A Brief Introduction to XMAS

XMAS sub-group of MIX*

February 9, 1999

1 Introduction

Background. Mediation of information from heterogeneous sources is a crucial task for future Web information systems. The MIX[1] project at SDSC/UCSD relies on the well-known mediator architecture [Wie92] to provide the user with an integrated view of the underlying sources. To facilitate a uniform and flexible representation of arbitrary source data, MIX employs XML[XML98b]. XML can be used for marking up and structuring data by means of semantically meaningful tags.

The novel features of the MIX architecture include:

- Data exchange and integration solely relies on XML, i.e., instance and schema information is represented by XML documents and XML DTDs, respectively.
- XML queries are denoted in a high-level, declarative query language XMAS[2], which builds upon ideas of languages like XML-QL, Yat, MSL, and UnQL [XML98a, CDSS98, PAGM96, BDFS97]. For example, XMAS allows object fusion and pattern matching on the input XML data. Additionally, XMAS features powerful grouping and order constructs for generating new integrated XML "objects" from existing ones.
- The graphical user interface BBQ (***B**lended **B**rowsing and **Q**uerying*) is completely driven by the mediator view DTD and integrates browsing and querying of XML data. Complex queries can be constructed in an intuitive way, which resembles QBE. Due to the nested nature of XML data and DTDs, BBQ employs a novel graphical way to specify the nesting and grouping of query results.

MIX Architecture. The main architecture of MIX is depicted in Fig. 1: The *graphical user interface* BBQ allows the construction of queries in an intuitive way. BBQ is driven by the XML DTDs of the mediator view and guides the user in formulating complex queries. A design goal of BBQ is to provide a seamless blend of browsing and querying modes, in the spirit of GARLIC's PESTO interface [CHMW96].

The MIX *mediator* comprises several modules to accomplish the integration; its main inputs are XMAS queries generated by BBQ, and the *mediator view definition* (also in XMAS) for the integrated view. The latter has to be provided by the "mediation engineer", and prescribe how the integrated data combines the wrapper views. The *resolution* module resolves the user query with the mediator view definition, resulting in a set of unfolded XML queries that refer to the wrapper views. These queries can be further simplified based on the underlying XML DTDs.

The DTD inference module can be used to automatically derive view DTDs from source DTDs and view definitions, thereby supporting the integration task of the mediation engineer.[3] The

*Bertram Ludäscher, Yannis Papakonstantinou, Pavel Velikhov; ludaesch@sdsc.edu, yannis@cs.ucsd.edu, pvelikho@cs.ucsd.edu

[1] ***M**ediation of **I**nformation using **X**ML*

[2] ***XML** **M**atching **A**nd **S**tructuring Language*

[3] In MIX, DTD inference is performed in a separate off-line step, i.e., before the user interacts with the mediator.

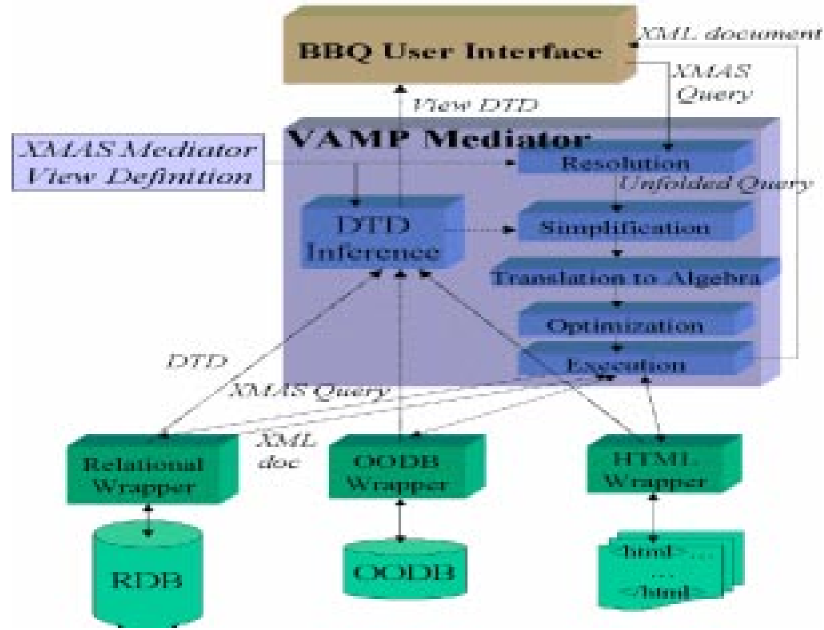

Figure 1: MIX architecture

translation module maps the simplified queries into the XMAS algebra which can then be further optimized. Finally, the execution engine issues XMAS queries against the wrappers, and returns the requested XML data to the user, after integrating the retrieved data according to the mediator view.

