UNIVERSITY OF AMSTERDAM

UvA-DARE (Digital Academic Repository)

Feature grammar systems. Incremental maintenance of indexes to digital media warehouses

Windhouwer, M.A.

Publication date
2003

Link to publication

Citation for published version (APA):
Windhouwer, M. A. (2003). *Feature grammar systems. Incremental maintenance of indexes to digital media warehouses*. [Thesis, externally prepared, Universiteit van Amsterdam].

General rights
It is not permitted to download or to forward/distribute the text or part of it without the consent of the author(s) and/or copyright holder(s), other than for strictly personal, individual use, unless the work is under an open content license (like Creative Commons).

Disclaimer/Complaints regulations
If you believe that digital publication of certain material infringes any of your rights or (privacy) interests, please let the Library know, stating your reasons. In case of a legitimate complaint, the Library will make the material inaccessible and/or remove it from the website. Please Ask the Library: https://uba.uva.nl/en/contact, or a letter to: Library of the University of Amsterdam, Secretariat, P.O. Box 19185, 1000 GD Amsterdam, The Netherlands. You will be contacted as soon as possible.

UvA-DARE is a service provided by the library of the University of Amsterdam (https://dare.uva.nl)

Download date:18 Nov 2026

Chapter 6

Feature Detector Scheduler

The library is a growing organism.
S.R. Ranganathan - Law five of *The Five Laws of Library Science*

The FDE, as introduced in Chapter 4, can easily construct a huge collection of hierarchical structures, which are related by references and thus, on a higher level of abstraction, form a graph. The previous chapter introduced a persistent storage model for these structures based on Monet. However, the annotations contained in these structures should be kept synchronized with the (external) multimedia objects they describe. Next to those external changes, the annotation extraction algorithms may change over time, *i. e.* another reason for maintenance. This chapter describes the contour of the *Feature Detector Scheduler* (FDS), whose main goal is to steer the FDE to execute incremental parses and thus propagate the localized changes. Unfortunately, there is no complete FDS implementation experience, but the sketch is backed by prototypes of the core parts.

The core parts of the FDS are shown in Figure 6.1. In the subsequent sections these components will pass the revue and their relations to each other will be described in some depth. The last section will also contain a short discussion on the implementation.

6.1 The Dependency Graph

The basis of the FDS is its analysis of the dependency graph. The dependency graph describes how all the known symbols of a feature grammar relate to each other. Figure 6.2 shows the dependency graph based on the example HTML feature grammar (see Example 3.1).

The symbols play these roles (notice that symbols may play multiple roles):

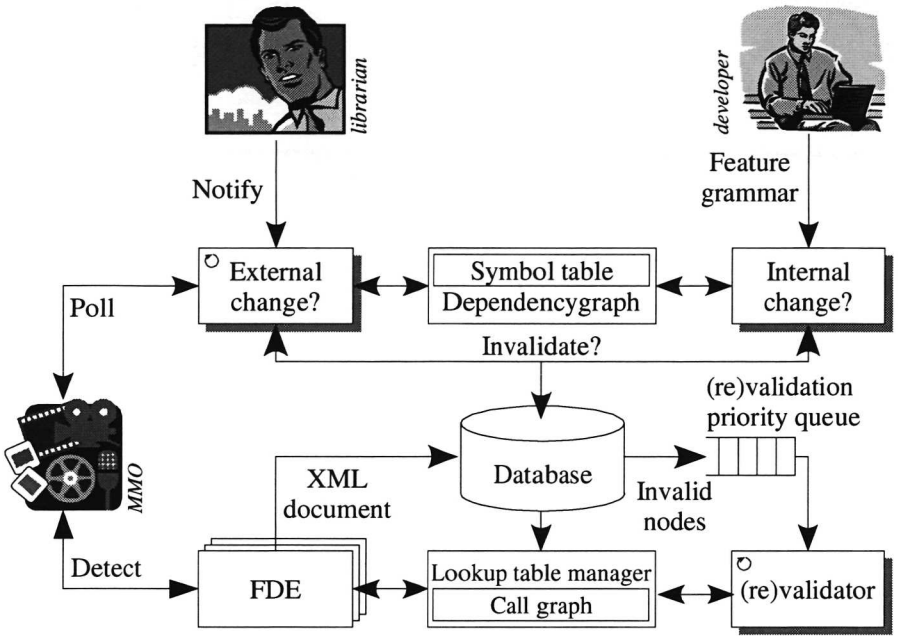

Figure 6.1: The FDS components

start symbols are the roots of the hierarchical structures, *e. g.* the $WebObject$ symbol;

