WEBVIGIL: SENTINEL SPECIFICATION AND USER-INTENT BASED CHANGE DETECTION FOR EXTENSIBLE MARKUP LANGUAGE (XML)

The members of the Committee approve the master's
thesis of Jyoti Jacob

Sharma Chakravarthy
Supervising Professor ______________________________

Alp Aslandogan ______________________________

Mohan Kumar ______________________________

WEBVIGIL: SENTINEL SPECIFICATION AND USER-INTENT BASED CHANGE DETECTION FOR EXTENSIBLE MARKUP LANGUAGE (XML)

by

JYOTI JACOB

Presented to the Faculty of the Graduate School of

The University of Texas at Arlington in Partial Fulfillment

of the Requirements

for the Degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE AND ENGINEERING

THE UNIVERSITY OF TEXAS AT ARLINGTON

May 2003

ACKNOWLEDGEMENTS

First and foremost, I would like to thank my advisor, Dr. Sharma Chakravarthy, for giving me an opportunity to work on this challenging topic and providing me ample guidance and support through the course of this research.

I would like to thank Dr. Mohan Kumar and Dr. Alp Aslandogan for serving on my committee.

I am grateful to Anoop Sanka and Naveen Pandrangi, part of the WebVigiL team, for their invaluable help and advice during the implementation of this work. I would like to thank all my friends in the ITLAB especially Raman, Alpa and Ambika for their help, support and encouragement.

I would like to acknowledge the support of the Office of Naval Research, the SPAWAR System Center-San Diego & by the Rome Laboratory (grant F30602-01-2-0543), and by NSF (grants IIS-0123730 and IIS-0097517).

I would like to thank my friend, Ganeshram Iyer for his guidance, support and patience. I would also like to thank my family and friends for their love and constant support throughout my academic career.

March 31, 2003

ABSTRACT

WEBVIGIL: SENTINEL SPECIFICATION AND USER-INTENT BASED CHANGE DETECTION FOR EXTENSIBLE MARKUP LANGUAGE (XML)

Publication No. ______

Jyoti Jacob, M.S

The University of Texas at Arlington, 2003

Supervising Professor: Sharma Chakravarthy

With the exponential increase of information on the web, there is a need for efficient retrieval and notification of selective information. Currently, users have to retrieve (by pull/poll) the pages manually to check for changes of interest, resulting in waste of human resources and associated high cost. Hence, WebVigiL is designed as a general-purpose, active capability-based monitoring and notification system, for handling the specification, management, and propagation of changes on unstructured/semi-structured documents based on user specification.

In this thesis, we present the semantics of a change specification language for specifying user policies for web page monitoring. We also present a design for efficient validation and storage of user specifications in a persistent repository. For handling customized change detection based on user-intent, we propose an algorithm for change detection to the contents of semi-structured documents. Though the approach taken is general, we will explain the change detection in the context of XML documents.

TABLE OF CONTENTS

LIST OF ILLUSTRATIONS

LIST OF TABLES

To My Family and Friends

CHAPTER 1

INTRODUCTION

The data on the Internet is growing at a rapid rate. The number of documents is large and spread over multiple repositories. This has greatly affected the way information is accessed, delivered and disseminated.

Users at present are not only interested in the new information available on the web pages but also in retrieving the changes of interest in a timely manner. For example, the user may not be interested in structural changes. Even for the content, the user may only be interested in specific changes (to keywords, phrases etc.). Traditionally, the burden of information retrieval is on the user, as he has to poll the interested websites manually to learn about changes to the contents of the page and pose appropriate user query to extract the information. As the amount of information is large and distributed over large number of sites, the users prefer the new information to be pushed to them rather than pulling it every time it changes. Hence, the emphasis is on selective change notification i.e. the changes are notified to the user based on specified interest/ policy. There are many situations when the user needs to be made aware of changes as soon as a document is modified. Manual polling for information change will result in missing some changes or delay in notification. For example, in many industries, the life cycle of projects are in years. Large number of documents such as the specification, requirement analysis, design etc are required and updated during the project life cycle. These changes need to be notified immediately to the project team members to ensure that the changes are propagated properly to other relevant documents for immediate actions. Hence, an approach is needed which replaces periodic polling and notifies the user of the relevant changes in a timely manner. This issue is even more relevant to web documents. People want to monitor stocks and need to know as and when the prices change. Students need to be notified immediately of changes to the course web page for which they are registered. Web administrators need to know, when and what has been changed on the web page by the developers. In addition, web site developers need to keep track of changes to their competitor's web pages. People need to know of changes to specific news items or articles of interest on the web.

