I3-D14

Composition Framework and Typing Technology tutorial

Project title:	Reasoning on the Web with Rules and Semantics
Project acronym:	REWERSE
Project number:	IST-2004-506779
Project instrument:	EU FP6 Network of Excellence (NoE)
Project thematic priority:	Priority 2: Information Society Technologies (IST)
Document type:	D (deliverable)
Nature of document:	R (report)
Dissemination level:	PU (public)
Document number:	IST506779/Dresden/I3-D14/D/PU/b1
Responsible editors:	Jakob Henriksson
Reviewers:	Mikael Berndtsson
Contributing participants:	Dresden, Linköping, Warsaw
Contributing workpackages:	I3
Contractual date of deliverable:	29 February 2008
Actual submission date:	29 February 2008

Abstract
This deliverable provides a tutorial on how to use the technology and the demonstrators developed within the I3 working group. In particular the deliverable includes a tutorial on how to use the Reuseware Composition Framework and the Xcerpt typing system XcerptT.

Keyword List
software composition, component based development, semantic web, query languages, Xcerpt, Reuseware, typing

Project co-funded by the European Commission and the Swiss Federal Office for Education and Science within the Sixth Framework Programme.

© *REWERSE 2008.*

Composition Framework and Typing Technology tutorial

Uwe Aßmann[1] and Andreas Bartho[1] and Wlodek Drabent[2,3] and Jakob Henriksson[1] and Artur Wilk[3]

[1] Fakultät Informatik, Technische Universität Dresden
Email: {uwe.assmann|andreas.bartho|jakob.henriksson}@tu-dresden.de

[2] Institute of Computer Science, Polish Academy of Sciences, Warsaw
Email: drabent@ipipan.waw.pl

[3] Linköpings universitet, Department of Computer and Information Science, Linköping
Email: artwi@ida.liu.se

29 February 2008

Abstract
This deliverable provides a tutorial on how to use the technology and the demonstrators developed within the I3 working group. In particular the deliverable includes a tutorial on how to use the Reuseware Composition Framework and the Xcerpt typing system XcerptT.

Keyword List
software composition, component based development, semantic web, query languages, Xcerpt, Reuseware, typing

Contents

Part I
Overview

During the span of the European Network of Excellence REWERSE the *Composition and Typing* working group I3 has developed both composition and typing technology for the rule-based languages used on the Web and the Semantic Web. Focus has in particular been on the semi-structured data query language Xcerpt [3, 5]. This deliverable provides tutorials, use-cases and examples of these technologies to show how they can be beneficial to end-users.

Part II gives examples on how the composition technology can advantageously be used by end-users in order to construct modular query programs and ontologies. Part III then gives use-cases and examples on how typing technology can be deployed to help query programmers detect errors in their query programs.

Part II
Composition Framework

1 Introduction

Composition technology has been made available that is able to provide languages, such as Xcerpt, with component-oriented language extensions. This technology has been developed in the composition framework Reuseware.[1] The general goal is to provide intuitive reusable entities to end-users, that is, to every-day programmers and users of the addressed languages. For the case of Xcerpt, equip Xcerpt programmers with the possibility to define reusable query entities. For Xcerpt in particular, the module concept was developed, enabling the separation of query programs into reusable and better understood parts (modules). For some published papers in this directions see [1, 2, 6, 7]. The composition framework is static in the sense that composition is done at compile-time, and not during run-time. Modules for Xcerpt are, for example, composed statically, enabling the Xcerpt interpreter to be reused as a black-box when executing modular Xcerpt programs.

Not only rule-based query languages like Xcerpt have been addressed, but also ontology languages such as OWL [9]. For ontology languages the concept of *role models* is provided as reusable entities.

Section 2 contains three composition applications with accompanying examples.

2 Composition Applications

First, in Section 2.1, the notion of modules for Datalog is discussed and exemplified. Then, in Section 2.2, the module concept is transferred to Xcerpt. Finally, in Section 2.3, role models for ontology languages are discussed. It should be noted and highlighted that all these applications have been developed and realized using the composition framework Reuseware.

[1] `http://www.reuseware.org`

2.1 Modular Datalog

Datalog programs consist of a number of predicates, so-called facts and rules, all written in one single file. If the number of predicates grows too big, the resulting program becomes large and unhandy to maintain. In order to split up the program into manageable parts, a module system would be useful. Having added that module concept to Datalog, the Modular Datalog language was born.

2.1.1 The original program

The example Datalog program is a simulation of a small company. The company consists of three departments, sale, marketing and research, each of which has employees and a manager. There are rules with which to decide who of the employees and managers will get a bonus for good work, and who will be get a serious talk under four eyes with the boss because of bad job performance. The performance of both the employees and the departments are rated. Managers' performance equals the performance of their respective departments. In the following, the program's fragments are presented in detail, along with some comments.

First of all, there is a fact base of employees and managers.

```
manages(steward,sales).
worksat(george,sales).
worksat(lisa,sales).

manages(james,research).
worksat(steve,research).
worksat(marco,research).

manages(helen,marketing).
worksat(john,marketing).
worksat(sarah,marketing).

worksat(E,D) :- manages(E,D).
```

The predicate `worksat` assigns an employee to a department. The predicate `manages` assigns a manager. The last rule says that a department's manager works at this very department.

In order to decide whom to give a bonus, the performance of both the departments and the individual employees must be rated.

```
performance(george, extremelygood).
performance(lisa,bad).
departmentperformance(sales,good).

performance(steve,average).
performance(marco,bad).
departmentperformance(research,bad).

performance(john,average).
performance(sarah,good).
departmentperformance(marketing,average).

performance(E,P) :- manages(E,D),departmentperformance(D,P).
```

The predicate `performance` assigns a performance rating to an employee. The predicate `departmentperformance` assigns a performance rating to a whole department. The last rule says that the performance of a manager is defined by the performance of his department. The

performance ratings can be one of the following: `extremelygood`, `good`, `average` or `bad`. As they are not comparable by default, an order is defined.

```
performancebetter(average,bad).
performancebetter(good,average).
performancebetter(extremelygood,good).
performancebetter(X,Z) :- performancebetter(X,Y),performancebetter(Y,Z).
performanceworse(X,Z) :- performancebetter(Z,X).
```

The first three facts establish the actual order: average is better than bad, good is better than average, extremelygood is better than good. The following rule computes the transitive closure of the performance rating, for instance that good is better than bad. The last rule is just a helper rule that says rating X is worse than rating Z if rating Z is better than rating X.

Now that the data is set up, the "query rules" can be introduced. First of all we would like to know who gets paid. Paying all employees is usually a good idea. This leads to the following rule.

```
payroll(E) :- worksat(E,_).
```

Everyone who works at some department (including managers) is considered employee and therefore paid.

Next we would like to know who performed good enough to get a bonus. In order to qualify for a bonus, the job performance must be above average.

```
bonuslimitabove(average).
```

The exact rule is as follows:

- The employee must have a performance above the limit (i.e. more than average)
- The employee must work in a department whose performance is above the limit (also more than average)

This results in the following Datalog rule.

```
bonus(E,D) :- bonuslimitabove(L),
              worksat(E,D),
              departmentperformance(D,DP),
              performancebetter(DP,L),
              performance(E,EP),
              performancebetter(EP,L).
```

Finding out who did bad is completely analogous.

- The employee must have a performance below the limit (i.e. less than average)
- The employee must work in a department whose performance is below the limit (also less than average)

```
foureyestalklimitbelow(average).
foureyestalk(E,D) :- foureyestalklimitbelow(L),
                     worksat(E,D),
                     departmentperformance(D,DP),
                     performanceworse(DP,L),
                     performance(E,EP),
                     performanceworse(EP,L).
```

2.1.2 The modular program

Once a Datalog program grows the code becomes unhandy. Therefore a module system has been added, which allows to separate unrelated facts and rules from each other and store them in different files. Each module is stored in its own file. Modules can be imported by the main program and by other modules. Additionally, there are different visibilities for predicates: `public` and `private`. While private predicates can only be used from within a module, public ones can be used by the importing module.

In the example program, the facts have been moved to modules. There is one module for each department containing the employees, managers and performances for this department. The module `department` merges these individual modules. Finally, there is a module `performance` which contains the order of the performance ratings. The modules and their usage will now be shown in more detail.

The modules for the individual departments have all the same structure. The sales department module shall serve as an example here.

```
1 MODULE sales

3 @ manages(steward,sales).
  @ worksat(george,sales).
5 @ worksat(lisa,sales).

7 @ performance(george, extremelygood).
  @ performance(lisa,bad).
9
  @ departmentperformance(sales,good).
```

A module starts with the keyword `MODULE`, followed by the module name. The name has no special meaning and is used to differentiate rules from different modules. For a better understandability one should choose a descriptive name that indicates the module's purpose. Below the module declaration are the individual facts. Predicates that have an `@`-sign in front of them are considered public. They can be accessed from other modules. The sales department module from above consists exclusively of public facts.

All the modules from the individual departments must somehow be aggregated to access their data. In order to keep the main program tidy and not clutter it with aggregation code, an additional module, `department` has been introduced.

```
  MODULE department
2
  IMPORT /salesdepartment.mdatalog AS sales
4 IMPORT /marketingdepartment.mdatalog AS marketing
  IMPORT /researchdepartment.mdatalog AS research
6
  @worksat(E,D) :- worksat(E,D).
8 @manages(E,D) :- manages(E,D).
  @performance(E,P) :- performance(E,P).
10 @departmentperformance(D,P) :- departmentperformance(D,P).


12
  worksat(E,D) :- in sales (worksat(E,D)).
14 worksat(E,D) :- in marketing (worksat(E,D)).
  worksat(E,D) :- in research (worksat(E,D)).
16
  manages(E,D) :- in sales (manages(E,D)).
```

```
manages(E,D) :- in marketing (manages(E,D)).
manages(E,D) :- in research (manages(E,D)).

performance(E,P) :- in sales (performance(E,P)).
performance(E,P) :- in marketing (performance(E,P)).
performance(E,P) :- in research (performance(E,P)).

departmentperformance(D,P) :- in sales (departmentperformance(D,P)).
departmentperformance(D,P) :- in marketing (departmentperformance(D,P)).
departmentperformance(D,P) :- in research (departmentperformance(D,P)).
```

The first line is the already known module declaration. The following lines contain import statements for modules. The path after the `IMPORT` keyword indicates a module's location. The name after the `AS` keyword is the name with which the imported module will be referenced from within the importing module.

Following the import declarations there are four public rules, followed by a number of private rules. The public rules do only forward the data from the private rules. The private rules in turn get their data from the individual department modules that have been described above. As one can see, modules' public rules are accessed via `in <modulename> ( <predicate> ).`

The `department` module has been written the way it is to show how public and private predicates can be used in parallel within a module. Even identical names do not conflict. However, as no data of the module should be hidden from the calling module, one could easily rewrite the module by making all rules public. The first four rules, that forward the private rules' data, could then be removed.

```
MODULE department

IMPORT /salesdepartment.mdatalog AS sales
IMPORT /marketingdepartment.mdatalog AS marketing
IMPORT /researchdepartment.mdatalog AS research

@worksat(E,D) :- in sales (worksat(E,D)).
@worksat(E,D) :- in marketing (worksat(E,D)).
@worksat(E,D) :- in research (worksat(E,D)).

@manages(E,D) :- in sales (manages(E,D)).
@manages(E,D) :- in marketing (manages(E,D)).
@manages(E,D) :- in research (manages(E,D)).

@performance(E,P) :- in sales (performance(E,P)).
@performance(E,P) :- in marketing (performance(E,P)).
@performance(E,P) :- in research (performance(E,P)).

@departmentperformance(D,P) :- in sales (departmentperformance(D,P)).
@departmentperformance(D,P) :- in marketing (departmentperformance(D,P)).
@departmentperformance(D,P) :- in research (departmentperformance(D,P)).
```

The last module is the `performance` module in which the order of the performance ratings is specified. It follows the same principles that have already been described.

```
MODULE performance

@ performancebetter(X,Y) :- performancebetter(X,Y).

performancebetter(average,bad).
performancebetter(good,average).
```

```
performancebetter(extremelygood,good).
performancebetter(X,Z) :- performancebetter(X,Y),performancebetter(Y,Z).
```

The first rule exposes the information from the private predicates to other modules. Again, this module could be rewritten by removing the first rule and making all predicates public.

The main program essentially imports the modules discussed above and defines the "query rules" `payroll`, `bonus` and `foureyestalk`, along with some helper rules.

```
IMPORT /department.mdatalog AS department
IMPORT /performance.mdatalog AS performance

worksat(E,D) :- in department(worksat(E,D)).
worksat(E,D) :- in department(manages(E,D)).
manages(E,D) :- in department(manages(E,D)).

performance(E,P) :- in department(performance(E,P)).
performance(E,P) :- manages(E,D),departmentperformance(D,P).

departmentperformance(D,P) :- in department(departmentperformance(D,P)).

performancebetter(P1,P2) :- in performance(performancebetter(P1,P2)).
performanceworse(P1,P2) :- in performance(performancebetter(P2,P1)).


payroll(E) :- worksat(E,_).


bonuslimitabove(average).
foureyestalklimitbelow(average).

bonus(E,D) :- bonuslimitabove(L),
              worksat(E,D),
              departmentperformance(D,DP),
              performancebetter(DP,L),
              performance(E,EP),
              performancebetter(EP,L).


foureyestalk(E,D) :- foureyestalklimitbelow(L),
                     worksat(E,D),
                     departmentperformance(D,DP),
                     performanceworse(DP,L),
                     performance(E,EP),
                     performanceworse(EP,L).
```

2.1.3 Summary

Modules have been used in Datalog to divide a large set of predicates into manageable modules. Predicates within modules can either be public or private, i.e. the developer can decide which data to expose to the public and which to hide. Additionally, it has been shown that modules can be nested.