detector symbols are the symbols which are associated with an annotation extraction algorithm, *e. g.* the $WebHeader$ symbol;

transparent symbols are anonymous symbols introduced by the rewrite rules, *e. g.* the $_grp_1_$ symbol, which is introduced to capture the grouped optionality of the $WebHeader$ and $WebBody$ symbols;

terminal symbols are instantiated with a value belonging to the symbols domain, *i. e.* they contain the real annotation information like the $date$ atom;

non-terminal symbols are the symbols without any other specific role. They provide an intermediate semantic (grouping) level.

This information is stored in the symbol table (see Section 4.3.1). From the production rules the basic dependency graph is derived. Then the various path expressions are resolved on this meta-level, *i. e.* all instantiation (value) related predicates are skipped. For example for this whitebox predicate:

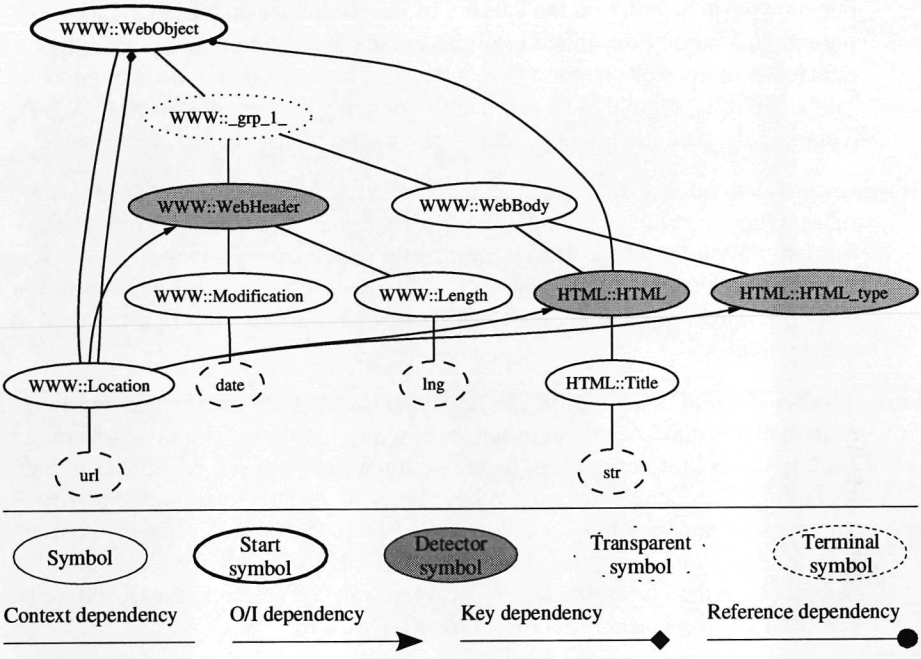

Figure 6.2: The HTML feature grammar dependency graph

```
%detector    ColorMap [
                 some $RGB in RGB satisfies
                      $RGB/Red != $RGB/Green
                   or $RGB/Red != $RGB/Blue ];
```

these paths are resolved, with $ColorMap$ as the context node, within the graph:

1. $self :: */preceding :: RGB$
2. $self :: */preceding :: RGB/child :: Red$
3. $self :: */preceding :: RGB/child :: Green$
4. $self :: */preceding :: RGB/child :: Blue$