Different paradigms have been used for monitoring the pages of interest. The different approaches used for information extraction are:

Pull Paradigm: In the traditional pull paradigm, the user retrieves information by performing an explicit action in the form of a query, application, or transaction execution at a pre-defined polling interval. The limitations of this approach are that the pages of interest have to be checked manually on a periodic basis. Hence, this approach results in changes being missed and there is added burden of checking all pages. In addition, when large number of web sites need to be monitored, it becomes very difficult for the user to keep track of all the changes to the web pages. Many websites such as stock pages provide the facility to notify the users of periodic changes to the page but they too rely on pull paradigm to refresh the pages at a regular interval.

Push Paradigm: In the push paradigm, the user does not have to query or retrieve information as it changes. The system is responsible for accepting user needs (in the form of situations to monitor, business rules, constraints, profiles, continuous search queries) and informs the user (or a set of users) when something of interest happens. Though it reduces the amount of data transfer, the underlying system requires a lot of enhancements or has to incorporate agents or mediators that can carry this out in a non-intrusive manner. In other words, the system needs to have the capability to selectively push information. At present most of the systems use a mailing list to send the same compiled changes to all the users. The limitation of this approach is that irrespective of the user's interest, he/she receives all the changes to the page.

1.1 Background

Monitoring Systems: For efficient retrieval and propagation of information, a system is needed, which combines both push and pull paradigms. WebVigiL is a monitoring system, which offers an alternate paradigm to monitoring changes using the combination of push-pull paradigm with the help of active capability. WebVigiL pulls the information from the web server based on user profile and propagates/pushes the relevant information to the end user. Though work in active technology has been well established in the database field, to use this technology for advanced applications such as monitoring rapidly changing web pages on a large network centric environment such as the web, enhancements need to be made to the existing architecture. WebVigiL also considers quality of service requirements such as timeliness into consideration to detect and notify the changes. In addition, WebVigiL is a scalable system, designed to detect even composite changes for large number of users. The overview of the paradigm used and the basic approach taken for effective monitoring is discussed in [1]

1.2 WebVigiL Architecture

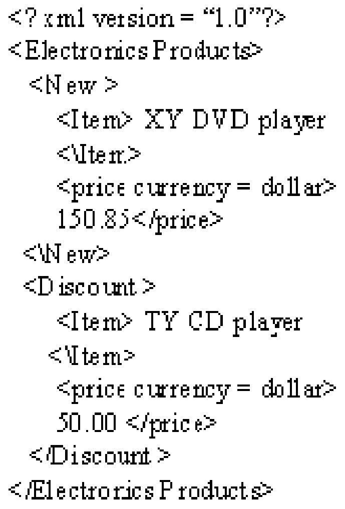
A well-defined change specification language is designed, which the user can use to submit his monitoring request or a sentinel. User-defined sentinels are verified both syntactically and semantically prior to persisting their details in the Knowledgebase. The change specification language, the verification module and the design of the Knowledgebase are discussed in detail in the subsequent chapters. Once a sentinel is validated, the ECA Rule Generator generates ECA rules for the run time management of that sentinel. The fetch module fetches pages for all active (or enabled) sentinels, forwards them to the version management module for adding them to the page repository and notifies the change detection module. The change detection module detects the changes according to the specification and notifies the presentation module. All the modules have been designed for efficient execution and scalability requirements.

Details about the WebVigiL architecture are given in [2] and also discussed in chapter 2. WebVigiL monitors changes to documents such as the Hypertext Markup language (HTML) and the eXtensible Markup Language (XML). As the focus of this thesis report is on change detection to XML documents, the structure and format of XML documents is described in section 1.3.

1.3 XML

eXtensible Markup Language (XML) is a simple text driven language derived from SGML for structured documents and data on the Web [3]. It was introduced in 1998 after being approved by the World Wide Web consortium. XML was originally designed to meet the challenges of large-scale electronic publishing, but now is increasingly used for data exchange. XML is likely to replace HTML as the standard web publishing language in the near future.