2.2 Modular Xcerpt

2.2.1 Motivation

Xcerpt is a powerful new query language for XML documents. It consists of a number of rules that specify patterns of nodes to be matched along with rewrite actions. As Xcerpt lacks the concept of modules, all collaborating rules have to be written in one single file. This makes larger programs hard to extend and maintain. With the help of Reuseware module support has been added to Xcerpt. The resulting version of Xcerpt is called Modular Xcerpt and shall be introduced in the following. This is not an introduction to the original Xcerpt language, basic knowledge is assumed.

2.2.2 Syntax

This section briefly explains the syntax used for Modular Xcerpt. Modular Xcerpt extends the syntax of Xcerpt. For an overview of the syntax of Xcerpt, please see, e.g., [5].

- **Module definition.** An Xcerpt module is associated with a name, it optionally imports other modules and contains a set of Xcerpt rules realizing the module. Furthermore, a module can define its interfaces that will be used by other modules or programs.
  - *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.
    ⟨*module*⟩ ::= '`MODULE`' ⟨*module-id*⟩ ⟨*import*⟩* ⟨*rules*⟩*
  - *Module interfaces.* We can declare allowed access points to a module to facilitate encapsulation and proper interfaces. Any construct term can be annotated with the `public` keyword to indicate that it can be queried by importing modules (see below).
    ⟨*interface-out*⟩ ::= '`public`' ⟨*construct-term*⟩
    Conversely, importing modules may provide data to an imported module. This data is used exclusively by `public` queries of the imported module.
    ⟨*interface-in*⟩ ::= '`public`' ⟨*query*⟩
- **Module deployment.** Not only is it possible to define modules, but they can be imported into other modules or programs. This part explains the constructs provided for this purpose.
  - *Module import.* We can import modules into other modules or programs. The only effect of a module import is that the associated module alias becomes available for use in module querying or provision statements.
    ⟨*import*⟩ ::= '`IMPORT`' ⟨*module-id*⟩ '`AS`' ⟨*alias-id*⟩
  - *Module querying.* We can query the consequences of the public construct terms of a module. The given query is matched only against the results from *public* rules of the given module but not against other rules from the current module.
    ⟨*module-access*⟩ ::= '`in`' ⟨*module-id*⟩ '`(`' ⟨*query*⟩ '`)`'
  - *Module provision.* We can feed or provide data to the public query terms of a module. The result of a rule with such a construct term is only sent to *public* queries in the given module. Private queries and queries of the current module are not executed.
    ⟨*module-provision*⟩ ::= '`to`' ⟨*module-id*⟩ '`(`' ⟨*construct-term*⟩ '`)`'

2.2.3 Examples

In the following we will explain how modules are used and what advantages they have. Our example shall be a mashup program that collects data of Audio CDs from various websites and displays it uniformly in a nicely formatted document. Please note that the original webpages are not used in this tutorial due to their huge size and hard-to-understand structure. Instead, strongly stripped-down copies are used. In this example the terms "tag" and "node" for the elements of an XML or Xcerpt tree are used interchangeably. "Tag" is used when referring to a textual representation whereas "node" is used when regarding the tree structure.

Starting simple - Collecting CD data from Amazon The first version of our Mashup program does not need to do much. It should be able to extract CD data from Amazon web pages and create an HTML file to display all titles and artists in a table.

Our program's "internal data" should be a list of all CDs, including title, artist, URL of the cover image and songs. Using pseudo-Xcerpt code, the structure looks like this:

```
cd [
    artist,
    title,
    coverlink,
    songs [
        song, song ... song
    ]
]
```

We decided to include the cover image and the songs, even though we do not need them yet. It is not hard and saves work when we later want to add them to the output.

Our program needs a goal rule, in which the data to be output is specified. As already said, we would like to have a table with all CD titles and artists. The output should be an HTML document called mashup.html. The goal rule looks as follows:

```
GOAL
    out {
        resource {"file:mashup.html", "xml"},
        html [
            head [
                title [ "Mashup" ]
            ],
            body [
                table [
                    all tr [
                        td [ var ARTIST ],
                        td [ var TITLE ]
                    ]
                ]
            ]
        ]
    }
FROM
    cd [[
        artist [var ARTIST],
        title [var TITLE]
    ]]
END
```

In the `FROM` block the rule matches all `[cd]` nodes with both an artist and a title. The contents of the artist and the title nodes are written to the HTML table constructed in the `GOAL` block. This rule alone does nothing, however. We have to provide the data that can be matched, i.e. one or more `cd` nodes with an `artist` and a `title` child node.

To do so, we have to convert the existing data from the Amazon format to our internal data format with another rule. Figure 1 shows what the page looks like (or rather the stripped-down copies).

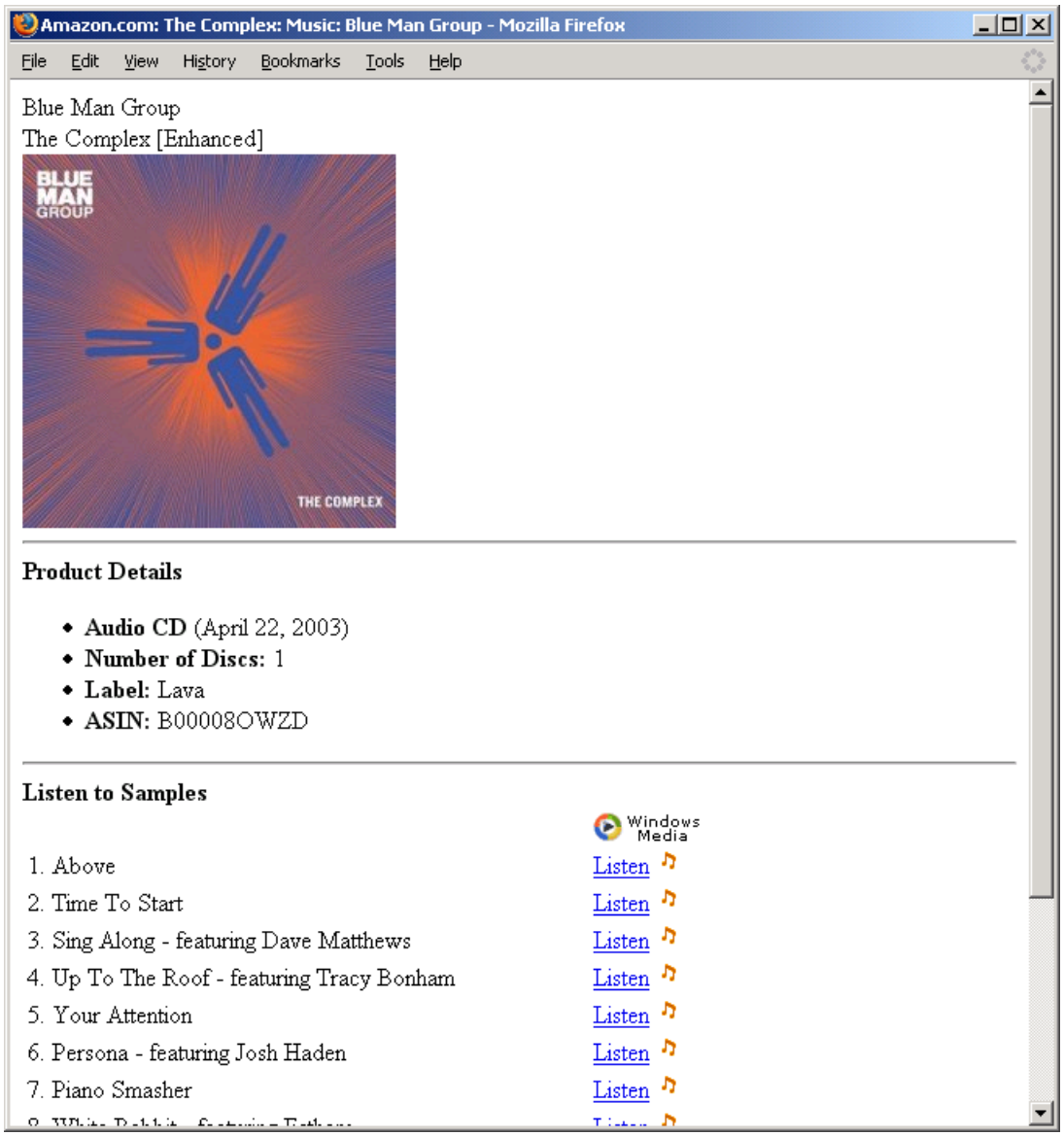

Figure 1: An input document for the mashup program

The following listing displays the HTML structure of the Amazon web pages.

```
<html>
    <head>
        <title>Amazon.com: TITLE: Music: ARTIST</title>
    </head>
    <body>
        ATRIST<br/>
        TITLE<br/>
        <img src=COVERLINK ... />
        <hr/>
        <b>Product Details</b>
        <br />
        <ul>
            <li><b>Audio CD</b> ...</li>
            <li><b>Number of Discs:</b> 1</li>
            <li><b>Label:</b> ...</li>
            <li><b>ASIN:</b> ...</li>
        </ul>
        <hr/>
        <b>Listen to Samples</b>
        <table ... >
            <tr>
                <th ... > ... </th>
                <th ... > ... </th>
            </tr>
            <tr>
                <td>SONGTITLE 1</td>
                <td>
                    <a ... >Listen</a>
                    <img .../> <!-- listen icon -->
                </td>
            </tr>

            ...

            <tr>
                <td>SONGTITLE x</td>
                <td>
                    <a ... >Listen</a>
                    <img .../> <!-- listen icon -->
                </td>
            </tr>
        </table>
    </body>
</html>
```

The contents of the Amazon web pages must be converted into our internal data format. The subsequent rule does this for the copy of “Blue Man Group - The Complex”. Note how the `FROM` clause mirrors the HTML structure of the input document.

```
CONSTRUCT
    cd [
        artist[var ARTIST],
        title[var TITLE],
        coverlink[var COVERLINK],
        songs[
            all song [var SONGTITLE]
        ]
```

```
    ]
FROM
    in { resource { "file:data/amazon-blue_man_group-the_complex.html", "xml" },
        html [
            head [[]],
            body [[
                var ARTIST, br,
                var TITLE, br,
                img {
                    attributes {
                        src{var COVERLINK}
                    }
                },
                table [[
                    tr[
                        th [[]]
                    ],
                    tr[
                        td [var SONGTITLE],
                        td [[]]
                    ]
                ]]
            ]]
        ]
    }
END
```

Running the program now results in an HTML document containing a table with only one row.

```
<html>
    <head>
        <title>Mashup</title>
    </head>
    <body>
        <table>
            <tr>
                <td>Blue Man Group</td>
                <td>The Complex [Enhanced]</td>
            </tr>
        </table>
    </body>
</html>
```

For all CDs to appear in the mashup the rule must be repeated for all Amazon web pages. The only difference is the resource location. To add data from different websites, similar rules must be written, specifically tailored to the web pages' structure. Again, if there is one CD per web page, we have to provide one rule for each CD.

Figure 2 shows the final output, involving four CDs from two different websites: Amazon and Buch.de.

Figure 3 shows the overall structure of the program. Each box corresponds to a rule. The data being passed between the rules is annotated at the arrows.

As one can see, this naive approach has several disadvantages. The file becomes very huge very fast and the program is hardly maintainable due to the enormous amount of code doubling. Therefore we would like to split up the program into modules. Code should be reused when possible.

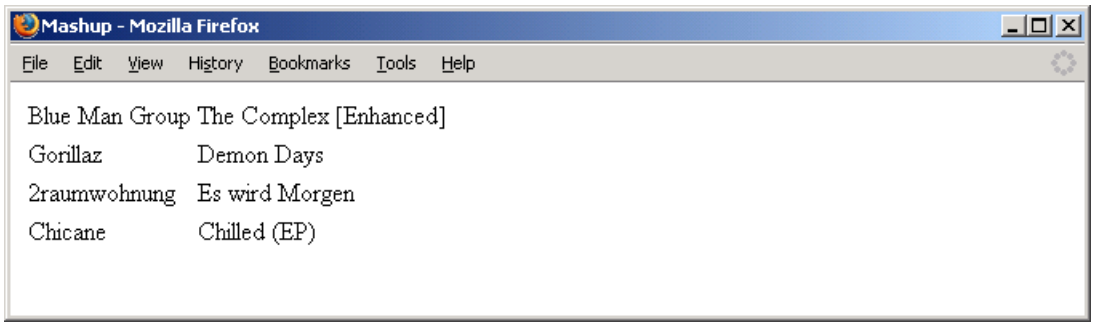

Figure 2: Final output of the simple mashup program

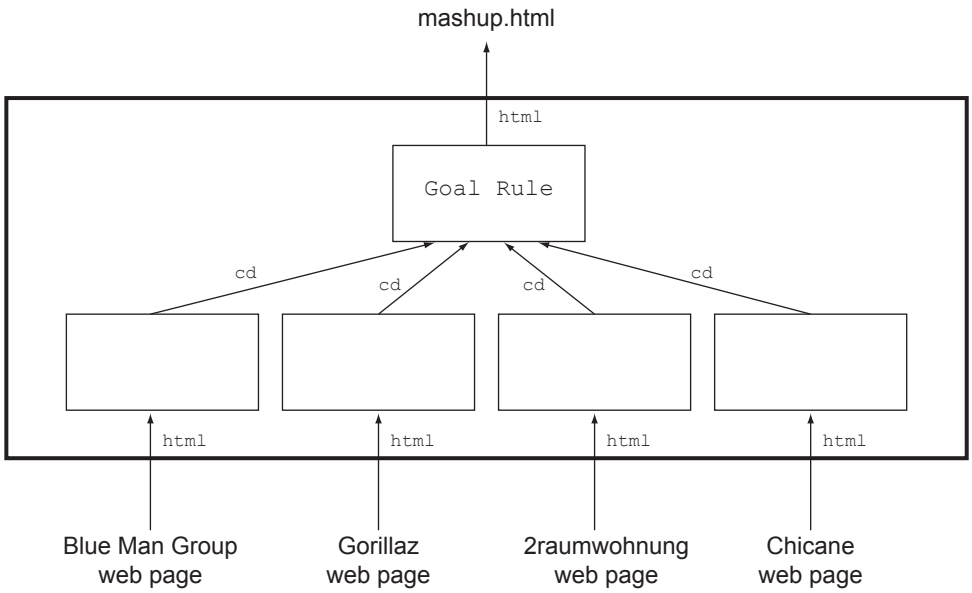

Figure 3: Rule chaining of the simple mashup program

Introducing data import modules The code that can be reused are the transformation rules that convert the web pages' data into our internal data format. Therefore we need two modules, one for the Amazon importer and one for the Buch.de importer.