Wrappers are used on top of the heterogeneous sources to export data in a uniform format to the mediator. In contrast to previous approaches, which rely on "schema-less" semistructured models, we use XML as our model for semistructured data and exploit the structuring information provided by XML DTDs. Depending on the source's capabilities, a wrapper may support only certain types of XML queries, in which case it is the meditor's responsibility to issue only such queries.

Similar to TSIMMIS, MIX's query evaluation corresponds to a *lazy* approach (also called *on demand* or *virtual*), i.e., XML queries (expressed in XMAS) are unfolded and rewritten at runtime as they flow downwards from the user to the sources. In contrast, for example STRUDEL [FFK+97] and FLORID [LHL+98] follow the *eager* (or *warehousing*) approach, where data integration occurs in a separate materialization step, *before* the actual user queries.

2 XMAS by Example

XMAS is a declarative rule-based query language for XML. XML can be seen as an instance of the semistructured data model. Thus it can hold a wide variety of data, i.e., data with few or irregular structure (e.g., wrapped HMTL pages) but also highly structured data like data from relational databases. In the following examples, we consider the latter to illustrate the basic features of querying XML with XMAS.

Consider, the following neighborhoods relation:

neighborhoods			
zip	*name*	*type*	*population*
91901	Alpine	Rural/Town	13238
91903	Alpine	Rural/Town	4783
91902	Bonita	Urban/Suburban	18120
92003	Bonsall	Rural/Town/Satellite City	5897
92004	Borrego Springs	Rural/Satellite City	2377
91905	Boulevard	Rural	1018
...	...	...	...

We can represent such a relation directly in XML as follows: The neighborhoods relation is represented by a `neighborhoods` element. For each tuple of the relation we create a `neighborhood` subelement. Thus, `neighborhoods` can hold an arbitrary number of `neighborhood` subelements. Finally, we represent each attribute (*zip*, *name*, ...) as a subelement of `neighborhood` with the corresponding name.[4] This element structure can be described in the following form as an XML DTD:

```
<!ELEMENT neighborhoods        (neighborhood)*>
<!ELEMENT neighborhood         (zip, name, type, population)>
<!ELEMENT zip                  (#PCDATA)>
<!ELEMENT name                 (#PCDATA)>
<!ELEMENT type                 (#PCDATA)>
<!ELEMENT population           (#PCDATA)>
```

Observe that the declaration of the element name is followed by a regular expression specifying the element content. The actual relational data can now be represented in XML as follows:

```
<neighborhoods>
    <neighborhood>
        <zip>91901</zip>
        <name>Alpine</name>
        <type>Rural/Town</type>
        <population>13238</population>
     </neighborhood>
     <neighborhood>
        <zip>91903</zip>
        <name>Alpine</name>
        <type>Rural/Town</type>
        <population>4783</population>
     </neighborhood>
     <neighborhood>
        <zip>91902</zip>
        <name>Bonita</name>
        <type>Urban/Suburban</type>
        <population>18120</population>
     </neighborhood>
        ...
</neighborhoods>
```

How can we query such an XML database with XMAS? The basic idea is to use XMAS rules which are of the form

`CONSTRUCT` *head* `WHERE` *body*

The *body* (`WHERE`-clause) specifies the data which is to be extracted from the XML sources under consideration, the *head* (`CONSTRUCT`-clause) describes how the extracted data is arranged into a new answer XML document. These clauses roughly correspond to SQL's `WHERE` and `SELECT` clauses, respectively. However, rather than defining relational views like SQL or Datalog, XMAS rules define *XML views* on the underlying (XML) data.