XML is a well- formed document and is extensible. The user can define his/her tags, which describe the contents. The given XML tags in a document is validated by a Document Type Definition (DTD)[4] or XML Schema, both of which are selfdescriptive. An XML document does not contain presentation tags. Unlike HTML, presentation and contents are separate in XML. To briefly describe XML, consider a small part of a well-formed XML document as shown in Figure 1. 1. The tags are called elements in XML. Every XML document must have only one root element that contains all other elements in the document. All tags must be properly terminated with an end tag. Hence, XML is a well-formed document.



```
<? xml version = “1.0”?>
<Electronics Products>
  <New >
     <Item> XY DVD player
     <\Item>
     <price currency = dollar>
     150.85</price>
  <\New>
  <Discount >
     <Item> TY CD player
    <\Item>
     <price currency = dollar>
     50.00 </price>
  </Discount >
</Electronics Products>
```

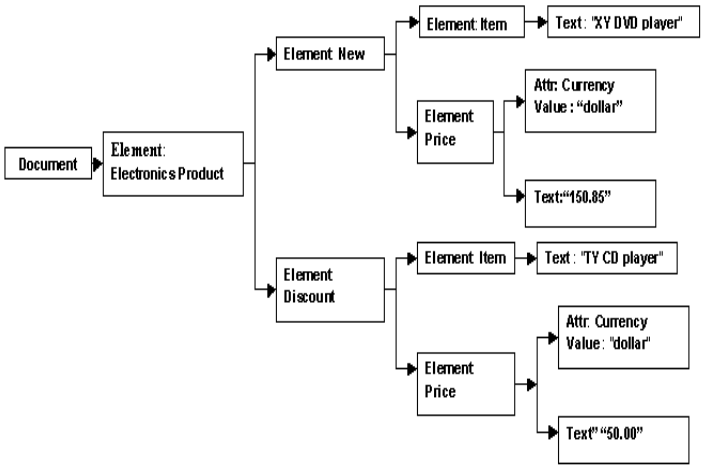
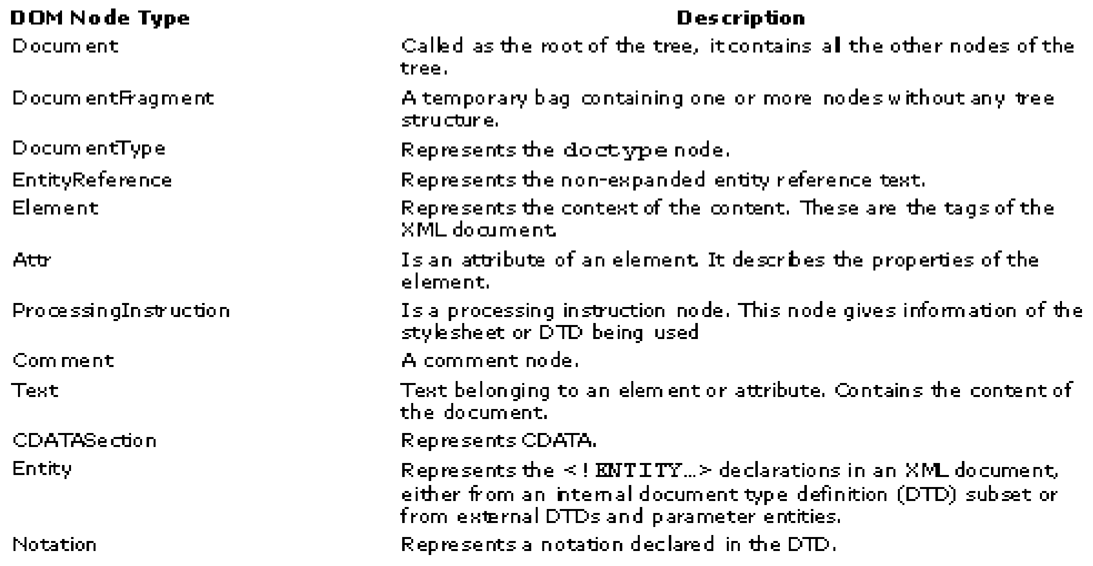


Figure 1. 1 XML document Figure 1.2 DOM tree of the XML document

The XML document can be represented as a Document Object Model (DOM). According to [4], the W3C DOM is a language and platform-neutral definition, which represent the XML documents as objects, which allows the user to read, search, modify, add to and delete data from a document. Hence, DOM allows for a standard functionality for document navigation and manipulation of the content of the XML documents. Apart from elements, many other DOM nodes constitute the XML document. The DOM of the XML document given in Figure 1. 1 is shown in Figure 1.2.