Figure 4 shows the structure of the mashup program with data import modules present.

A module is stored in its own file and looks very much like the main program. The most notable difference is that it starts with a module declaration.

```
1 MODULE AmazonImport
  ...
```

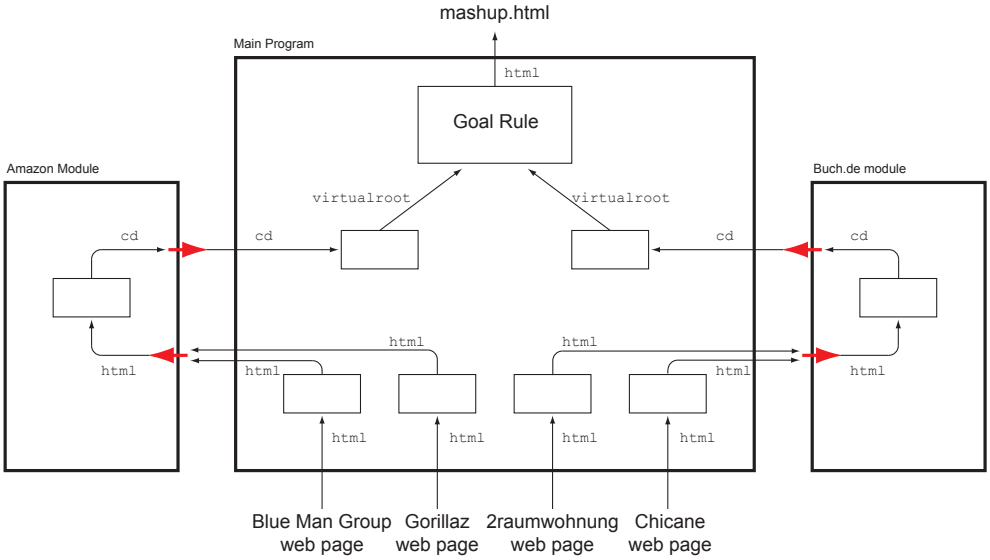

Figure 4: Rule chaining of the mashup program with data import modules

The name after the `MODULE` keyword is entirely arbitrary. It does not need to reflect the actual filename and it is not used anywhere. For comprehensibility one should choose a describing name, though. After the module declaration the actual transformation rules follow. Below is the complete content of the Amazon module file.

```
MODULE AmazonImport

CONSTRUCT
    public cd [
        artist[var ARTIST],
        title[var TITLE],
        coverlink[var COVERLINK],
        songs[
            all song [var SONGTITLE]
        ]
    ]
FROM
    public html [
        head [[]],
        body [[
            var ARTIST, br,
            var TITLE, br,
            img {
                attributes {
                    src{var COVERLINK}
                }
            },
            table [[
                tr[
                    th [[]]
                ],
```

```
            tr[
28              td [var SONGTITLE],
                td [[]]
30          ]
        ]]
32    ]]
  ]
34 END
```

This rule is mostly identical to the rules from the main program that convert the Amazon data. The differences are that there is no resource statement — we do not want to tie the importer to a specific resource — and the addition of the `public` keyword to both the `CONSTRUCT` and the `FROM` clause.

Adding `public` to the `CONSTRUCT` clause makes the rule public, i.e. available for use from outside the module. When no `public` modifier is added, the rule is considered private and can only be used by other rules within the same module. For a module to be useful at least one rule must be public.

Adding `public` to the `FROM` clause makes the rule available for data injection from outside the module. Without the `public` modifier the rule can only match data from the same module or the module's submodules.

Conceptually speaking, a module is a component with inputs and outputs, i.e. an interface. All public `FROM` clauses are inputs, all public `CONSTRUCT` clauses are outputs. The behaviour of the module is hidden and unknown to the calling modules. The module inputs and outputs are reflected by the red arrows in figure 4.

Modular Xcerpt does, of course, also offer facilities to actually use the declared modules. First of all, the modules to be used must be imported. We want to use the modules from the main program, therefore we add the following import statements at the beginning of the file.

```
IMPORT /import/Amazon.mxcerpt AS Amazon
2 IMPORT /import/Buchde.mxcerpt AS BuchDE
```

The path after the `IMPORT` keyword is the location to the file in which the module is defined. The name after the `AS` keyword defines the alias with which the module will be accessed. This name does not have to be the same as the actual name of the module.

Having the imports declared, we can shorten the rules that access the webpage data:

```
CONSTRUCT to Amazon (
2    var DATA
)
4 FROM
    in {
6       resource { "file:data/amazon-blue_man_group-the_complex.html", "xml" },
        var DATA
8   }
END
```

The `CONSTRUCT to <ModuleName> ( ... )` is Modular Xcerpt's way to connect to the input of a module. The above rule takes the input resource's complete data and passes it to the Amazon module without any change. There is one such rule for each webpage to be processed. The Buch.de pages are sent to the BuchDE module, of course.

Now that the data can be fed into the modules, we need a way to get it out of the modules again. This is done by another short rule, which is connected to the Amazon import module's output.

```
CONSTRUCT
    virtualroot[all var CDINFO]
FROM in Amazon (
    var CDINFO →cd[[]]
)
END
```

This rule retrieves all datasets that the Amazon module computed. Note that in contrast to the original, non-modular program, we have to introduce an artificial root node, `virtualroot`. Construct clauses with nothing than a variable usage are only allowed when constructing to a module. This is not the case here. For the BuchDE module there is an analogous rule.

The complete data can now uniformly be accessed via the virtualroot tag. The original `GOAL` rule that writes the HTML document must be slightly altered. Instead of only matching the `cd` nodes, it does now have to match `virtualroot` with the `cd` nodes as children embedded.

```
GOAL
    out {
        resource {"file:mashup.html", "xml"},
        html [
            head [
                title [ "Mashup" ]
            ],
            body [
                table [
                    all tr [
                        td [ var ARTIST ],
                        td [ var TITLE ]

                    ]
                ]
            ]
        ]
    }
FROM
    virtualroot [[
        cd [[
            artist [var ARTIST],
            title [var TITLE]
        ]]
    ]]
END
```

It is important to note that rules which send data to a module and rules which retrieve data from the module must both be defined in the same module or the main program, i.e. in the same file. It is, for example, not possible to query in the main program a rule from module X, which is fed by a rule from module Y.

Summary With the introduction of modules we were able to reuse code and split the Xcerpt program up into managable parts. A module is a component that accepts some input and computes output, which can be accessed by the calling module.

Introducing data export modules As time progresses, the simple HTML output is not sufficient anymore. Something more sophisticated is needed. The new output shall be in the

HTML format again. Additionally to the CD title and the artist it should display the album cover and a list of all songs.

As we expect the need for more output formats in the future, we will use modules to be more flexible.

The output will be created by multiple cooperating modules. The HtmlCreator module creates the final HTML document, i.e. the `html` tags and the `head`, including `title` and (CSS) `style`. The actual html `body` is also added to the final document. It is not computed by the HtmlCreator module, however.

The actual content of the body is produced by the TableLayout modules, either MobileTableLayout or ElaborateTableLayout.

- The simple layout that has been used so far is adequate to display data on mobile devices. It will be computed by the MobileTableLayout module.

- A more advanced layout is reasonable for bigger screens and faster internet connections. It will be computed by the ElaborateTableLayout module.

As HTML documents can easily be customized using CSS, we also want to have flexibility there. The StyleSheet module offers easy exchangeability for CSS stylesheets.

Figure 5 shows how the modules interact. The import modules are left out due to space restrictions.

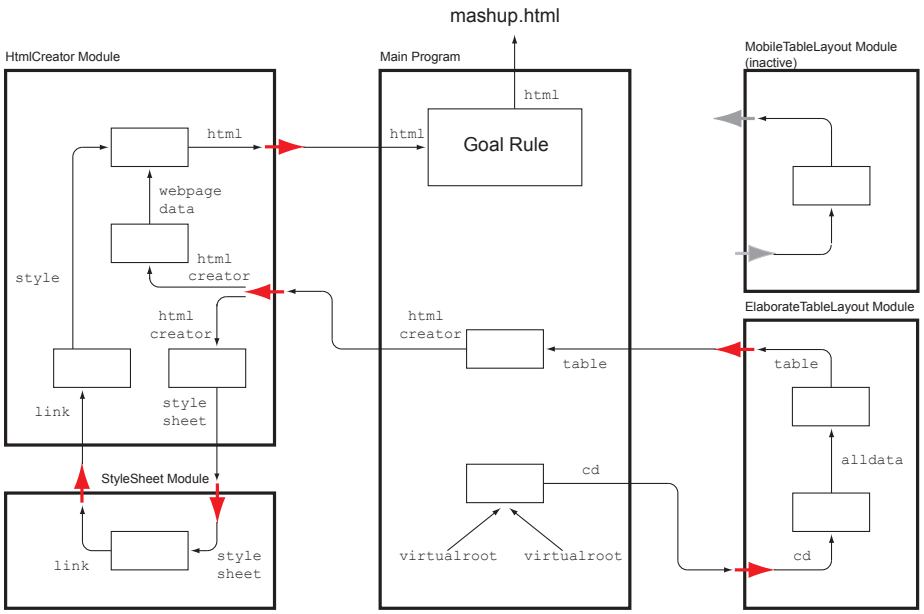

Figure 5: Rule chaining of the mashup program with data export modules

In the following, each of the modules is described in more detail.

The Main program With the module approach the GOAL rule is drastically simplified.

```
GOAL
    out {
        resource {"file:mashup.html", "xml"},
        var DATA
    }
FROM
    in HtmlCreator (
        var DATA
    )
END
```

It receives the fully computed output from the HtmlCreator module and writes it directly to the HTML file. In order to retrieve data from the HtmlCreator, some other data has to be input first. This is done in the following rule.

```
CONSTRUCT
    to HtmlCreator (
        htmlcreator [
            title [ "Music Center" ],
            stylesheet [ "data/yellowstyle.css" ],
            body [
                var BodyContents
            ]
        ]
    )
FROM
    in MobileTableLayout (
        var BodyContents
    )
END
```

The HtmlCreator module is fed with an `htmlcreator` node containing three child nodes which can be seen as a kind of parameters: the title of the page to be produced, the CSS stylesheet to be applied and the actual page content. The specification of the stylesheet is optional. This fact is specified in the HtmlCreator module and must be documented in its interface description. The page content, stored in variable `BodyContents` is drawn from another module, the MobileTableLayout in this example.

The MobileTableLayout, too, must get its data from somewhere.

```
CONSTRUCT
    to MobileTableLayout (
        var CDINFO
     )
FROM
    virtualroot[[var CDINFO]]
END
```

This rule sends all `cd` nodes from the `virtualroot` and sends it to the MobileTableLayout module. The `virtualroot` nodes are created by the data import facility that has been explained above. Note that, even though `CDINFO` does not match all children `cd` nodes at once but one after another, there is no `all` keyword in the construct clause. The matched data is sent to the module sequentially.

The TableLayout modules As already explained, there are two modules that compute the `body` content of the final output document. Only one of them can be active at a time.

The MobileTableLayout module contains a rule to make a simple table from the data that it receives.

```
MODULE MobileTableLayout

CONSTRUCT
    public
        table [
            all tr [
                td [ var ARTIST ],
                td [ var TITLE ]
            ]
        ]
FROM
    public cd [[
        artist [var ARTIST],
        title [var TITLE]
    ]]
END
```

The table construction code is the same code that was originally in the `GOAL` rule, before we started employing modules for output computation. Consequently, the output is the same as depicted in figure 2.

The ElaborateTableLayout module produces a more sophisticated table, which also includes CD covers and the list of songs.

```
MODULE ElaborateTableLayout

CONSTRUCT
    public
        table [
            attributes {
                class {"maintable"}
            },
            all var DATASET
        ]
FROM
    alldata[[
        dataset[[var DATASET]]
    ]]
END


CONSTRUCT
    alldata [
        all dataset [
            tr [
                attributes {
                    class {"heading"}
                },
                td [
                    attributes {
                        colspan { "2" }
                    },
                    var ARTIST, ":", var TITLE
                ]
```

```
            ],
            tr [
                attributes {
                    class {"value"}
                },
                td [
                    img[
                        attributes {
                            src { var COVERLINK },
                            width { "250" },
                            height { "250" }
                        }
                    ]
                ],
                td [
                    table [
                        attributes {
                            class { "songtitles" }
                        },
                        all tr [
                            td [ var SONG ]
                        ]
                    ]
                ]
            ]
        ]
    ]
FROM
    public cd [[
        artist [var ARTIST],
        title [var TITLE],
        coverlink [var COVERLINK],
        songs [[
            song [var SONG]
        ]]
    ]]
END
```

Here we see a module that consists of two rules. This is due to a limitation in Xcerpt. The layout should use two table rows for each data set, the first one for artist and title, the second one for cover and songlist.

```
<table>
    <tr>
        <td colspan="2">artist 1 : title 1</td>
    </tr>
    <tr>
        <td>cover 1</td>
        <td>list of songs 1</td>
    </tr>
       ...
    <tr>
        <td colspan="2">artist x : title x</td>
    </tr>
    <tr>
        <td>cover x</td>
        <td>list of songs x</td>
    </tr>
</table>
```

To build a table containing all CD data, the `all` keyword has to be used in the construction clause. Unfortunately, it cannot. Placing the `all` keyword in front of the `table` tag would result in an individual table for each CD data set.

```
<table>
    data 1
</table>
...
<table>
    data x
</table>
```

Placing the `all` keyword in front of each table row (`tr` tag) would result in a table which contains all rows with artists and titles, followed by all rows with covers and songlists.

```
<table>
    <tr>
        <td colspan="2">artist 1 : title 1</td>
    </tr>
       ...
    <tr>
        <td colspan="2">artist x : title x</td>
     </tr>
    <tr>
        <td>cover 1</td>
        <td>list of songs 1</td>
     </tr>
       ...
    <tr>
        <td>cover x</td>
        <td>list of songs x</td>
    </tr>
</table>
```

We actually need something between the `table` and the `tr` node to place the `all` keyword in front of, but this would result in invalid HTML. Splitting the construction into two rules solves the dilemma.

