of Semantic Web Reasoning (PPSWR)*, pages 90–104, 2006.
6. J. Clark. XSL Transformations, Version 1.0. Recommendation, W3C, 1999.
7. F. Giasson and Y. Raimond. Music ontology specification. Specification, Zitgist LLC, 2007.
8. J. Henriksson, J. Johannes, S. Zschaler, and U. Aßmann. Reuseware – adding modularity to your language of choice. *Proc. of TOOLS EUROPE 2007: J. of Object Technology*, 2007.
9. E. Prud'hommeaux and A. Seaborne. SPARQL Query Language for RDF. Candidate recommendation, W3C, 2007.
10. S. Schaffert. *Xcerpt: A Rule-Based Query and Transformation Language for the Web*. Dissertation/Ph.D. thesis, University of Munich, 2004.
11. S. Schaffert and F. Bry. Querying the Web Reconsidered: A Practical Introduction to Xcerpt. In *Proc. Extreme Markup Languages (Int'l. Conf. on Markup Theory & Practice)*, 2004.
12. A. Wilk and W. Drabent. A Prototype of a Descriptive Type System for Xcerpt. In *Proc. of Workshop on Principles & Practice of Sem. Web Reasoning (PPSWR)*, LNCS 4187. 2006.