Details about the various types of DOM nodes and their description are given in Figure 1. 3.



DOM Node Type	Description
Document	Called as the root of the tree, it contains all the other nodes of the tree.
DocumentFragment	A temporary bag containing one or more nodes without any tree structure.
DocumentType	Represents the doctype node.
EntityReference	Represents the non-expanded entity reference text.
Element	Represents the context of the content. These are the tags of the XML document.
Attr	Is an attribute of an element. It describes the properties of the element.
ProcessingInstruction	Is a processing instruction node. This node gives information of the stylesheet or DTD being used
Comment	A comment node.
Text	Text belonging to an element or attribute. Contains the content of the document.
CDATASection	Represents CDATA.
Entity	Represents the <!ENTITY...> declarations in an XML document, either from an internal document type definition (DTD) subset or from external DTDs and parameter entities.
Notation	Represents a notation declared in the DTD.

Figure 1. 3 Description of DOM Nodes

1.4 Focus of the Thesis

The main contributions of this paper are:

Use of a new algorithm for customized change detection of ordered semi-structured documents such as XML: An XML document is a semi-structured, well-formed document with ordered elements. In addition, WebVigiL supports customized changes to sections of the document such as phrases, keywords etc. The existing tools outlined in Chapter 3 cannot be used for customized change detection for ordered XML documents as per the requirements of WebVigiL. Hence, a new algorithm is proposed, which takes into consideration the structure and order of the XML document and detects customized changes.

Expressiveness of change specification and its semantics: In order for the user to specify his/her notification and monitoring requirements, an expressive change specification language is needed which incorporates the monitoring requirements of the user and also captures inheritance, event-based duration and composite changes. We have formulated a change specification language wherein the user can specify the page of interest, the type of change, the duration of the monitoring request, fetching and notification details and the preferences for comparison.

Efficient storage and retrieval of the meta-data: The meta-data extracted from the user specification needs to be stored in a persistent and recoverable manner. In addition, WebVigiL allows the user to inherit properties from previous sentinel to create a new sentinel. Hence, the meta-data information should be stored only after proper syntactic and semantic validation for correctness of the data. The data stored in the Knowledgebase should be available for other WebVigiL modules at run-time. Hence, efficient retrieval of information is important. In addition, APIs have been developed for efficient retrieval for inheritance of sentinel properties.

The remainder of the thesis is organized as follows. In Chapter 2, we give an overview of the current architecture. In Chapter 3, related change monitoring tools and specification languages are discussed. Chapter 4 addresses issues related to change detection to XML pages and proposes an algorithm to detect customized changes on XML documents. In Chapter 5, we discuss the syntax and semantics of the change specification language. In Chapter 6, the design of the Knowledgebase and the validation and verification module is discussed in detail. Chapter 7 deals with the implementation issues of these modules and Chapter 8 concludes the thesis with emphasis on future work.

CHAPTER 2

CURRENT ARCHITECTURE

WebVigiL is a change detection and notification system, which can monitor and detect changes to unstructured and semi-structured documents in general. The current work addresses HTML/XML documents that are part of a web repository. WebVigiL aims at investigating the specification, management, and propagation of changes as requested by the user in a timely manner while meeting the quality of service requirements[2].