The lower rule creates the two table rows and groups them together under a temporary `dataset` node. All `dataset` nodes (one for each CD data) are grouped together under the `alldata` node.

The upper rule takes all table rows from the `dataset` nodes and adds them under a `table` node, which results in valid HTML. Figure 6 shows the created output in a browser.

The HtmlCreator module As already mentioned, the HtmlCreator module creates the HTML output. It receives the page title, the CSS stylesheet location and the actual body content from outside and puts them together.

```
MODULE HtmlCreator

CONSTRUCT
    public
        html [
            attributes {
                xmlns { "http://www.w3.org/1999/xhtml" },
                lang { "en" }
            },
```

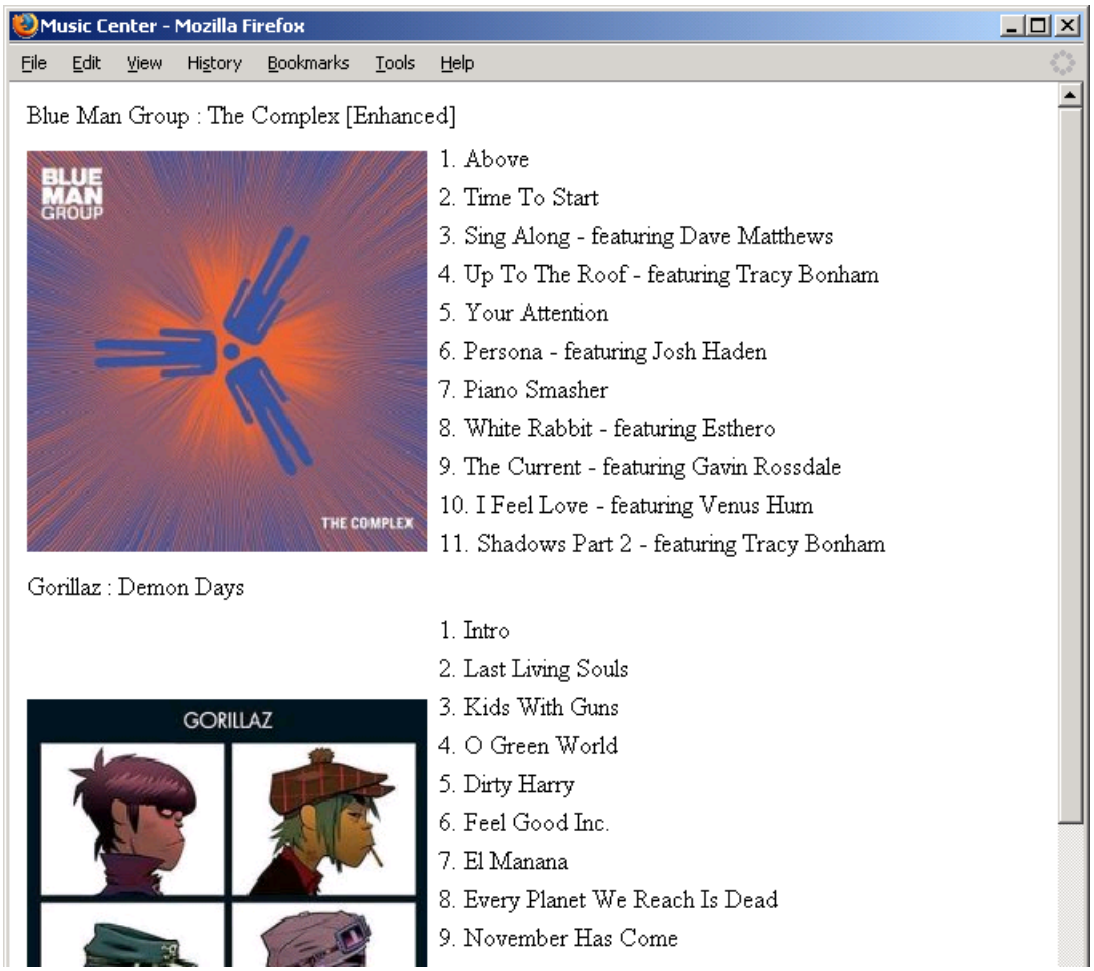

Figure 6: Output produced by ElaborateTableLayout

```
10            head [ var TITLE, optional var STYLE],
              var BODY
12        ]
  FROM
14      or {
          webpagedata [ var TITLE, var BODY ],
16        style [var STYLE]
      }
18  END
```

The `FROM` clause matches all `webpagedata` and `style` nodes. The keyword `optional` in front of the `STYLE` variable in the `CONSTRUCT` clause indicates that the rule can be applied even if there is no style available.

The `webpagedata` and `style` nodes are produced by the following helper rules.

```
CONSTRUCT
```

```
    webpagedata [ var TITLE, var BODY ]
FROM
    public
        htmlcreator [[
            var TITLE →title [[ ]],
            var BODY →body [[ ]]
        ]]
END


CONSTRUCT to StyleSheet (
    var DATA
)
FROM
    public
        htmlcreator [[
            var DATA →stylesheet [[ ]]
        ]]
END


CONSTRUCT
        style[ var STYLE ]
FROM
        in StyleSheet (
                var STYLE
        )
END
```

Both the rules for `webpagedata` and `style` match the `htmlcreator` node that is being passed to the HtmlCreator module from the main program. The former rule just wraps the title and body "parameters" in a `webpagedata` tag. The latter rule creates the HTML code for CSS inclusion with the help of the StyleSheet module. This additional module has been introduced to display the usage of nested modules.

The StyleSheet module In the main program one can easily specify the CSS stylesheet to be used by the final HTML output. The StyleSheet module builds the appropriate HTML code around it.

```
MODULE StyleSheet

CONSTRUCT
    public
    link [
        attributes {
            rel {"Stylesheet"},
            type {"text/css"},
              href {var STYLESHEET}
          }
    ]
FROM
    public
        stylesheet [var STYLESHEET]
END
```

For example, specifying stylesheet ["data/yellowstyle.css"] will result in the following HTML code:

```
1 <link rel="Stylesheet" type="text/css" href="data/yellowstyle.css" />
```

Figure 7 shows the output created with the ElaborateTableLayout module and the yellowstyle stylesheet applied.

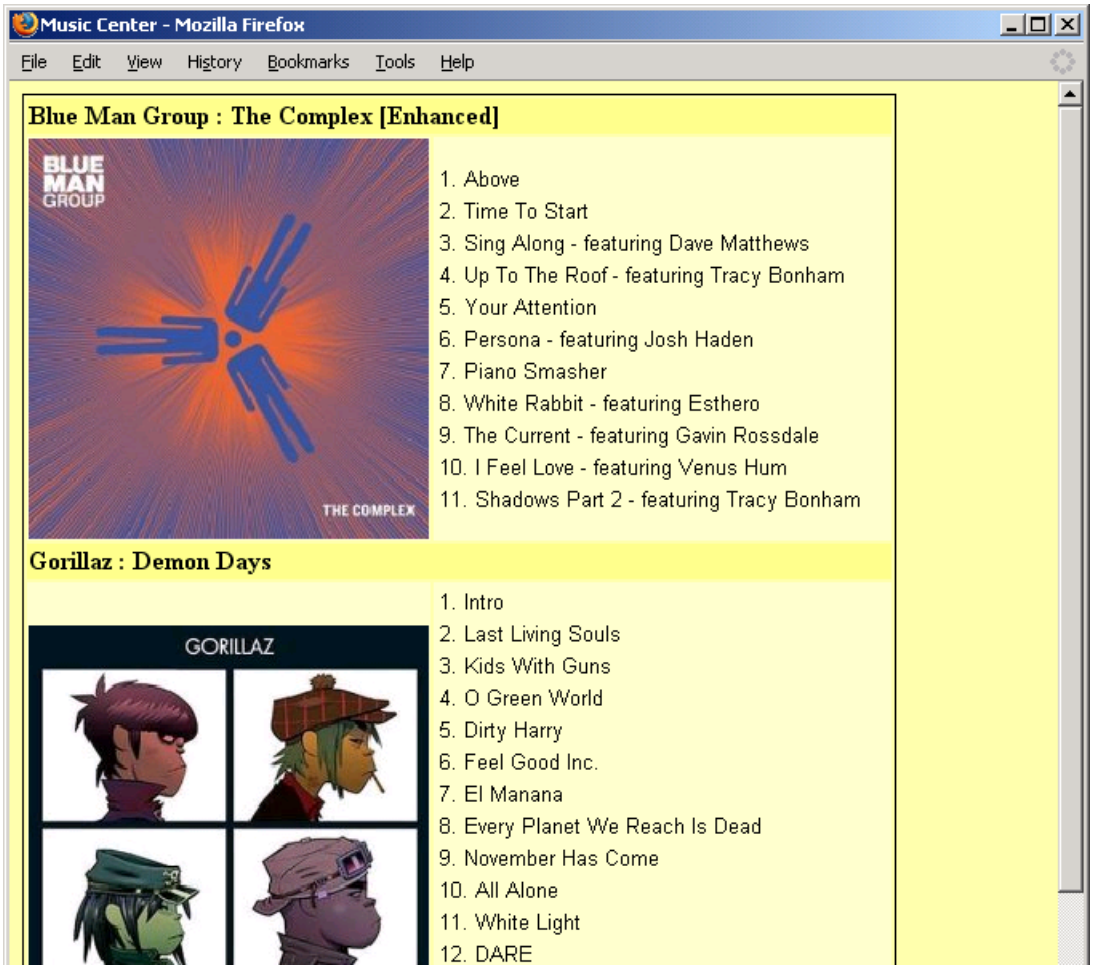

Figure 7: ElaborateTableLayout output with stylesheet applied

Summary Modules have been used to create a flexible set of rules. These can be easily exchanged and parameterized to control the final output of the program. Furthermore, it has been shown that modules can be nested.

2.3 Role modeling with Ontologies

For ontologies we have investigated another kind of component, the *role model*. A role model is a unit of deployment in role modeling, a discipline going back as far as to the advent of

database modeling. We do not introduce role modeling here in detail, but for a starting point for role modeling in ontologies see [10] and references therein.

Essentially, a role model constitutes a set of related concepts (in role modeling called *role types*) and their relationships. We believe that role models constitute useful and natural units for component-based ontology engineering. Role models are developed as components and intended to be deployed as such, in contrast to existing approaches aimed at extracting ontological units from ontologies not necessarily designed to be modular. In the following we give some simple examples of role models and how they can be reused in ontologies.

A role model can be said to capture a particular concern of a more fundamental domain. What makes role models reusable is that a particular concern can be reused in different domains, hence, different ontologies. The role model below describes the *product* concerns and introduces concepts such as *Product*, *Producer*, *Consumer* etc. What is important in role modeling, and in particular for role types, are relationships. Hence, the role model below describes the relationships between the introduced concepts, relationships that hold for the concepts regardless of the domain they are used in. That is, a producer always produces some product, which is being consumed by someone (the consumer) etc.

```
Rolemodel: Product

        Role: Product
                EquivalentTo: ( maker SOME Producer )

        Role: Producer
                EquivalentTo: ( produces SOME Product )

        Role: Consumer
                EquivalentTo: ( consumes SOME Product )
```

The role model above is based on Manchester Syntax, a syntactical variant for OWL [8]. For the purpose of defining role models we extend the considered syntax with the constructs *Rolemodel* and *Role* (demonstrated above, the other syntactical constructs belong to the Manchester Syntax). For example, in the above, the role type *Product* is stated to be equal to the set of individuals that are in the relationship *maker* with some individual of concept (role type) *Producer*. Essentially, the domain of the relationship *maker* is stated to be the role type *Product*. The other concepts (role types) are defined in a similar fashion.

The above role model describes role types related to products. Thus, any domain ontology that needs to talk about, or model, products can reuse the role model. Below is a domain ontology relating to wines that reuses the role model.

```
Ontology:

Import /Product.rowlm

Class: Wine
        Plays: Product
        DisjointWith: Pizza

Class: RedWine
        SubClassOf: Wine
        EquivalentTo: hasColor

Class: WhiteWine
        SubClassOf: Wine
```

```
        EquivalentTo: hasColor

Class: Pizza
        Plays: Product

Class: Winery
        Plays: Producer

Wine(merlot99_1)
Pizza(fourseasons_1)
```

The ontology above is also based on the Manchester Syntax, but it is additionally extended with two constructs to import and integrate role models: *Import* and *Plays*. A role model can be imported using the *Import* construct. Concepts in the ontology can then be associated to the role types in the imported role model using the *Plays* construct. In the ontology above, for example, it is stated that wines can play the role of being products. The benefit is that all the relationships between role types encoded in the role model are reused for the *Wine* concept. The same holds for the concept *Pizza* etc.