For example, assume we want to retrieve all names of "big" neighborhoods, say where the population is greater than 30,000. In XMAS we can write the following:

[4] Alternatively, we could make them attributes of the parent element.

```
CONSTRUCT
<big_neighborhoods>
  <big_neighborhood>
      <name>$N</>
  </> {$N}
</>
WHERE
  <neighborhoods>
    <neighborhood>
      <name>$N</>
      <population>$P</>
    </>
  </>
IN "http://www.npaci.edu/DICE/MIX/tutorial/neighborhoods.xml"
AND $P>30000.
```

Body. Consider first the rule body, i.e., the `WHERE`-clause: Observe how the *tree pattern* mimics the tree structure of the input XML document, and how variables `$N` and `$P` are used to "get a hold" of the data at the corresponding locations in the tree. More precisely, the tree pattern specifies that the root element of the XML document (at `http://...`) has to be of type `neighborhoods`. Within `neighborhoods` there must be some `neighborhood` subelement, which itself contains `name` and `population` subelements.[5] In this way, the tree pattern specifies a list of pairs of variable bindings for `$N` and `$P`. From this list, we want to select only those which satisfy the condition `$P>30000`. Summarizing, the body defines a list $[(n_1, p_1), \ldots, (n_k, p_k)]$ of all variable bindings for (`$N`,`$P`), which match (or *satisfy*) the body.

Head. In the rule head (`CONSTRUCT`-clause) we specify how to construct an answer XML element from this list. The head consists of an XML tree pattern which contains some or all of the variables from the body. Ignoring the *collection label* {`$N`} for the moment, we see that the head defines a root element `big_neighborhoods` with a `big_neighborhood` subelement, having in turn a `name` subelement. The latter is used to hold the bindings for `$N` which have been obtained through the body.

Now, if we omit the collection label {`$N`}, we obtain *one* `big_neighborhoods` element *for each binding* $n_1, \ldots, n_k$. This answer structure is not intended[6]. Instead, we want to have only one root element `big_neighborhoods` which has a number of subelements `big_neighborhood`, one for each name `$N` obtained from the body. This is accomplished using the *collection label* {`$N`} which follows the `big_neighborhood` element (and thus applies to this element). This collection label specifies that

(i) we create exactly one `big_neighborhood` element for each binding $n_1, \ldots, n_k$ of `$N` (thereby fixing the value of `$N` within the `big_neighborhood` element to one n_i), and

(ii) that all these elements are collected as subelements of the parent element (here: `big_neighborhoods`).

For elements in the head which do not have an explicit collection label, there is an implicit collection label. Roughly speaking, the implicit collection variables of an element E are those which are *free* in E.[7] In our case, the collection label for `big_neighborhoods` is {} since there is no free variables within this element (the `$N` gets bound by the collection label {`$N`}). The collection label for the `name` element is {`$N`} since `$N` occurs free within the `name` element. However, note that this label will not cause `$N` to range over different bindings, since `$N` is already bound from above.

[5] Note that we allow abbreviated end-tags of the form `</>`.

[6] Indeed, it is not well-formed XML since there must be a unique root element.

[7] A variable is free in an element, if it is not bound by a collection label; see the definitions below for a detailed explanation on how to determine these implicit labels.

Expressing Relational Algebra Operations. It is instructive to see how the usual relational algebra operators (σ, π, $\bowtie$, ...) can be expressed in XMAS. Note that the XMAS rule above performs a *selection* based on `$P` (in the body) followed by a *projection* on the name and a *renaming*[8] (in the head). Thus, neglecting the renaming and the tree structure of the output, the rule essentially computes $\pi_{name}(\sigma_{population>30000}(neighborhoods))$.