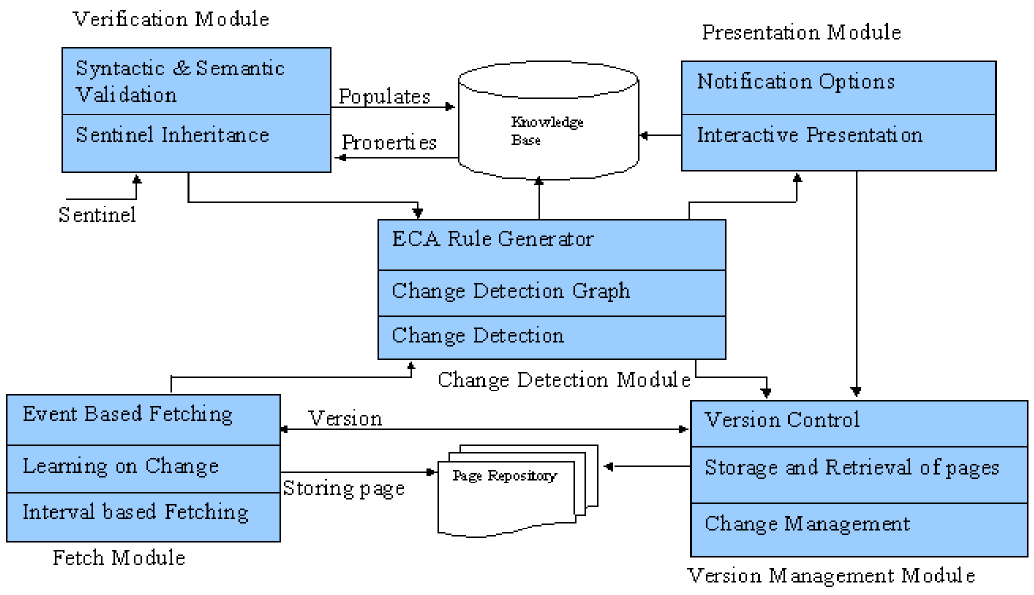


Figure 2. 1 WebVigiL Architecture

The high-level block diagram shown in Figure 2. 1 details the architecture of WebVigiL. Users specify their interest in the form of a sentinel that is used for change detection and presentation. Information from the sentinel is extracted, and based on user-profile, changes are detected and notified to the user in an appropriate manner. The functionality of each module in the architecture is described briefly in the following sections.

2.1 User specification

User-specified changes need to be monitored and notified to the user in different ways and on different devices as specified in their profiles/policies. Hence, there is a need to define an expressive specification language to specify user profile. WebVigiL provides an expressive language with well-defined semantics for specifying the monitoring requirements of a user, pertaining to the Web. The Change Specification language is discussed in detail in Chapter 5.

2.2 Knowledgebase (KB)

Knowledgebase is a persistent repository containing meta-data about each user, and details of the contents of the sentinel (frequency of notification, change type etc.). User input is parsed and required information is extracted and stored for later use. Details about the Knowledgebase and validation module are discussed in Chapter 6.

2.3 ECA Rule Generator

Every valid user request arriving at WebVigiL, initiates a series of operations that occur at different points in time. Some of these operations are: creation of a sentinel (based on start time), monitoring the requested page, detecting changes of interest, notifying the user(s) of the change, and deactivation of sentinel. In WebVigiL, for every sentinel, the ECA rule generation module generates Event Condition Action (ECA) rules [5, 6] to perform some of these operations.

Briefly, an event-condition-action rule has three components: an event (occurrence of an event), a condition (checked when the associated event occurs), and an action (operations to be carried out when the condition evaluates to true). The ECA rules along with the local event detector (LED) [7] are used for: i) generating fetch rules for retrieving pages, ii) detecting events of interest and propagating pages to detect primitive and composite changes, and iii) for activation and deactivation of sentinels.

2.3.1 Activation/Deactivation

WebVigiL does not monitor continuous queries but monitors interval-based monitoring request. Hence, each sentinel has a start and end time during which a sentinel is enabled by default. A sentinel can be disabled (does not detect changes during that period) or enabled (detects changes) by the user during its lifespan. In WebVigiL it is the responsibility of ECA rule generation module to create appropriate events and rules to enable/disable sentinels. We achieve this as follows. Consider the scenario where s1 is defined in the interval [12/02/02, 01/02/03]. At time 12/02/02 sentinel s1 has to be enabled. Following are the events and rules that are generated to enable sentinel s1at compile time:

```
Event Temp1    =  createTemporalEvent(12/02/02)
Event start_s1   =  createEvent("start_s1")
Rule T1            = createRule (Temp1)
Event Fetch_s1 = createPeriodicEvent (Start_s1, 2, End_s1)
```