The above ontology, written in an extended language for the purpose of role modeling, is compiled down to an ontology written in the underlying non-extended language. This compilation can be described by the following algorithm:

1. Make all imported role type definitions available as classes in the ontology.
2. For each role type R used in the ontology:
   (a) Let $\{C_1, \ldots, C_n\}$ be the set of classes which R is said to play (using the *Plays* construct). Then add the axiom $R \sqsubseteq C_1 \sqcup \ldots \sqcup C_n \sqcup \bot$ to the ontology.
   (b) For each role assertion $R(a)$, make the same assertion available in the resulting ontology, now referring to the class-representative for the role type R.
3. Remove *Import* and *Plays* statements.

The above semantics implies that a non-played role R may not be instantiated by any individual since $R \sqsubseteq \bot$ would be added to the ontology (i.e. R is always interpreted as the empty set). The semantics of our role modeling extension is an immediate consequence of the translation by using the standard semantics of standard ontology languages (such as OWL).

For example, our ontology above would be compiled into the following non–role-based ontology.

```
Ontology:
        Class: Consumer
               SubClassOf: Bottom
               EquivalentTo: ( consumes SOME Product )

        Class: Producer
               SubClassOf: Winery
               EquivalentTo: ( produces SOME Product )

        Class: Product
               SubClassOf: (Pizza OR Wine)
               EquivalentTo: ( maker SOME Producer )

        Class: Wine
```

```
            DisjointWith: Pizza

    Class: RedWine
            SubClassOf: Wine
            EquivalentTo: hasColor

    Class: WhiteWine
            SubClassOf: Wine
            EquivalentTo: hasColor

    Class: Pizza
    Class: Winery

    Wine ( merlot99_1 )
    Pizza ( fourseasons_1 )
```

Figure 8 shows a screenshot of the environment for composing ontologies using role models.

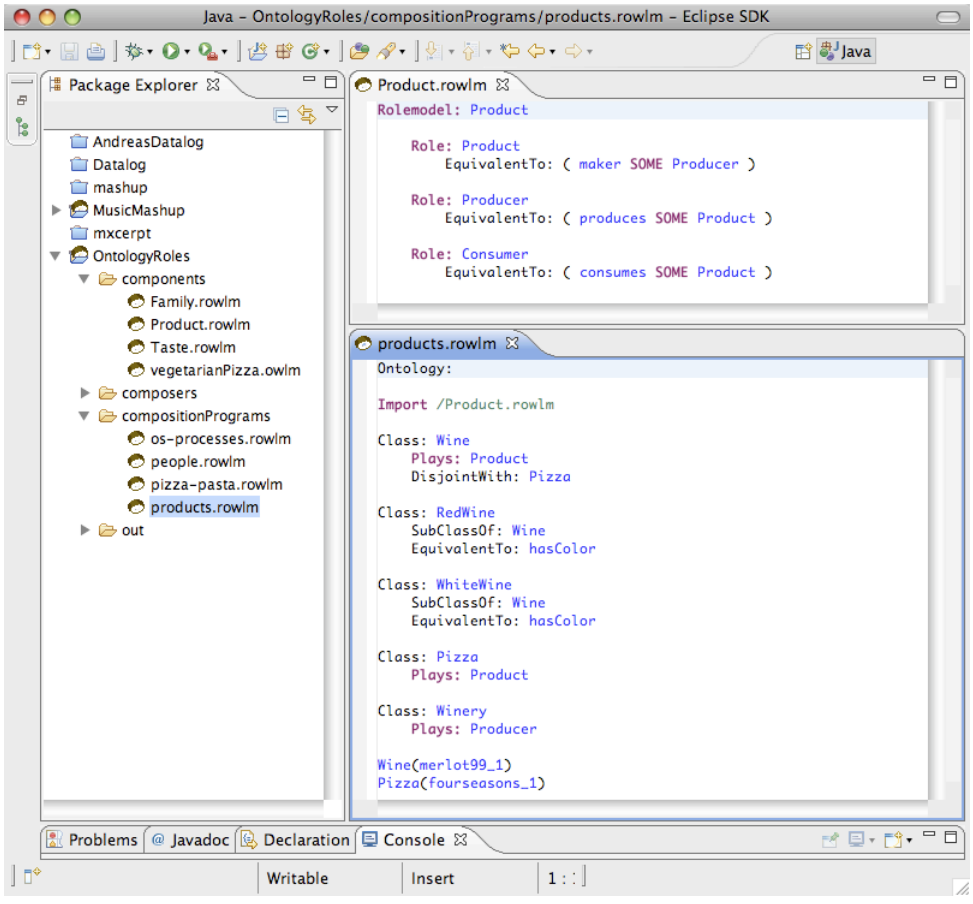

Figure 8: Screenshot of the *Product* role models and an ontology importing the role model.

Summary In the above we have shown the definition and use of role models. Role models are interesting reuse units for ontologies since they cross-cut domains, and hence, domain ontologies.

Part III
Typing

This part describes typing technologies for finding errors in the rule-based query and transformation language Xcerpt.

3 Typing tool

This section presents a prototype of the type system (typechecker) *XcerptT*. Similarly to the prototypical runtime system for Xcerpt, it has been implemented in the functional language Haskell. The prototype has been attached as a module to the Xcerpt prototype. The current version of the typechecker supports type specifications given with the formalisms of Type Definitions or DTDs.

The prototype is restricted to the fragment of Xcerpt for which the formal semantics is provided. Moreover, it is restricted to non recursive Xcerpt programs. It is still under development and the goal is to extend it towards the full Xcerpt.

The prototype of the type system together with the Xcerpt runtime system can be accessed online via the link `http://ida.liu.se/~artwi/XcerptT`.

3.1 Usage of the Prototype

This section uses the notation where square brackets [ ] and the elements denoted by triangle parentheses <...> belong to a metalanguage[2].

The type system is invoked like the standard Xcerpt runtime system (i.e. executing `xcerpt` or `xcerpt.exe`). To perform type checking (or type inference) of a program a parameter `-t` is used:

`xcerpt -t` <*program file*> [<*type specification*>]

The typing mechanism can also be invoked using the interactive Xcerpt command mode with the command:

`:type` <*program file*> [<*type specification*>]

In the abovementioned commands <*program file*> is an Xcerpt program and <*type specification*> is a text file specifying the types of external resources (i.e. the resources referred to in targeted query terms $\texttt{in}(r, q)$ in the program) and the types of expected results. <*type specification*> file may contain:

- a Type Definition i.e. rules defining types,
- input type specifications; each such specification specifies a type $type(r)$ of a queried resource r,

[2] [ ] represents optional part and <...> is a nonterminal which can be replaced with a text without spaces.

- output type specifications specifying result types for particular rules.

The input type specification has the syntax:

```
Input::
  [ resource = <resource URI> ]
  [ typedef  = <typedef location> ]
  typename = <type name>
```

and the output type specification has the syntax:

```
Output::
  [ rule   = <index>]
  [ typedef  = <typedef location>]
  typename = <type name>
```

where

- *<resource URI>* is an URI of the resource being queried whose type we specify. If the parameter `resource` is omitted the input type specification specifies a type of every resource occurring in the *<program file>* whose type was not specified (overridden) by other input type specification. A *<type specification>* can contain at most one input type specification without the parameter `resource`.

- *<typedef location>* is a URI of an external file containing a Type Definition (or DTD). If the the parameter `typedef` is omitted the input or output type specification refers to the local Type Definition i.e. specified in the current *<type specification>* file.

- *<type name>*, if used in an input type specification, it is a type name specifying the type of the resource the specification refers to. If it is used in an output type specification it is a type name specifying the result type of the rule the specification refers to. It can be the most general type `Top` or a type name which is defined in the Type Definition or the DTD the input or output type specification refers to. If the specification refers to a DTD then a type name can be one of the element names declared in the DTD.

- *<index>* is a number of the query rule in the Xcerpt program whose output type we specify. It can be obtained by counting the query rules in the program starting from one e.g. the index of the second query rule in a program is 2. If the parameter `rule` is omitted the output type specification concerns the first goal in the Xcerpt program (or the first query rule if the program contains no goals). A *<type specification>* can contain at most one output type specification without the parameter `rule`.

This is an example of a <type specification> file *books.xts*:

```
Publications -> publications[ Book* Article* ]
Article -> article[ Title Author+ Proceedings ]
Proceedings -> proceedings[ Title Editor+ ]
Book -> book[ Title Author+ Editor+ ]
Title -> title[Text ]
Author -> author[ P ]
Editor -> editor[ P' ]
```

```
P -> person[ S ]
P' -> person[ F? S? ]
Person -> person[ F+ S ]
F -> firstname[ Text ]
S -> surname[ Text ]
AuthorsEditors -> authors-editors[ Person+ ]

Input::
  resource = file:publications.xml
  typename = Publications

Output::
  rule = 1
  typename = AuthorsEditors
```

Invoking the typing mechanism (e.g. with the command `xcerpt -t` <*program file*> <*type specification*>) starts the process of type inference for the program. The type inference is done using the knowledge of types of resources given by input type specifications. If the type of a resource is not specified by any input type specification it is assumed to be the most general type $\top$ (which can be seen as a default type of a resource). After the type of results for each query rule of the program has been inferred, type checking is performed for each query rule for which output type specification is provided and for which the inferred result type is not empty. Type checking for a rule includes an inclusion check: it is checked whether the inferred result type is included in the corresponding output type (specified by the output type specification). If the check fails an intersection emptiness check is performed: it is checked whether the intersection of the inferred result type and the specified output type is empty. Thus there are three possible results of type checking for a rule:

- *OK* - the inferred result type is included in the specified type i.e. the rule is correct wrt. the specified type,
- *Failed* - the intersection of the inferred result type and the specified type is empty i.e. there is a weak type error for the rule,
- *Unsuccessful* - the inferred result type is not included in the specified type but the intersection of both types is not empty i.e. the rule may be incorrect wrt. the specified type.

Invoking the typing mechanism without <*type specification*> parameter has the same effect as invoking it with an empty <*type specification*> file.

As a result of typing an Xcerpt program we get a printout that for each query rule of the program, contains:

- the inferred result type for the rule, for example, `Rule 2: Person` (0 stands for empty result type)
- if there is a result type specified for the rule and if the inferred result type is not empty, the result of type checking, for example,
  `Type checking: Failed`,
- variable-type mappings for variables occurring in the rule

Moreover the printout contains a Type Definition defining all the inferred types and the types of the queried resources.

For the types being intersection of other types their content model is provided by a DFA instead of a regular type expression. A DFA is presented by descriptions of all its states. Each such a description is of the form $\mathtt{S_i} => \mathtt{a_1} > \mathtt{S_{k_{i1}}} \ldots \mathtt{a_n} > \mathtt{S_{k_{in}}}$, where S_i is the number of the state being described, $a_1, \ldots, a_n$ are the symbols of the alphabet on which the DFA is defined and each $S_{k_{ij}}$ is the number of the state reached from the state S_i by reading the symbol a_j. Additionally, the number of the state being described may be preceded by the character '>' which denotes the initial state or it may be followed by the character '!' which denotes a final state. This is an example of a DFA corresponding to the language defined by a regular expression AF^*:

```
 0  => A>0 F>0
>1  => A>2 F>0
 2! => A>0 F>2
```

A name given by the system for a type being the intersection of types T_1,T_2 is $T_1\hat{}T_2$. The type checker also invents type names for the newly inferred types. The devised new type names are the labels of the corresponding construct terms occurring in heads of query rules. If there is a need to define a type with a type name which has already been used the new type name is augmented with an index i.e. a number added at the and of the type name (underscore separated). If a type name with a given index already exists the new type name has the index increased by 1.

Here we present an output of the type system prototype for the following Xcerpt program:

```
GOAL
  authors-editors[ all var X ]
FROM
  books[[
    book{{
      title[ var Y ],
      author[ var X ],
      editor[ var X ]
    }}
  ]]
END

CONSTRUCT
  books[ all var X ]
FROM
  in{ resource{ "file:publications.xml" },
    desc var X →book{{ }}
  }
END
```

A type specification for the program is given by *publications.xts* file from the previous example. The obtained output is:

```
============================================================
Rule 1: authors-editors
Type checking: Failed (no results of type AuthorsEditors)
------------------------------------------------------------
Y->Text, X->P^P'
```

```
=========================================================
Rule 2: books
---------------------------------------------------------
X->Book
=========================================================
=========================================================
Type Definition:
---------------------------------------------------------
authors-editors -> authors-editors[ P^P'+ ]
books -> books[ Book+ ]
Publications -> publications[ Book* Article* ]
Article -> article[ Title Author+ Proceedings ]
Proceedings -> proceedings[ Title Editor+ ]
Book -> book[ Title Author+ Editor+ ]
Title -> title[ Text ]
Author -> author[ P ]
Editor -> editor[ P' ]
P -> person[ S ]
P' -> person[ F? S? ]
Person -> person[ F+ S ]
F -> firstname[ Text ]
S -> surname[ Text ]
AuthorsEditors -> authors-editors[ Person+ ]
P^P' -> person[
 0  => S>0
>1  => S>2
 2! => S>0
 ]
=========================================================
```

The printout contains the inferred result types for the first and the second rule, which are respectively, *authors-editors* and *books*. It also contains information of the inferred types for the particular variables occurring in the rules. All the types are defined by the Type Definition from the bottom of the printout. As an output type specification is provided for the first rule the printout contains the result of type checking for the rule.

Figure 9 presents a screenshot of the online type system interface.

4 Use-cases and examples

This section presents examples of simple scenarios showing the way the presented type system can be helpful for programmers for checking correctness of Xcerpt programs. The examples show the way the type system can facilitate finding errors in programs. The programs presented in this section, except the last one (as the prototype is not operational for recursive programs), have been type checked by our prototype and the corresponding printouts are presented in Section 4.4.