This whole construction process results in these relationships between the nodes in the dependency graph:

context dependencies are the most basic parent/child relationship between the LHS and the symbols in the RHS of a rule. The child, *i. e.* the RHS symbol, always

depends for its validity on the validity of the parent, *i. e.* the LHS symbol. For parents the validity depends on all the mandatory children, *i. e.* those dependencies with a lower bound of one or more. The *WebObject* does not depend on the *_grp_1_* symbol to be valid, however, *_grp_1_* depends on *WebObject*. This way the parent/child dependency also enforces sibling dependencies;

output/input dependencies between annotation extraction algorithms: an algorithm takes it input parameter from the result parse tree of another algorithm. Assuming that the output of the algorithm directly depends on this input, changes in the values of the parameter symbols lead to the need to rerun the algorithm. For example: the *WebHeader* detector depends on the *Location* non-terminal as input parameter;

key/reference dependencies define the relationships between quasi-roots and foots. Start symbols may specify an initial set of required tokens. The same tokens are used to resolve references, *i. e.* they function as a composite key to an (unique) hierarchical structure. When those key values change the referential integrity of the relationships have to be revalidated. A tree rooted by a *WebObject* symbol is identified by its *Location*. Links found in a HTML page are represented by references to the corresponding *WebObject* trees. These references should be removed when specific *WebObject* trees are deleted.

Upon a change in the feature grammar system the dependency graph is recreated. However, to localize change and start updating the parse trees in the database the FDS needs one (or even several) starting points. The next section shows how these starting points are identified.

6.2 Identifying Change

Section 1.2.4 identified two sources, which are also reflected in the feature grammar language (see Section 3.2.5), of change in a DMW: external and internal. The FDS has to cope with both sources by using the handles specified in the feature grammar system and given by the developer.

6.2.1 External Changes

The first type of changes happen outside of the annotation system: the source multimedia data, which may or may not reside in the same database as the annotations, changes[1]. The FDS may be notified of such changes by four sources: (1) the librarian, (2) the FDE requests validation of a (new) multimedia object,(3) an (external) system, *e. g.* a database trigger, or (4) the exploration date of stored data passes a certain threshold.

[1]Notice that a new multimedia object is also considered a change.

The FDS reacts to these notifications by polling the source data. This polling is supported by a special detector, which is associated with a start symbol. The polling detector always receives the initial token set plus additional tokens needed to establish any modification of the source data since the annotations were extracted. For example the polling for web objects takes not only the location of the object but also the stored modification date (see the example in Section 3.2.5). The polling detector can then simply send a HTTP `HEAD` request to the web server and determine if the returned modification date is newer than the stored date.

When the object is considered modified its root node is invalidated in the database and added to the (re)validation queue.

6.2.2 Internal Changes

Updates to the feature grammar system and its associated detector implementations are considered internal to the annotation system, *i. e.* it has all the information about them. The stored annotations should reflect the current output of these implementations. These internal changes are always related to detector symbols, *i. e.* grammar components, and thus to the start condition, the detector function and the stop condition, *i. e.* the production rules. Notice that instead of single multimedia objects, as is the case with external change, internal changes refer to sets of multimedia objects.

The Start Condition

The start condition changes when one or more of the XPath expressions for binding of the parameters change. This invalidates the lookup table, *i. e.* the memoized parse trees, rooted by this detector as the input sentences may have changed. The consequence is that all these trees are deleted and the parse trees they were member of are scheduled for revalidation. During the revalidation process the new lookup table will be reconstructed using the new input sentences.

The Detector Function

Changes in the implementation of a detector function are reflected by the version declaration for the specific detector. Such a version consists of three levels. The lowest level indicates a *service revision*. These revisions will not lead to invalidation of any nodes in the current stored parse trees, so the FDS does not have to take any further action, *i. e.* the data is updated when an external change or more severe internal change triggers revalidation. Changes of the next level, the *minor revisions*, will lead to invalidation of the partial parse trees. However, the data may still be used to answer queries. Those revalidations are scheduled with a low priority. High priorities are used for invalidations caused by *major revisions*. In these cases the changes are so severe that the stored data has become unusable and are deleted from the database.

The Stop Condition

The production rules of the feature grammar component describe the valid output sentences. When these rules change the output sentences have to be revalidated. Changes to these rules can be found by a tree matching algorithm [ZS90, CGM97, CAM01]. With the rise of XML these algorithms have found a new public, and several implementations are (freely) (on line) available (*e. g.* [IBM01, YW03]). From both feature grammar versions the derived XML schema document[2] (see Section 5.2.1) is fed into the diff implementation. For example, the *Delta* tool [Ltd03] will report these changes, when the web objects MIME type is extracted by $WebHeader$:

```
...
    <WWW:WebHeader deltaxml:delta="WFmodify">
        <WWW:Modification deltaxml:delta="unchanged"/>
        <WWW:Length deltaxml:delta="unchanged"/>
        <WWW:MIME_type deltaxml:delta="add" type=".non-terminal."/>
    </WWW:WebHeader>
...
    <WWW:MIME_type deltaxml:delta="add" type=".non-terminal.">
        <WWW:Primary type=".non-terminal."/>
        <WWW:Secondary type=".non-terminal."/>
    </WWW:MIME_type>
    <WWW:Primary deltaxml:delta="add" type=".non-terminal.">
        <fg:str type=".terminal.atom."/>
    </WWW:Primary>
    <WWW:Secondary deltaxml:delta="add" type=".non-terminal.">
        <fg:str type=".terminal.atom."/>
    </WWW:Secondary>
...
```