Fetch_s1 is a periodic event [8] created with “start_s1” as the start event, the frequency of page fetch, and End_s1 as the end event. The rule associated with it handles the fetching of pages for s1. A rule associated with an event is fired/triggered when the event is raised. More than one rule can be associated with an event. When event Temp1 is triggered at the specified time point, rule T1 is executed, which in turn raises the event start_s1. Triggering of the event start_s1 activates the sentinel s1 by activating the periodic event used for fetching the web page specified in s1. Now, if another sentinel s2, which is defined over the interval [start(s1), end(s1)] arrives, the following events and rules are generated in order to enable s2:

```
Event start_s2      = createEvent(“start_s2”)
Rule r_start_s2    = createRule(start_s1)
```

Here we are associating the rule r_start_s2 to the event start_s1, which was created at the arrival of sentinel s1. This rule actually raises the start_s2 event to activate the periodic event associated with s2. In this manner, ECA rules are used to asynchronously activate and deactivate sentinels at run time. Once the appropriate events and rules are created, the local event detector handles the execution at run time. By enabling/disabling of sentinel we mean addition/deletion of that sentinel to the change detection graph that is detailed in section 2.5.

2.4 Change Detection

The web user’s interest has extended from mere viewing of information to monitoring evolution of selective information on the pages. Hence, the change detection tool should be capable of detecting preferred change, such as the appearance/disappearance of objects of user’s interest on a page. Consider the scenario: A student wants to monitor the college schedule of classes for a particular course name (keyword). In such cases, detecting changes to the complete page results in excessive computation and dissemination of irrelevant information. Hence, there is a need to support detecting changes based on user's intent. WebVigiL supports change detection of Hypertext Markup Language (HTML) and eXtensible Markup Language (XML), which are the standard formats for electronic publishing on the web. In HTML, the changes are detected to content-based tags such as links and images, presentation tags and changes to specific content such as keywords, phrases etc. In XML, currently, we support only change detection to content. The change detection will be discussed in Chapter 4.

2.5 Change Detection Graph (CDG)

When a page is fetched, for every sentinel that is interested in that page, change is computed and notified to the user. In situations where there are two or more sentinels interested in the same type of change on the same page we have to compute the change more than once. We avoid this by capturing the relationship between the pages and sentinels, and grouping the sentinels on the change and target web page. Hence all sentinels interested on the same type of change and on the same page are grouped together. In order to represent this relationship we construct a change detection graph (CDG). Consider a sentinel s3, monitoring the change "images" on the URL www.yahoo.com. Another sentinel s1 also monitors the same web page but for a composite change "links" and "images". The change detection graph for these two sentinels is shown in Figure 2. 2. The different types of nodes in the graph are as follows:

URL node: A URL node is a leaf node that denotes the *page of interest*. The number of URL nodes in the graph is equal to the number of *distinct* pages the system is monitoring at any particular instant of time.

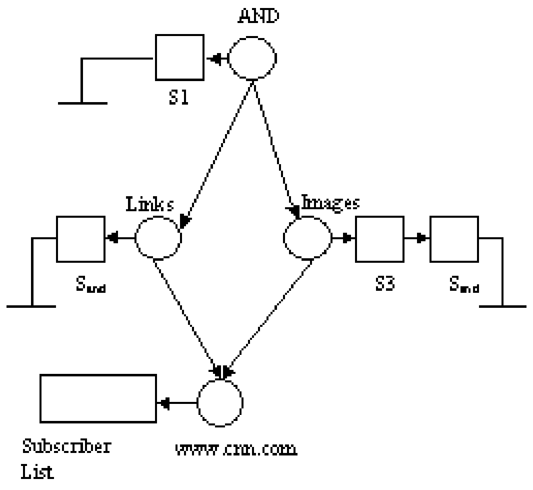


Figure 2. 2 Change Detection Graph

Change type node: All level-1 nodes in the graph belong to this category. This node represents the *type of change* on a page (all words, links, images, keywords, phrases, table, list, regular expression, any change).

Composite Node: A Composite node represents a combination of change types. The combinations are created using binary operators as explained in section 5.3. All higher-level nodes (> level-1) in the graph belong to this type. Currently we support composite changes on a single page.