A *join* can be obtained by equating variables in the body: Assume, e.g., that we have an XML source with information about schools, including their zip codes. We can integrate the information from these sources by joining on the zip code:

```
CONSTRUCT
<neighborhoods_med>
  <neighborhood_med>
    $N
    $S
  </> {$N, $S}
</>
WHERE
  <neighborhoods>
    $N: <neighborhood>
          <zip>$Z</>
        </>
  </>
IN "http://www.npaci.edu/DICE/MIX/tutorial/neighborhoods.xml"
AND
  <schools>
    $S: <school>
          <zip>$Z1</>
        </>
  </>
IN "http://www.npaci.edu/DICE/MIX/tutorial/schools.xml"
AND  $Z=$Z1
```

Here we have defined an integrated (or "mediated") view `neighborhoods_med` where each subelement `neighborhood_med` contains one neighborhood `$N` and one school `$S` such that `$N.zip=$S.zip`. Note how the collection label is used to collect `neighborhood_med` subelements as children of the root element.

What happens if we omit the equation `$Z=$Z1`? Then the result constructed in the head corresponds to $neighborhoods \times schools$, i.e., a cartesian product just like in the relational case.

Finally, note the different occurrences of variables in the body: `$Z` and `$Z1` occur in the content position of a `zip` element. Hence, these *content variables* are bound to the character content of the `zip` element.[9] In contrast, `$N` and `$S` are *element variables* since they precede (separated by a colon "`:`") a complete element. In this way we can pick complete elements from the source without knowing their precise structure (indeed their structure may vary for different picked elements). The latter is very useful when querying semistructured data, i.e., XML documents having a DTD that is less rigid than a "relational" DTD, or documents for which a DTD is not given: For example, we can obtain the complete document tree `$D` of *any* XML source and the name `$N` of its root element as follows:[10]

```
CONSTRUCT
<answer>
  <document>$D</>
```

[8]Here, `neighborhood` becomes `big_neighborhood`, etc. Since XMAS allows variables at tag positions, one can express data-driven renaming and schema update.

[9]If `zip` had subelements, these variables would range over those subelements (in the case of mixed content: subelements *and* character contents).

[10]A more general construct for querying semistructured data are *regular path expressions*, which are not discussed here.

```
  <root_name>$N</>
</>
WHERE
  $D: <$N/> IN ...
```

2.1 XMAS Semantics

Let us take a closer look at the sematics of a XMAS rule

CONSTRUCT *head* WHERE *body* .

Assume that the body contains the variables $\bar{X} = X_1, \ldots, X_n$ (for notational convenience, we denote variables by uppercase letters and omit the "$" symbol). Thus, evaluation of the body yields a list of *variable bindings* $Bs = [\bar{x}_1, \ldots, \bar{x}_k]$ where each $\bar{x}_i = (x_{i,1}, \ldots x_{i,n})$ defines a value for all the variables $\bar{X}$. This list is obtained by matching the tree patterns in the body with the corresponding input XML documents and applying the usual operations (AND, OR, =, etc.); see Section B.1 for details.

The more involved construction is the semantics in the head. Assume for the moment that all elements in the head have an attached collection label. The semantics $sem(C, Bs, \beta)$ of a XMAS expression C in the head is given wrt. the variable bindings Bs (obtained from the body) and a variable binding β for the currently *fixed variables* (initially, β is empty). Using these, we define the semantics of *head* to be[11]

$$sem(head, Bs, \emptyset)$$

Depending on the expression C, we define $sem(C, Bs, \beta)$ recursively as follows:

- if $C = s$ (i.e., we are looking at a string constant) then we return a list containing that string:

$$sem(C, Bs, \beta) := [\, s \,]$$

- if $C = X_i$ (i.e., we are looking at a variable from the body) then we return a list of all bindings for X_i in Bs which satisfy the currently fixed variables β, i.e.,

$$sem(C, Bs, \beta) := [\, x_i \mid \bar{x} \in \sigma_\beta(Bs) \,]$$

- if $C = \langle p\ \boldsymbol{A} \rangle\, C'\, \langle / \rangle\ \{\bar{Y}\}$ (i.e., we are looking at an element pattern with collection label $\{\bar{Y}\}$) then we return a list as follows:

  First consider $\pi_{\bar{Y}}(\sigma_\beta(Bs))$ which is the list of variable bindings for Bs which satisfy β, projected on the collection variables $\bar{Y}$. This yields a list of bindings describing the range of the variables $\bar{Y}$. For *each* such binding ι we construct *one* element $\langle \iota(\beta(p\ \boldsymbol{A}))/\rangle$. The contents of this element is defined by applying *sem* recursively, but now with the additional fixed variables ι. Finally, we return the list of all these elements which thereby become children of the enclosing parent element. Thus,

$$sem(C, Bs, \beta) := [\, \langle \iota \circ \beta(p\ \boldsymbol{A}) \rangle\ sem(C', Bs, \iota \circ \beta)\ \langle / \rangle \mid \iota \in \pi_{\bar{Y}}(\sigma_\beta(Bs)) \,]$$

- if $C = f(C_1, \ldots, C_n)$ (i.e., we are looking at some function on lists like ORDERBY), then we just return the result of that function applied to the *sem* of the arguments:

$$sem(C, Bs, \beta) := f(sem(C_1, Bs, \beta), \ldots, sem(C_n, Bs, \beta))$$

[11] Strictly speaking, for *head*, the function *sem* has to return a one-element list; otherwise the query result is not an XML document and hence undefined.

Implicit Collection Labels. Recall the effect of a collection label $\{\bar{Y}\}$ when applied to an element E. For each binding of $\bar{y}_i$ of $\{\bar{Y}\}$ it creates an element E_i; all of these become children of the enclosing parent element. There are two shortcuts which allow to omit the explicit mentioning of a collection label:

First, we may simply omit the collection label. This indicates that we do not want to fix the bindings of variables at this level in the answer tree.

Second, we can use brackets around the E to indicate that we want to make a collection at this level of the tree: the collection label corresponding to the bracketed expression $[E]$ comprises all free variables in E. To illustrate this, consider the following XMAS head:

```
<ans>
  <a>
    $A
    <b>
      $B
      <c> $C </c> {$C}
    </b> {$B}
  </a> {$A}
</ans>
```

This creates one `<ans/>` element which has for each value of `$A` a subelement `<a/>`, each of which has the value of `$A` as the first child, followed by a list of `<b/>` elements, one for each value of `$B` (given the already fixed value of `$A`); similarly for `$C`. In a sense, this corresponds to a nested loop structure, with the outermost loop ranging over the values of {`$A`}, the innermost loop ranging over {`$C`}, and the {`$B`} loop inbetween. With the implicit collection labels defined by brackets, we can write this more intuitively as follows:

```
<ans>
  [<a>
    $A
    [<b>
      $B
      [<c> $C </c>]
    </b>]
  </a>]
</ans>
```

Now assume we drop the brackets around the `<b/>` element. Then we create for each pair of values for `$A` and `$B` one `<a/>` element (since both `$A` and `$B` are free within `<a/>`), which corresponds to the explicit collection label {`$A,$B`} for the `<a/>` element (the implicit label {`$C`} for `<c/>` is not changed).

Formally, we can rewrite any head expression with implicit collection label into an equivalent one with explit collection label; see Appendix B.2.

References

[BDFS97] P. Buneman, S. B. Davidson, M. F. Fernandez, and D. Suciu. Adding Structure to Unstructured Data. In F. Afrati and P. Kolaitis, editors, *6th Intl. Conference on Database Theory (ICDT)*, number 1186 in LNCS, pp. 336–350, Delphi, Greece, 1997. Springer.

[CDSS98] S. Cluet, C. Delobel, J. Simeon, and K. Smaga. Your Mediators Need Data Conversion! In *Proceedings of the 1998 ACM SIGMOD International Conference on Management of Data*, pp. 177–188, 1998.

[CHMW96] M. J. Carey, L. M. Haas, V. Maganty, and J. H. Williams. PESTO: An Integrated Query/Browser for Object Databases. In *Intl. Conference on Very Large Data Bases (VLDB)*, pp. 203–214, 1996.

[FFK+97] M. F. Fernandez, D. Florescu, J. Kang, A. Y. Levy, and D. Suciu. STRUDEL: A Web-site Management System. In *ACM Intl. Conference on Management of Data (SIGMOD)*, pp. 549–552, 1997.

[LHL+98] B. Ludäscher, R. Himmeröder, G. Lausen, W. May, and C. Schlepphorst. Managing Semistructured Data with FLORID: A Deductive Object-Oriented Perspective. *Information Systems*, 23(8), 1998.