In this case the detector can directly be identified, in other cases the context dependencies in the dependency graph have to be traversed from child to parent to the first detector node(s)[3].

6.3 Managing Change

Using the indicators of the previous section a priority queue has been filled with invalid (partial) parse trees. In the process of revalidation the FDS communicates about the parse trees with several FDEs and keeps knowledge about the global relationships between these trees. The next sections will describe this process and the use of this knowledge in some more detail.

[2] if the original grammar is not available anymore the schema can be derived from the meta information in the Monet database.

[3] When a intermediate non-terminal is reused in different contexts, there may be multiple detector roots.

6.3.1 The (Re)validation Process

The invalidation of nodes in the parse tree, and the scheduling of their revalidation, is handled by the FDS using these steps (the example assumes that the $WebHeader$ detector implementation has changed, *i. e.* an internal change):

1. The FDS has invalidated all partial parse trees which have an instantiation of a $WebHeader$ symbol as root. This involved $WebHeader$, $Modification$ and $Length$ nodes (and related terminal nodes), as can be derived by traversing the rule dependencies. For these parse trees incremental parsing processes are scheduled, which can be handled by the FDE. The followup action of the FDS is determined by the result returned by the FDE.

2. If the sub-tree is still valid, the output/input dependencies are checked for modification. If there has been a modification the dependent detector needs to be revalidated.

3. If the subtree has become invalid, the context dependencies are followed upward to the first detector or start symbol which is marked invalid. The FDS will repeat the whole procedure for these invalid symbols.

4. The referential integrity is checked using key/reference dependencies, when key values have been changed or parse trees have become invalid.

6.3.2 Lookup Table Management

During the revalidation task other (memoized) parse trees (see Section 4.3.3) may be encounter. The FDE provides the input sentence for the mapping described by the requested parse tree. The FDS will query the lookup table stored in the database for the existence of this mapping and return the appropriate information:

1. when the mapping is available: the unique identifier of the tree;

2. when the mapping does not exist: a *null* value;

3. when the mapping is unknown: a new unique identifier;

4. when the mapping is under construction: a timeout after which the FDE will have to resent its request;

5. when a deadlock is detected (see below): the unique identifier and the type of deadlock (direct of indirect).

Next to the requests to resolve or initialize a parse tree, the FDEs also inform the FDS about the validation results (independently of the database storage). Using this information the FDS maintains a call graph of parse trees currently under construction.

Detecting a cycle in this graph indicates a (in)direct deadlock, which is reported to the FDE. The FDE passes this information on to the detector (see Section 4.3.5), and will return the result, *i. e.* the mapping is available or does not exist (meaning that the deadlock could not be resolved).

6.4 Discussion

As stated in the introduction this chapter contains a sketch of the envisioned system architecture of the FDS, backed by a partial prototype implementation. Figure 6.1 shows the various components. Based on the detection and localization of internal and/or external changes incremental FDE runs are scheduled and controlled by the FDS.

The external change detection component is mainly an interface to the polling detectors. This is a continuous process. With a slight extension to the XML schema documents, *i. e.* to also contain the XPath expressions, the internal change detector component can be completely implemented around an implementation of a tree difference algorithm. The algorithm to derive the dependency graph from the symbol table has been implemented in the latest version of Acoi, as will be shown in the next chapter. The FDE has also been implemented (see Chapter 4) and experiments with incremental parsing have been conducted. However, adaptations, like submitting partial parse trees to both the database and the FDS, may be needed. The (re)validator and the lookup table manager have not been implemented, so the sketches of these components have to be validated by an actual implementation in the future.