In the graph, to facilitate the detection and propagation of changes, the relationship between nodes at different levels is captured using the subscription/notification mechanism. The higher-level nodes subscribe to the lower level nodes in the graph. This subscription information is maintained in the subscriber list at each node. At the URL node, this list contains the references to the change type nodes. At the change type nodes each sentinel will have a subscriber that will contain the references to the composite nodes. When a page is fetched, the associated URL node is notified about the page. The URL node propagates this page to all the change type nodes that have subscribed to it. Finally at the change type nodes the change is computed between the current pages received and an appropriate reference page (based on the compare option) that is fetched from the page repository. If there is any change then the sentinels subscribed to it are notified. When this change type is a part of a composite change, those composite nodes are notified.

2.6 Event-Based Fetching

WebVigiL monitors only those pages that are registered with it. For this purpose it has to fetch the pages when a change in meta-data (such as the last modified time stamp or checksum) is detected.

2.6.1 ECA Paradigm for Fetching

Event-Based Fetch module is responsible for monitoring and fetching pages that are of interest to the sentinels registered with WebVigiL. Periodic events [8] are defined and rules are associated with them as discussed earlier. Whenever a periodic event occurs, the corresponding rule is fired, which then checks (condition part of the rule) for change in meta-data of the page and fetches the page (action part of the rule) if there is a change in meta-data. Thus the periodic event controls both the polling interval and the lifespan of the fetch process. By meta-data of the page, we mean page properties such as the page size, last modified time stamp and checksum of the page. A fetch cycle for a page is triggered only when there is a change between the meta-data of the current version of the page to that of the previous version. Depending upon the nature (static/dynamic) of page being monitored the complete set or subset of the meta-data is used to evaluate the change. A fetch rule is created and used to poll the page of interest specified in the sentinel.

Best Effort Rule: In situations where the user has no information about the change frequency of a page, it is necessary to tune the fetch frequency to the actual change frequency of a page. BE Rule uses a best-effort Algorithm [9] to achieve this tuning. In the best-effort algorithm (BEA), the next fetch interval (P_{next}) is determined from the history of changes to that page. When the next polling interval is determined, the BE Rule changes the interval "t" of the periodic event.

Interval-Based Rule: The user can explicitly specify a fetch frequency. A periodic event with periodicity (interval t) equal to the given interval is created and an IB rule IB_i is associated with it to fetch the page. As a result there will be more than one IB rule on a given page with different or same periodicity, where each rule is associated with a unique periodic event (i.e., with different start and end times).

2.7 Caching and Management of pages

An important feature of WebVigiL architecture is its centralized server based repository service that archives and manages versions of pages. WebVigiL retrieves and stores only those pages needed by a sentinel. The primary purpose of the repository service is to reduce the number of network connections to the remote web server, there by reducing network traffic. When a remote page fetch is initiated, the repository service checks for the existence of the remote page in its cache and if present, the latest version of the page in the cache is returned. In cases of cache miss, the repository service requests that the page be fetched from the appropriate remote server.

Subsequent requests for the web page can access the page from the cache instead of repeatedly invoking a fetch procedure.

2.8 Presentation and Notification

Change presentation is the last phase of web monitoring where the detected changes are presented to the user. The presentation method selected should clearly show the detected differences between two web pages to the user. Therefore, computing and displaying the detected differences in a meaningful manner is very important. Based on the number of changes detected and the notification mechanism (email, PDS, fax) where the changes need to be displayed, we plan to use a heuristic cost model for choosing the presentation mechanism for displaying changes.

The user may want to be notified immediately of changes on particular pages. In such cases, immediate notification should be sent to the user. WebVigiL provides the semantics of *immediate* and *best effort* (section 5.4). Alternatively, frequency of change detection will be very high for web pages that are modified frequently. Since frequent notification of these detected changes will prove to be a bottleneck on the network, it is preferable to send notification periodically. In addition, WebVigiL provides the provision to the user to view the information when required. A user dashboard is provided for that purpose. The user can specify the desired notification interval in the sentinel. The WebVigiL server, based on the notification frequency can push the information to the user, thus propagating the “just in time”(JIT) paradigm.

2.9 Summary

The system incorporates the concepts of active paradigm for effective monitoring of web pages based on user profiles/policies.