[PAGM96] Y. Papakonstantinou, S. Abiteboul, and H. Garcia-Molina. Object Fusion in Mediator Systems. In *Intl. Conference on Very Large Data Bases (VLDB)*, pp. 413–424, 1996.

[Wie92] G. Wiederhold. Mediators in the Architecture of Future Information Systems. *IEEE Computer*, 25(3):38–49, 1992.

[XML98a] XML-QL: A Query Language for XML. W3C note, `http://www.w3.org/TR/NOTE-xml-ql`, 1998.

[XML98b] Extensible Markup Language (XML) 1.0. W3C recommendation, `http://www.w3.org/TR/REC-xml`, 1998.

A XMAS Grammar

```
rule             ::= "CONSTRUCT" head "WHERE" body

head             ::= startTag collection endTag
collection       ::= head collection_label                                 % explicit listify
                   | head                                                  % implicit listify
                   | "[" head "]"                                          % implicit listify
                   | identifier "(" collection ("," collection)? ")"      % function application
                   | variable
collection_label ::= "{" variable* "}"

body             ::= body "AND" body
                   | body "OR" body
                   | "NOT" body
                   | "(" body ")"
                   | bodyPattern
bodyPattern      ::= bodyContent "IN" source
                   | predicate
                   | assignment
bodyContent      ::= (variable ":")? bodyElement                          % ⟦variable⟧ ⊆ elem
                   | variable                                             % ⟦variable⟧ ⊆ elem ∪ str (*)
bodyElement      ::= startTag bodyContent* endTag
                   | emptyElement
startTag         ::= "<" term attributes ">"                              % ⟦term⟧ ⊆ tag
endTag           ::= "</" (identifier)? ">"
emptyElement     ::= "<" term attributes "/>"                             % ⟦term⟧ ⊆ tag
attributes       ::= (term1 "=" term2)*                                   % ⟦term1⟧ ⊆ att, ⟦term2⟧ ⊆ str
term             ::= variable | constant
variable         ::= "$" identifier
```

Remark. To avoid ambiguities between the lazy (i.e., empty) *endTag* $\langle/\rangle$ and an unrestricted *bodyElement*, we have to write the latter as $\langle\rangle\langle/\rangle$ (indeed, this is enforced by the rule for *emptyElement*).

B XMAS Semantics (Continued)

Remarks.

- The XMAS variable prefix "$" is dropped.
- For simplicity, we assume that element names, attribute names, and character data come from the same data domain **D**. Then **E** is the set of all elements (=XML trees) over **D**.
- Variable bindings can be extended to XMAS expressions (especially, the XMAS body) in the obvious way.

- Variable bindings can be conceived as sets of variable/value pairs, i.e., $\beta = \{X_1{=}x_1, \ldots, X_k{=}x_k\}$. Thus we can use them as selection conditions (see below).

B.1 Body

Definition 1 (Variable Binding)
A *variable binding* β over a sequence of variables $\bar{X} = X_1, \ldots, X_n$ is a mapping β which assigns to every variable $X \in \bar{X}$ either an element $e \in \mathbf{E}$ or a data value $d \in \mathbf{D}$. $\square$

Definition 2 (Binding List)
Given a sequence of variables $\bar{X} = X_1, \ldots, X_n$, a *binding list* over $\bar{X}$ is a list $[\beta_1, \ldots, \beta_k]$ s.t. each β_i is a variable binding over $\bar{X}$. $\square$

Remarks. Below, for convenience of notation, we blur the difference between a variable binding and a tuple. Indeed, given a binding list Bs over a fixed sequence $\bar{X}$ of variables, each variable binding defines a tuple and vice versa.

Definition 3 (Satisfaction)
Let $e = \langle p\ \boldsymbol{A}\rangle\, C\, \langle /p\rangle$ be an element from $\mathbf{E}$, β a variable binding, and B a XMAS body. Then $e, \beta \models B$ holds ...