4.1 Usecase 1: CD Store

Consider a Type Definition:

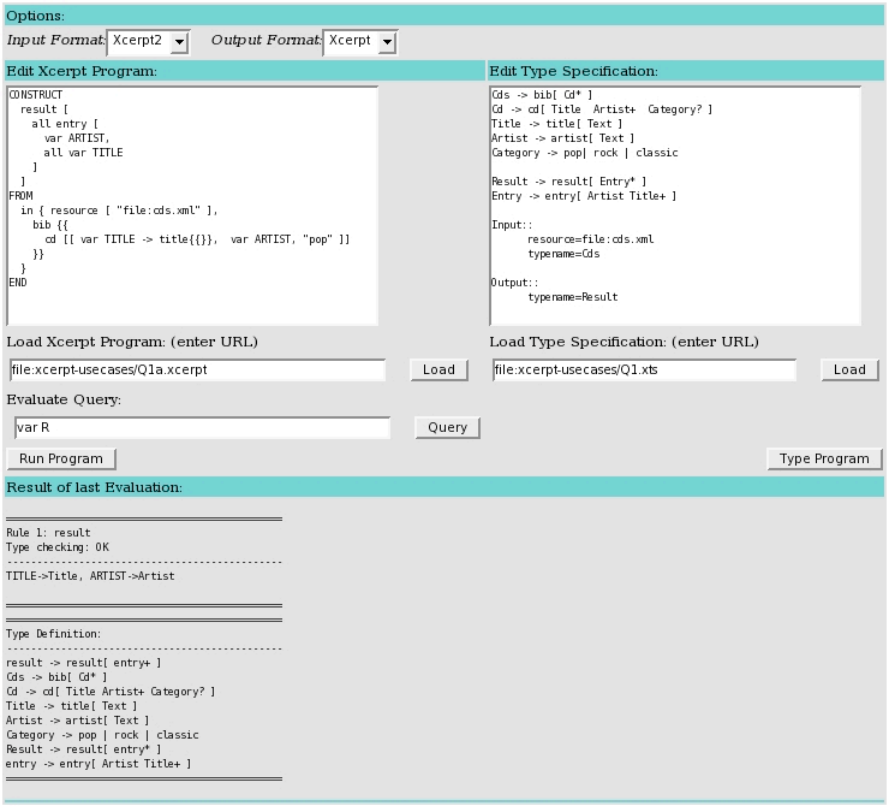

Figure 9: Type system prototype interface.

$$
\begin{array}{lcl}
Cds & \rightarrow & bib\,[\,Cd^{*}\,] \\
Cd & \rightarrow & cd\,[\,Title\ \ Artist^{+}\ \ Category^{?}\,] \\
Title & \rightarrow & title\,[\,Text\,] \\
Artist & \rightarrow & artist\,[\,Text\,] \\
Category & \rightarrow & \text{"}pop\text{"}\ \mid \text{"}rock\text{"}\ \mid \text{"}classic\text{"}
\end{array}
$$

The query rule below queries a document *cds.xml* of the type *Cds* defined above. The intention of the query rule is to collect artists together with all the titles of the CD's of the category "pop".

```
CONSTRUCT
  pop-entries[
    all entry[
      var ARTIST,
      all var TITLE
    ]
  ]
FROM
```

```
 9  in{ resource[ "file:cds.xml" ],
      bib{{
11      cd[[ var TITLE, var ARTIST, "pop" ]]
      }}
13  }
  END
```

First, we assume that no result type specification is given for the rule.

Printout CDSTORE.1 in Section 4.4 is the result of typing the query rule by the type-checker. We assume that the intention of an author of the query rule is that the variable TITLE will be bound to data terms *title*[...] and the variable ARTIST will be bound to data terms *artist*[...]. The type system infers the types of variables used in the query rule. They are given by the following variable-type mappings: $[TITLE \mapsto Title,\ ARTIST \mapsto Artist]$, $[TITLE \mapsto Artist,\ ARTIST \mapsto Artist]$. As the variable TITLE is intended (by the programmer) to take values only of the type *Title*, the inferred types for variables suggest that the query rule is incorrect with respect to the programmer's expectations.

Based on the inferred types of variables the query rule result type is inferred. The inferred result type is *pop-entries* defined as

$$\begin{array}{lcl} pop\text{-}entries & \rightarrow & pop\text{-}entries\,[\,entry^{+}\,] \\ entry & \rightarrow & entry\,[\,Artist\ (Title \mid Artist)^{+}\,] \\ Artist & \rightarrow & artist\,[\,Text\,] \\ Title & \rightarrow & title\,[\,Text\,] \end{array}$$

Looking at this definition of the inferred result type of the rule, the programmer can also realize that the results of the rule may be different from expected ones (as an entry should not contain more than one artist).

Now, let us assume that a result type specification is provided for the rule and the specified result type is *Entries* as defined below:

$$\begin{array}{lcl} Entries & \rightarrow & pop\text{-}entries\,[\,Entry^{*}\,] \\ Entry & \rightarrow & entry\,[\,Artist\ \ Title^{+}\,] \\ Artist & \rightarrow & artist\,[\,Text\,] \\ Title & \rightarrow & title\,[\,Text\,] \end{array}$$

Printout CDSTORE.2 in Section 4.4 corresponds to this case. Now the system can automatically check that the inferred result type *pop-entries* is not included in the type *Entries* (as the type *entry* is not a subtype of the type *Entry*). This information suggests a type error. However, a type inclusion check failure is not a proof of a type incorrectness of the program as the inferred result type *pop-entries* is not exact[3] (as the query rule uses the construct `all`). There is no weak type error for the rule as the intersection of the types *Entries* and *pop-entries* is not empty. Nevertheless, there is a type error for the rule. The intention for the query rule is to produce a result containing entries with one artist and all his/her titles (at least one). However, the query rule may produce a result with entries containing more than one artist, for example:

[3] The inferred result type *pop-entries* is a superset of the actual set of possible results which is (*Entries′*) defined as

$$Entries' \rightarrow pop\text{-}entries[Entry'^{+}]$$
$$Entry' \rightarrow entry[Artist\ (Title \mid Artist)^{*}\ Title\ (Title \mid Artist)^{*}]$$

```
pop-entries[
  entry[ artist[ "artist1" ], title[ "title1" ] ],
  entry[ artist[ "artist2" ], title[ "title1" ], artist[ "artist1" ] ]
]
```

The abovementioned result is obtained if the query rule is applied to a data term:

```
bib[
  cd[
    title[ "title1" ],
    artist[ "artist1" ],
    artist[ "artist2" ],
    "pop"
  ]
]
```

4.2 Usecase 2: Bibliography

Consider the following Type Definition:

$$
\begin{array}{lcl}
\mathit{Bibliography} & \rightarrow & \mathit{bib}\,[\,(\mathit{Book} \mid \mathit{Article} \mid \mathit{InProceedings})^*\,] \\
\mathit{Book} & \rightarrow & \mathit{book}\,\{\,\mathit{Title}\ \mathit{Authors}\ \mathit{Editors}\ \mathit{Publisher}^?\,\} \\
\mathit{Article} & \rightarrow & \mathit{article}\,\{\,\mathit{Title}\ \mathit{Authors}\ \mathit{Journal}^?\,\} \\
\mathit{InProceedings} & \rightarrow & \mathit{inproc}\,\{\,\mathit{Title}\ \mathit{Authors}\ \mathit{Book}\,\} \\
\mathit{Title} & \rightarrow & \mathit{title}\,[\,\mathit{Text}\,] \\
\mathit{Authors} & \rightarrow & \mathit{authors}\,[\,\mathit{Person}^*\,] \\
\mathit{Editors} & \rightarrow & \mathit{editors}\,[\,\mathit{Person}^*\,] \\
\mathit{Publisher} & \rightarrow & \mathit{publisher}\,[\,\mathit{Text}\,] \\
\mathit{Journal} & \rightarrow & \mathit{journal}\,\{\,\mathit{Title}\ \mathit{Editors}\,\} \\
\mathit{Person} & \rightarrow & \mathit{person}\,[\,\mathit{FirstName}\ \mathit{LastName}\,] \\
\mathit{FirstName} & \rightarrow & \mathit{first}\,[\,\mathit{Text}\,] \\
\mathit{LastName} & \rightarrow & \mathit{last}\,[\,\mathit{Text}\,]
\end{array}
$$

4.2.1 No Result Type Specified

The query rules from this section query a document *bibliography.xml* of the type *Bibliography* defined above.

```
CONSTRUCT
  result[
    all var AUTHOR,
    titles[ all var TITLE ]
  ]
FROM
  in{ resource[ "file:bibliography.xml" ],
    Bib{{
      Book{{ Author[ var AUTHOR ], Title[ var TITLE ] }}
    }}
  }
END
```

The corresponding printout in Section 4.4 is BIBLIOGRAPHY.1. The query rule returns no results when it is applied to a document of type *Bibliography* because of the labels' mismatch.

The labels occurring in the body of the rule are written with capital letters while labels occurring in the Type Definition are written with lower case letters. Thus, the query rule does not match the type of the database and the result type inferred for this query rule is empty. We get an emptiness error for the rule.

This is another example of a query rule with an emptiness error.

```
CONSTRUCT
  results[
    all publisher[ var NAME , var URL ]
  ]
FROM
  in{ resource[ "file:bibliography.xml" ],
    bib{{
      book{{ publisher[ name[ var NAME ], url[ var URL ] ] }}
    }}
  }
END
```

The corresponding printout in Section 4.4 is also BIBLIOGRAPHY.1. The inferred result type is empty due to the fact that the query term in the body of the query rule cannot be matched against data terms of type *Bibliography*. This is because the query looks for *name*[...] and *url*[...] as direct subterms of *publisher*[...] while data terms of type *Publisher* contain only text.

The next query rule does not match the document because of the square brackets used to match data terms *book*{...}. According to the type of the document direct subterms of *book*{...} are unordered and cannot be matched with a query term being an ordered pattern.

```
CONSTRUCT
  result[
    all var AUTHOR,
    titles[ all var TITLE ]
  ]
FROM
  in{ resource[ "file:bibliography.xml" ],
    bib[[
      book[[ title[ var TITLE ], author[ var AUTHOR ] ]]
    ]]
  }
END
```

The corresponding printout in Section 4.4 is also BIBLIOGRAPHY.1.

An emptiness error is obtained also for the next query rule. This is caused by the wrong usage of the variable PERSON. Its first occurrence will be bound to data terms of the type *Person* while its second occurrence will be bound to direct subterms of a data term *person*[...] which can be of the type either *FirstName* or *LastName*. As the intersection of type *Person* with each of latter types is empty the inferred query result type is also empty.

```
CONSTRUCT
  result[
    all var PERSON,
    titles[ all var TITLE ]
  ]
FROM
  in{ resource[ "file:bibliography.xml" ],
    bib{{
      book{{
        editors{{ var PERSON }}
```

```
      }},
12    book{{
        authors{{
14        person{{ var PERSON }}
        }}
16    }}
    }}
18  }
END
```

The corresponding printout in Section 4.4 is BIBLIOGRAPHY.1.

4.2.2 Result Type Specified

The next example of a query rule illustrates a transformation of an XML document to a format similar to HTML. The format is defined by the following Type Definition which specifies the result type *TextBook* [4]:

$$
\begin{array}{lcl}
\mathit{TextBook} & \rightarrow & \mathit{book}\,[\,\mathit{Cover}\ \mathit{Body}\,] \\
\mathit{Cover} & \rightarrow & \mathit{cover}\,[\,\mathit{Title}\ \mathit{Author}^{*}\ \mathit{Publisher}^{?}\,] \\
\mathit{Body} & \rightarrow & \mathit{body}\,[\,\mathit{Abstract}^{?}\ \mathit{Chapter}^{*}\,] \\
\mathit{Title} & \rightarrow & \mathit{title}\,[\,\mathit{Text}\,] \\
\mathit{Author} & \rightarrow & \mathit{author}\,[\,\mathit{Text}\,] \\
\mathit{Publisher} & \rightarrow & \mathit{publisher}\,[\,\mathit{Text}\,] \\
\mathit{Abstract} & \rightarrow & \mathit{abstract}\,[\,\mathit{Text}\,]
\end{array}
$$

$$
\begin{array}{lcl}
\mathit{Chapter} & \rightarrow & \mathit{chapter}\,[\,\mathit{Title}\ \mathit{Section}^{*}\,] \\
\mathit{InlineContent} & \rightarrow & \mathit{inline}\,[\,\mathit{Text} \mid \mathit{Bf} \mid \mathit{Em}\,] \\
\mathit{Section} & \rightarrow & \mathit{section}\,[\,\mathit{Title}^{?}\ (\mathit{Paragraph} \mid \mathit{Table} \mid \mathit{List})^{*}\,] \\
\mathit{Em} & \rightarrow & \mathit{em}\,[\,\mathit{InlineContent}\,] \\
\mathit{Bf} & \rightarrow & \mathit{bf}\,[\,\mathit{InlineContent}\,] \\
\mathit{Paragraph} & \rightarrow & \mathit{p}\,[\,\mathit{InlineContent}^{*}\,] \\
\mathit{Table} & \rightarrow & \mathit{table}\,[\,\mathit{TableRow}^{+}\,] \\
\mathit{List} & \rightarrow & \mathit{list}\,[\,\mathit{ListItem}\,] \\
\mathit{TableRow} & \rightarrow & \mathit{tr}\,[\,\mathit{TableCell}^{*}\,] \\
\mathit{ListItem} & \rightarrow & \mathit{item}\,[\,\mathit{InlineContent}^{*}\,] \\
\mathit{TableCell} & \rightarrow & \mathit{td}\,[\mathit{InlineContent}^{*}\,]
\end{array}
$$

Consider a query rule which queries a document *bibliography.xml*:

```
1 CONSTRUCT
  book[
3   cover[ title[ "List_of_Books" ] ],
    body[
5     table[
        all tr[
7         td[ var TITLE],
          td[ all em [ var FIRST, var LAST ] ]
9       ]
      ]
```

[4] The Type Definition and the two following examples of query rules were devised by Sacha Berger.

```
    ]
  ]
FROM
  in{ resource [ "file:bibliography.xml" ],
    bib{{
      book{{
        title[ var TITLE ],
        authors[[
          person{{
            first[ var FIRST ],
            last[ var LAST ]
          }}
        ]]
      }}
    }}
  }
END
```

Let us assume that no type specification for the document *bibliography.xml* is provided. In such case the system infers a very rough approximation of the set of results of the rule (printout BIBLIOGRAPHY.2 in Section 4.4) and the inferred result type *book* is not included in the specified result type *TextBook*. Thus a type error is possible. To make sure about that it is checked whether the intersection of the inferred result type and the specified result type is empty. Indeed the intersection is empty. A weak type error, which we get, implies that the rule will not return any results of the type *TextBook*. Notice that the weak type error has been obtained without any type specification for the queried data.