- for $B = (B_1\ \texttt{AND}\ B_2)$ $:\Leftrightarrow$ $e, \beta \models B_1$ and $e, \beta \models B_2$ *(similar for `OR`, `NOT`)*
- for $B = (x : \langle p'\rangle \boldsymbol{A}'\, C'\, \langle / \rangle)$ $:\Leftrightarrow$
  - $\beta(x) = e$,
  - $\beta(p') = p$,
  - f.a. $(a'{=}v') \in \boldsymbol{A}'$ ex. $(a{=}v) \in \boldsymbol{A}$ s.t. $\beta(a'){=}a$ and $\beta(v'){=}v$, and
  - f.a. $c' \in C'$ ex. $c \in C$ s.t. $c, \beta \models c'$
- for $B = B'\ \texttt{IN}\ \mathit{source}$ $:\Leftrightarrow$
  - $e' = \mathit{root_element}(\mathit{source})$,
  - $e', \beta \models B'$ $\square$

Order Among Sibling Elements. Observe that this definition implies that the order among sibling elements in the body is irrelevant. Thus, XMAS relieves the rule programmer from remembering the specific order in the input. However, the order in which elements occur in the *head* is significant, since it is used to construct the XML output (hence should not be left to the system).

B.2 Head

We assume that the XMAS head contains explicit collection labels.

Definition 4 (Head semantics)
The semantics of a XMAS rule `CONSTRUCT` *head* `WHERE` *body* is given as follows:[12]

Let $\bar{X} = X_1, \ldots, X_n$ be the sequence of free variables of *body*, and Bs the ordered list of variable bindings which satisfy *body*. Then the semantics of *head* is defined as

$$sem(head, Bs, \emptyset)$$

where the function $sem(C, Bs, \beta)$ is defined as follows:[13]

[12]Here, the selection σ_β and the projection $\pi_{\bar{X}}$ produce lists of tuples/variable bindings instead of sets (thus preserving a given order).

[13]cf. Section 2.1

- if $C = s$ *% string constant*

  then $sem(C, Bs, \beta) := [\, s\,]$

- if $C = X_i$ *% variable*

  then $sem(C, Bs, \beta) := [\, x_i \mid \bar{x} \in \sigma_\beta(Bs)\,]$

- if $C = \langle p\ \boldsymbol{A}\rangle\, C'\, \langle/\rangle\ \{\bar{Y}\}$ *% element pattern (with collection label)*

  then $sem(C, Bs, \beta) := [\, \langle \iota \circ \beta(p\ \boldsymbol{A})\rangle\ sem(C', Bs, \iota \circ \beta)\ \langle/\rangle \mid \iota \in \pi_{\bar{Y}}(\sigma_\beta(Bs))\,]$

- if $C = f(C_1, \ldots, C_n)$ *% list function*

  then $sem(C, Bs, \beta) := f(sem(C_1, Bs, \beta), \ldots, sem(C_n, Bs, \beta))$ □

For every head expression C, the explicit collection label $ex_label(C)$ can be derived, based on $free(C)$, i.e., the free variables of C:

Definition 5 (Free Variables)
The free variables of a XMAS expression in the head are defined as follows:

$$free(C) := \begin{cases} \{\} & ;\text{ if } C = s \\ \{\} & ;\text{ if } C = [C'] \\ \{X\} & ;\text{ if } C = X \\ free(p) \cup free(\boldsymbol{A}) \cup free(C') & ;\text{ if } C = \langle p\ \boldsymbol{A}\rangle\ C'\ \langle/p\rangle \\ free(C') \setminus \{\bar{X}\} & ;\text{ if } C = C'\ \{\bar{X}\} \end{cases}$$

□

Observe that a collection label $\{\bar{X}\}$ binds exactly the variables $\bar{X}$ and a bracket expression $[C']$ binds all variables free variables inside C'.

Using this definition, we can define for every expression the *explicit collection label* as follows:

Definition 6 (Explicit Collection Label)

$$ex_label(C) := \begin{cases} \{\} & ;\text{ if } C = s \\ free(C') & ;\text{ if } C = [C'] \\ \{\} & ;\text{ if } C = X \\ \{\} & ;\text{ if } C = \langle p\ \boldsymbol{A}\rangle\ C'\ \langle/p\rangle \\ \{\bar{X}\} & ;\text{ if } C = C'\ \{\bar{X}\} \end{cases}$$

□