The weak type error is due to the structure of the construct term used as a head of the query rule. The construct term creates a data term *body*[...] with a data term *table*[...] as a direct subterm. According to the type specification *body*[...] can not contain any *table*[...] direct subterms. Note that in this case the inferred types of variables do not matter. Whatever variable-type mappings we get from the body of the query rule the result type is still wrong due to the structure of the construct term which does not conform to the specified result type.

Consider another query rule which queries the document *bibliography.xml*. This time we assume that both type specifications are given i.e. type specification for the document *bibliography.xml* (the type *Bibliography*) and a result type specification for the rule (the type *TextBook*).

```
CONSTRUCT
  book[
    cover[ title [ "Books" ] ],
    body[
      chapter[
        title[ "List_of_Books_and_Authors"],
        section[
          table[
            all tr[
              td[ inline [ var TITLE ] ],
              td[ inline [ var NAME ] ]
            ]
          ]
        ]
      ]
    ]
  ]
FROM
```

```
19  in{ resource[ "file:bibliography.xml" ],
      bib{{
21      book{{
          title[ var TITLE ],
23        desc var NAME
        }}
25    }}
    }
27 END
```

A type error is possible for the query rule as the inferred result type *book* is not included in the type *TextBook* (printout BIBLIOGRAPHY.3 in Section 4.4). We are not sure about type incorrectness of the rule as the inferred result type is not exact (due to a construct `all`). This time there is no weak type error discovered for the rule, which means that the rule may produce results of the specified result type. Thus the structure of the head of the rule conforms to the specified result type. The type inclusion check failure is due to the variables which get wrong values i.e. not of the types required by the result type specification. The variable NAME used in the body of the query rule can be bound to any data term which is a direct or an indirect subterm of *book*[...] (except a data term *title*[...]). Thus, the variable NAME may be mapped to the types: *Authors*, *Editors*, *Publisher*, etc. In the construct term the variable NAME is used to build content of cells of a table and according to the type specification it should be of a one of the types allowed for subterms of *inline*[...] which are *Text*, *Bf* and *Em*. A type error is likely as the union of the inferred types for the variable NAME is not included in the union of the types *Text*, *Bf* and *Em*.

The next rule is almost the same as the previous one. The difference is in the usage of the variable NAME in the body of the rule, which now can be bound only to the direct subterm of an element *last*[...].

```
1  CONSTRUCT
     book[
3      cover[ title [ "Books" ] ],
       body[
5        chapter[
           title[ "List_of_Books_and_Authors"],
7          section[
             table[
9              all tr[
                 td[ inline [ var TITLE ] ],
11               td[ inline [ var NAME ] ]
               ]
13           ]
           ]
15       ]
       ]
17   ]
   FROM
19   in{ resource[ "file:bibliography.xml" ],
       bib{{
21       book{{
           title[ var TITLE ],
23         desc last[ var NAME ]
         }}
25     }}
     }
27 END
```

The result of type checking performed by the type system for the rule is positive. Thus the rule is correct wrt. the specified result type *TextBook*. The corresponding typechecker printout is BIBLIOGRAPHY.4 in Section 4.4.

4.3 Usecase 3: Bookstore

Here we present an example of an Xcerpt program being one of the use cases for Xcerpt presented in [11]. As no result type specification is given for the program the type system is only able to perform type inference and check emptiness of the inferred types. This results in a specification of the inferred result type for the program. Such a type specification provided by the inference mechanism can be used for documentation purposes. Additionally, it can be used by a programmer to check manually if the inferred result type conforms to his/her expectations.

The use case is similar to one from *XQuery Use Cases* (*XMP-Q5* in [4]). The program queries two online bookstores and provides a summary over the prices for books in both book stores. The summary is given using two representations: HTML representation and a representation suitable for mobile devices, in the WML format (wireless markup language[5]). The program uses rule chaining to separate the query part from the presentation part and creates an intermediate representation for the data (in the example below: for each book, a *book-with-prices*[...] data term containing *title*[...], *price-a*[...] and *price-b*[...] subterms for the price in the first bookstore and the price in the second bookstore). This representation is then queried by the two rules that create HTML and WML representations.

The schemata defining the structure of databases for the two bookstores are given in [11] using the Relax NG notation and can be expressed by the following Type Definition.

$$
\begin{array}{lcl}
\mathit{Bib} & \rightarrow & \mathit{bib}\,[\,\mathit{Book}^*\,] \\
\mathit{Book} & \rightarrow & \mathit{book}\,[\,\mathit{Book\_attr}\ \mathit{Title}\ (\mathit{Authors} \mid \mathit{Editor})\ \mathit{Publisher}\ \mathit{Price}\,] \\
\mathit{Book\_attr} & \rightarrow & \mathit{attr}\,\{\,\mathit{Book\_year}\,\} \\
\mathit{Book\_year} & \rightarrow & \mathit{year}\,[\,\mathit{Text}\,] \\
\mathit{Title} & \rightarrow & \mathit{title}\,[\,\mathit{Text}\,] \\
\mathit{Authors} & \rightarrow & \mathit{authors}\,[\,\mathit{Author}^*\,] \\
\mathit{Author} & \rightarrow & \mathit{author}\,[\,\mathit{Last}\ \mathit{First}\,] \\
\mathit{Editor} & \rightarrow & \mathit{editor}\,[\,\mathit{Last}\ \mathit{First}\ \mathit{Affil}\,] \\
\mathit{Last} & \rightarrow & \mathit{last}\,[\,\mathit{Text}\,] \\
\mathit{First} & \rightarrow & \mathit{first}\,[\,\mathit{Text}\,] \\
\mathit{Affil} & \rightarrow & \mathit{affiliation}\,[\,\mathit{Text}\,] \\
\mathit{Publisher} & \rightarrow & \mathit{publisher}\,[\,\mathit{Text}\,] \\
\mathit{Price} & \rightarrow & \mathit{price}\,[\,\mathit{Text}\,] \\
\mathit{Reviews} & \rightarrow & \mathit{reviews}\,[\,\mathit{Entry}^*\,]
\end{array}
$$

$$
\begin{array}{lcl}
\mathit{Entry} & \rightarrow & \mathit{entry}\,[\,\mathit{Title}\ \mathit{Price}\ \mathit{Review}\,] \\
\mathit{Review} & \rightarrow & \mathit{review}\,[\,\mathit{Text}\,]
\end{array}
$$

The type of the document *bib.xml* is *Bib* and the type of the document *reviews.xml* is *Reviews*. This is the Xcerpt program:

```
GOAL
  out{
    resource[ "file:prices.html" , "html" ],
```

[5] http://www.wapforum.org/DTD/wml_1.1.xml

```
    html[
5     head[ title [ "Price Overview" ] ],
      body[
7       table[
          tr[ td[ "Title" ],
9         td[ "Price at A" ],
          td[ "Price at B" ] ],
11        all tr[ td[ var Title ],
          td[ var PriceA ],
13        td[ var PriceB ] ]
        ]
15    ]
    ]
17  }
  FROM
19  books-with-prices[[
      book-with-prices[[
21      title[[ var Title ]],
        price-a[[ var PriceA ]],
23      price-b[[ var PriceB ]]
    ]]
25  ]]
  END
27
  GOAL
29  out{
      resource[ "file:prices.wml" , "xml" ],
31    wml[
        all card[
33        "Title: " , var Title ,
          "Price A: " , var PriceA,
35        "Price B: " , var PriceB
        ]
37    ]
    }
39  FROM
    books-with-prices[[
41    book-with-prices[[
        title[[ var Title ]],
43      price-a[[ var PriceA ]],
        price-b[[ var PriceB ]]
45    ]]
    ]]
47  END

49  CONSTRUCT
    books-with-prices[
51    all book-with-prices[
        title[ var T ],
53      price-a[ var Pa ],
        price-b[ var Pb ]
55    ]
    ]
57  FROM
    and{
59    in{ resource [ "file:bib.xml" ],
        bib[[
61        book[[
            title[ var T ],
```

```
63          price[ var Pa ]
          ]]
65      ]]
      },
67    in{
        resource[ "file:reviews.xml" ],
69      reviews[[
          entry[[
71          title[ var T ],
            price[ var Pb ]
73        ]]
        ]]
75    }
  }
77 END
```

The type system infers results types of the rules. The inferred result type for the third query rule is *books-with-prices*. The inferred result types for the first and the second goal are respectively *html* and *wml*. These types are defined by the following Type Definition:

$$
\begin{array}{lcl}
\mathit{books\text{-}with\text{-}prices} & \rightarrow & \mathit{books\text{-}with\text{-}prices}\,[\,\mathit{book\text{-}with\text{-}prices}^{+}\,] \\
\mathit{book\text{-}with\text{-}prices} & \rightarrow & \mathit{book\text{-}with\text{-}prices}\,[\,\mathit{title}\ \mathit{price\text{-}a}\ \mathit{price\text{-}b}\,] \\
\mathit{price\text{-}a} & \rightarrow & \mathit{price\text{-}a}\,[\,\mathit{Text}\,] \\
\mathit{price\text{-}b} & \rightarrow & \mathit{price\text{-}b}\,[\,\mathit{Text}\,] \\
\mathit{title} & \rightarrow & \mathit{title}\,[\,\mathit{Text}\,]
\end{array}
$$

$$
\begin{array}{lcl}
\mathit{html} & \rightarrow & \mathit{html}\,[\,\mathit{head}\ \mathit{body}\,] \\
\mathit{head} & \rightarrow & \mathit{head}\,[\,\mathit{title}_1\,] \\
\mathit{title}_1 & \rightarrow & \mathit{title}\ [\,\mathit{Text}_1\,] \\
\mathit{Text}_1 & \rightarrow & \text{"}\mathit{Price\ Overview}\text{"}
\end{array}
$$

$$
\begin{array}{lcl}
\mathit{body} & \rightarrow & \mathit{body}\,[\,\mathit{table}\,] \\
\mathit{table} & \rightarrow & \mathit{table}\,[\,\mathit{tr}\ \mathit{tr}_1^{+}\,] \\
\mathit{tr} & \rightarrow & \mathit{tr}\,[\,\mathit{td}\ \mathit{td}_1\ \mathit{td}_2\,] \\
\mathit{td} & \rightarrow & \mathit{td}\,[\,\mathit{Text}_2\,] \\
\mathit{Text}_2 & \rightarrow & \text{"}\mathit{Title}\text{"} \\
\mathit{td}_1 & \rightarrow & \mathit{td}\,[\,\mathit{Text}_3\,] \\
\mathit{Text}_3 & \rightarrow & \text{"}\mathit{Price\ at\ A}\text{"} \\
\mathit{td}_2 & \rightarrow & \mathit{td}\,[\,\mathit{Text}_4\,] \\
\mathit{Text}_4 & \rightarrow & \text{"}\mathit{Price\ at\ B}\text{"} \\
\mathit{tr}_1 & \rightarrow & \mathit{tr}\,[\,\mathit{td}_3\ \mathit{td}_3\ \mathit{td}_3\,] \\
\mathit{td}_3 & \rightarrow & \mathit{td}\,[\,\mathit{Text}\,]
\end{array}
$$

$$
\begin{array}{lcl}
\mathit{wml} & \rightarrow & \mathit{wml}\,[\,\mathit{card}^{+}\,] \\
\mathit{card} & \rightarrow & \mathit{card}\ [\,\mathit{Text}_5\ \mathit{Text}\ \mathit{Text}_6\ \mathit{Text}\ \mathit{Text}_7\ \mathit{Text}\,] \\
\mathit{Text}_5 & \rightarrow & \text{"}\mathit{Title}:\ \text{"} \\
\mathit{Text}_6 & \rightarrow & \text{"}\mathit{PriceA}:\ \text{"} \\
\mathit{Text}_7 & \rightarrow & \text{"}\mathit{PriceB}:\ \text{"}
\end{array}
$$

The corresponding printout in Section 4.4 is BOOKSTORE. Since no result type specification is given the type system checks only if the inferred result type for each rule is not empty. If a

result type specification was given, the type system could check whether the data produced by the program conforms to HTML and WML formats.

4.4 Typechecker Results

This section presents printouts from the typechecker prototype. The printouts are results of typing the program examples from Section 4. The way how the obtained results should be interpreted is explained in Section 3.

CDstore.1

```
======================================================================
Rule 1: pop-entries
----------------------------------------------------------------------
TITLE->Artist, ARTIST->Artist
TITLE->Title, ARTIST->Artist
======================================================================
======================================================================
Type Definition:
----------------------------------------------------------------------
pop-entries -> pop-entries[ entry+ ]
entry -> entry[ Artist (Artist|Title)+ ]
Cds -> bib[ Cd* ]
Cd -> cd[ Title Artist+ Category? ]
Title -> title[ Text ]
Artist -> artist[ Text ]
Category -> "pop" | "rock" | "classic"
======================================================================
```

CDstore.2

```
======================================================================
Rule 1: pop-entries
Type checking: Unsuccessful (results not of type Entries possible)
----------------------------------------------------------------------
TITLE->Artist, ARTIST->Artist
TITLE->Title, ARTIST->Artist
======================================================================
======================================================================
Type Definition:
----------------------------------------------------------------------
pop-entries -> pop-entries[ entry+ ]
entry -> entry[ Artist (Artist|Title)+ ]
Cds -> bib[ Cd* ]
Cd -> cd[ Title Artist+ Category? ]
Title -> title[ Text ]
Artist -> artist[ Text ]
Category -> "pop" | "rock" | "classic"
Entries -> pop-entries[ Entry ]
```

```
Entry -> entry[ Artist Title+ ]
======================================================================
```

BIBLIOGRAPHY.1

```
======================================================================
Rule 1: 0
----------------------------------------------------------------------
0
======================================================================
======================================================================
Type Definition:
----------------------------------------------------------------------
TextBook -> book[ Cover Body ]
Cover -> cover[ Title Author* Publisher? ]
Body -> body[ Abstract? Chapter* ]
Title -> title[ InlineContent ]
Author -> author[ Text ]
Publisher -> publisher[ Text ]
Abstract -> abstract[ Text ]
Chapter -> chapter[ Title Section* ]
InlineContent -> inline[ Text|Bf|Em ]
Section -> section[ Title (Paragraph|Table|List)* ]
Em -> em[ InlineContent ]
Bf -> bf[ InlineContent ]
Paragraph -> p[ InlineContent* ]
Table -> table[ TableRow+ ]
List -> list[ ListItem ]
TableRow -> tr[ TableCell* ]
ListItem -> item[ InlineContent* ]
TableCell -> td[ InlineContent* ]
Bibliography -> bib[ (Book|Article|InProceedings)* ]
Book -> book{ Publisher? Editors Authors Title1 }
Article -> article{ Journal? Authors Title1 }
InProceedings -> inproc{ Book Authors Title1 }
Title1 -> title[ Text ]
Authors -> authors[ Person* ]
Editors -> editors[ Person* ]
Journal -> journal{ Editors Title1 }
Person -> person[ FirstName LastName ]
FirstName -> first[ Text ]
LastName -> last[ Text ]
======================================================================
```

BIBLIOGRAPHY.2

```
======================================================================
Rule 1: book
Type checking: Failed (no results of type TextBook)
```

```
----------------------------------------------------------------------
* -> Top
======================================================================
======================================================================
Type Definition:
----------------------------------------------------------------------
book -> book[ cover body ]
body -> body[ table ]
table -> table[ tr+ ]
tr -> tr[ td td_1 ]
td_1 -> td[ em+ ]
em -> em[ Top Top ]
td -> td[ Top ]
cover -> cover[ title ]
title -> title[ Text_1 ]
Text_1 -> "List_of_Books"
======================================================================
```

BIBLIOGRAPHY.3

```
======================================================================
Rule 1: book
Type checking: Unsuccessful (results not of type TextBook possible)
----------------------------------------------------------------------
TITLE->Text, NAME->Publisher
TITLE->Text, NAME->Text
TITLE->Text, NAME->Editors
TITLE->Text, NAME->Person
TITLE->Text, NAME->FirstName
TITLE->Text, NAME->LastName
TITLE->Text, NAME->Authors
======================================================================
======================================================================
Type Definition:
----------------------------------------------------------------------
book -> book[ cover body ]
body -> body[ chapter ]
chapter -> chapter[ title_1 section ]
section -> section[ table ]
table -> table[ (tr|tr_1|tr_2|tr_3|tr_4|tr_5|tr_6)+ ]
tr_6 -> tr[ td td_13 ]
td_13 -> td[ inline_13 ]
inline_13 -> inline[ Authors ]
tr_5 -> tr[ td td_11 ]
td_11 -> td[ inline_11 ]
inline_11 -> inline[ LastName ]
tr_4 -> tr[ td td_9 ]
td_9 -> td[ inline_9 ]
inline_9 -> inline[ FirstName ]
tr_3 -> tr[ td td_7 ]
```

```
td_7 -> td[ inline_7 ]
inline_7 -> inline[ Person ]
tr_2 -> tr[ td td_5 ]
td_5 -> td[ inline_5 ]
inline_5 -> inline[ Editors ]
tr_1 -> tr[ td td ]
tr -> tr[ td td_1 ]
td_1 -> td[ inline_1 ]
inline_1 -> inline[ Publisher ]
td -> td[ inline ]
inline -> inline[ Text ]
title_1 -> title[ Text_2 ]
Text_2 -> "List_of_Books_and_Authors"
cover -> cover[ title ]
title -> title[ Text_1 ]
Text_1 -> "Books"
TextBook -> book[ Cover Body ]
Body -> body[ Abstract? Chapter* ]
Chapter -> chapter[ Title Section* ]
InlineContent -> inline[ Text|Bf|Em ]
Section -> section[ Title? (Paragraph|Table|List)* ]
Em -> em[ InlineContent ]
Bf -> bf[ InlineContent ]
Paragraph -> p[ InlineContent* ]
Table -> table[ TableRow+ ]
List -> list[ ListItem ]
TableRow -> tr[ TableCell* ]
ListItem -> item[ InlineContent* ]
TableCell -> td[ InlineContent* ]
TextBook_1 -> book[ Cover Body_1 ]
Cover -> cover[ Title Author* Publisher? ]
Body_1 -> body[ Abstract? Chapter_1* ]
Author -> author[ Text ]
Publisher -> publisher[ Text ]
Abstract -> abstract[ Text ]
Chapter_1 -> chapter[ Title Section_1* ]
InlineContent_1 -> inline[ Text|Bf_1|Em_1 ]
Section_1 -> section[ Title? (Paragraph_1|Table_1|List_1)* ]
Em_1 -> em[ InlineContent_1 ]
Bf_1 -> bf[ InlineContent_1 ]
Paragraph_1 -> p[ InlineContent_1* ]
Table_1 -> table[ TableRow_1+ ]
List_1 -> list[ ListItem_1 ]
TableRow_1 -> tr[ TableCell_1* ]
ListItem_1 -> item[ InlineContent_1* ]
TableCell_1 -> td[ InlineContent_1* ]
Bibliography -> bib[ (Book|Article|InProceedings)* ]
Book -> book{ Title Authors Editors Publisher? }
Article -> article{ Title Authors Journal? }
InProceedings -> inproc{ Title Authors Book }
Title -> title[ Text ]
```

```
Authors -> authors[ Person* ]
Editors -> editors[ Person* ]
Journal -> journal{ Title Editors }
Person -> person[ FirstName LastName ]
FirstName -> first[ Text ]
LastName -> last[ Text ]
=====================================================================
```

BIBLIOGRAPHY.4

```
=====================================================================
Rule 1: book
Type checking: OK
---------------------------------------------------------------------
TITLE->Text, NAME->Text
=====================================================================
=====================================================================
Type Definition:
---------------------------------------------------------------------
book -> book[ cover body ]
body -> body[ chapter ]
chapter -> chapter[ title_1 section ]
section -> section[ table ]
table -> table[ tr+ ]
tr -> tr[ td td ]
td -> td[ inline ]
inline -> inline[ Text ]
title_1 -> title[ Text_2 ]
Text_2 -> "List_of_Books_and_Authors"
cover -> cover[ title ]
title -> title[ Text_1 ]
Text_1 -> "Books"
TextBook -> book[ Cover Body ]
Body -> body[ Abstract? Chapter* ]
Chapter -> chapter[ Title Section* ]
InlineContent -> inline[ Text|Bf|Em ]
Section -> section[ Title? (Paragraph|Table|List)* ]
Em -> em[ InlineContent ]
Bf -> bf[ InlineContent ]
Paragraph -> p[ InlineContent* ]
Table -> table[ TableRow+ ]
List -> list[ ListItem ]
TableRow -> tr[ TableCell* ]
ListItem -> item[ InlineContent* ]
TableCell -> td[ InlineContent* ]
TextBook_1 -> book[ Cover Body_1 ]
Cover -> cover[ Title Author* Publisher? ]
Body_1 -> body[ Abstract? Chapter_1* ]
Author -> author[ Text ]
Publisher -> publisher[ Text ]
```

```
Abstract -> abstract[ Text ]
Chapter_1 -> chapter[ Title Section_1* ]
InlineContent_1 -> inline[ Text|Bf_1|Em_1 ]
Section_1 -> section[ Title? (Paragraph_1|Table_1|List_1)* ]
Em_1 -> em[ InlineContent_1 ]
Bf_1 -> bf[ InlineContent_1 ]
Paragraph_1 -> p[ InlineContent_1* ]
Table_1 -> table[ TableRow_1+ ]
List_1 -> list[ ListItem_1 ]
TableRow_1 -> tr[ TableCell_1* ]
ListItem_1 -> item[ InlineContent_1* ]
TableCell_1 -> td[ InlineContent_1* ]
Bibliography -> bib[ (Book|Article|InProceedings)* ]
Book -> book{ Title Authors Editors Publisher? }
Article -> article{ Title Authors Journal? }
InProceedings -> inproc{ Title Authors Book }
Title -> title[ Text ]
Authors -> authors[ Person* ]
Editors -> editors[ Person* ]
Journal -> journal{ Title Editors }
Person -> person[ FirstName LastName ]
FirstName -> first[ Text ]
LastName -> last[ Text ]
======================================================================
```

Bookstore

```
======================================================================
Rule 1: html
----------------------------------------------------------------------
Title->Text, PriceA->Text, PriceB->Text

======================================================================
Rule 2: wml
----------------------------------------------------------------------
Title->Text, PriceA->Text, PriceB->Text

======================================================================
Rule 3: books-with-prices
----------------------------------------------------------------------
T->Text, Pa->Text, Pb->Text

======================================================================
======================================================================
Type Definition:
----------------------------------------------------------------------
wml -> wml[ card+ ]
card -> card[ Text_5 Text Text_6 Text Text_7 Text ]
Text_7 -> "Price B:"
Text_6 -> "Price A:"
```

```
Text_5 -> "Title:"
html -> html[ head body ]
body -> body[ table ]
table -> table[ tr tr_1+ ]
tr_1 -> tr[ td_3 td_3 td_3 ]
td_3 -> td[ Text ]
tr -> tr[ td td_1 td_2 ]
td_2 -> td[ Text_4 ]
Text_4 -> "Price at B"
td_1 -> td[ Text_3 ]
Text_3 -> "Price at A"
td -> td[ Text_2 ]
Text_2 -> "Title"
head -> head[ title_1 ]
title_1 -> title[ Text_1 ]
Text_1 -> "Price Overview"
books-with-prices -> books-with-prices[ book-with-prices+ ]
book-with-prices -> book-with-prices[ title price-a price-b ]
price-b -> price-b[ Text ]
price-a -> price-a[ Text ]
Bib -> bib[ Book* ]
Book -> book[ Book_attr title (Authors|Editor) Publisher Price ]
Book_attr -> attr{ Book_year }
Book_year -> year[ Text ]
title -> title[ Text ]
Authors -> authors[ Author* ]
Author -> author[ Last First ]
Editor -> editor[ Last First Affil ]
Last -> last[ Text ]
First -> first[ Text ]
Affil -> affiliation[ Text ]
Publisher -> publisher[ Text ]
Price -> price[ Text ]
Reviews -> reviews[ Entry* ]
Entry -> entry[ title Price Review ]
Review -> review[ Text ]
=====================================================================
```

References

[1] U. Aßmann, S. Berger, F. Bry, T. Furche, J. Henriksson, and J. Johannes. Modular web queries—from rules to stores. *3rd International Workshop On Scalable Semantic Web Knowledge Base Systems (SSWS'07) (to appear). Vilamoura, Algarve, Portugal, Nov 27, 2007*, 2007.

[2] 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.

[3] F. Bry and S. Schaffert. A gentle introduction into xcerpt, a rule-based query and transformation language for xml. In *Proceedings of International Workshop on Rule Markup Languages for Business Rules on the Semantic Web, Sardinia, Italy (14th June 2002)*, 2002.

[4] D. Chamberlin et al. XML Query Use Cases. `http://www.w3.org/TR/xquery-use-cases/`, March 2007.

[5] Francois Bry and Sebastian Schaffert. The XML Query Language Xcerpt: Design Principles, Examples, and Semantics. In *Revised Papers from the NODe 2002 Web and Database-Related Workshops on Web, Web-Services, and Database Systems*, pages 295–310, London, UK, 2003. Springer-Verlag.

[6] J. Henriksson, F. Heidenreich, J. Johannes, S. Zschaler, and U. Aßmann. Extending grammars and metamodels for reuse – the reuseware approach. *To Appear in IET Software, Special Issue on Language Engineering*, 2007.

[7] J. Henriksson, J. Johannes, S. Zschaler, and U. Aßmann. Reuseware – adding modularity to your language of choice. *Proc. of TOOLS EUROPE 2007: Special Issue of the Journal of Object Technology (to appear)*, 2007.

[8] M. Horridge, N. Drummond, J. Goodwin, A. Rector, R. Stevens, and H. Wang. The manchester owl syntax. *OWL: Experiences and Directions (OWLED)*, November 2006.

[9] P. F. Patel-Schneider, P. Hayes, and I. Horrocks. OWL web ontology language semantics and abstract syntax. W3C Recommendation, 10 February 2004. Available at `http://www.w3.org/TR/owl-semantics/`.

[10] M. Pradel, J. Henriksson, and U. Aßmann. A good role model for ontologies: Collaborations. *International Workshop on Semantic-Based Software Development. Co-located with OOPSLA'07, Montreal, Canada, Oct 22, 2007*, 2007.

[11] S. Schaffert. *Xcerpt: A Rule-Based Query and Transformation Language for the Web*. Dissertation/Ph.D. thesis, Institute of Computer Science, LMU, Munich, 2004. PhD Thesis, Institute for Informatics, University of Munich, 2004.