CHAPTER 3

RELATED WORK

WebVigiL is a self-monitoring system, which allows for composite changes and notifies the user of changes as soon as it is detected. To achieve this capability, active technology can be effectively used. Active rules incorporated to make databases active has been well developed. Active database systems, based on rule definition, event detection, and action execution, add the additional functionality to recognize specific situations (external situations) and react to them in contrast to the traditional passive databases. Such databases not only detect composite events but also incorporate coupling and parameter modes for added capability. Active databases such as HiPAC, Sentinel, Ariel [10-12] have incorporated the active capability into the database. The usage of this capability has been very limited in web-based systems. WebVigiL uses active capability for the run time management of user-defined specifications to monitor and detect changes efficiently.

Many research groups have been working to address detecting changes to documents. GNU diff detects changes between two Unix and Linux files. Commercial products such as WordPerfect have a “mark changes” facility that can detect changes based on how documents can be compared (on either a word, phrase, sentence, or paragraph basis). Considerable work has been done investigating methods and techniques for detecting duplicated portions of code or portions of similar code in procedural software systems. But most of the previous work in changed detection has dealt only with flat-file and not for structured or unstructured web documents.

The adaptive push/pull [13] approach evaluates the effect of various approaches (push, pull, and combinations there-of) from the point of view of propagating changes from server to the client. Some change–monitoring tools such as ChangeDetection.com [14] have been developed using the push-pull paradigm. But these tools detect changes to the entire page instead of tracking changes to user specified components. The changes can be tracked only on limited pages and the user will be loaded with all the changes, irrespective of whether he/she requires it or not. Hence, these tools solve the problem of when to refresh a page but not how and when a page is modified. In addition, these tools have limited capabilities in terms of types of changes detected. The scalability of these systems is not very clear. Some tools have been developed, which allows detection of customized changes to web documents. The rest of the section discusses these tools.

3.1 Approaches for unstructured documents

Tools such as WebCQ and AIDE [15, 16] offer detection and notification of customized changes on the web pages. WebCQ detects changes based on user-profile. But it is not very clear whether, more than one type of change detection is supported for each request. WebVigiL provides the flexibility to detect multiple changes on a page.

[16] presents a set of tools (collectively called AT&T Internet Difference Engine), which view a HTML document as a sequence of sentences and sentence-breaking markups. A token is either a sentence-breaking markup or a sentence, which can be a sequence or words and non-sentence-breaking markups. AIDE uses HTMLdiff, which uses weighted LCS algorithm [17] to compare two tokens and computes a non-

negative weight reflecting the degree to which they match. All markups are represented and are compared. This approach may be expensive computationally as each sentence may need to be compared with all sentences in the document. Thus in situations where the user is interested in change to a particular phrase, HTMLdiff will end up computing change to the whole page, resulting in excessive computational cost. In addition, the mechanisms used for HTML change detection cannot be mapped directly to XML as XML is used to define content rather than presentation unlike HTML [3, 18]. Users can create their own tags in XML to define the context of the contents. Hence the use of tags as a sentence-breaking markup, as is used by AIDE [16] cannot be used in XML as contents between different tags denote different context.

3.2 Approaches for XML (semi-structured) documents

XML documents are semi-structured and contain well-defined tags, which define the text Hence an XML document can be represented as a tree. Tree to tree comparison to detect differences is a NP-hard problem but many algorithms have been proposed for tree-tree comparison taking some tree features into consideration. [19-21].

3.2.1 Changes detection in hierarchical structured documents

Chawathe et. al.[21] proposed algorithms for detecting changes in hierarchical structured data. The algorithm proposed for hierarchical structured data can be given as:

Given two trees T_1(old tree) and T_2(new tree) , a minimum cost edit script is generated , which gives a sequence of edit operations such as insert, delete, update and move, which transforms one tree to another. The definition of the edit script is : Given a sequence of edit sequence E= e_1..........e_m of edit operations, T1 $\xrightarrow{E}$ Tm+1, if there exists T2 ,…Tm such that T1 $\xrightarrow{e1}$ T2 $\xrightarrow{e2}$ …….Tm+1. A sequence of edit operations transforms T1 $\overset{E}{\rightarrow}$ T2 then T1 is considered isomorphic to T2. A matching function "compare" finds the matching nodes in T1 and T2 after satisfying the "equal" function and includes them in the matching M nodes set. This equal function is given as:

1. For leaf nodes $x \in$ T1 and $y \in$ T2, $(x,y) \in$ matching only if labels $l(x) = l(y)$ and compare$(v(x),v(y)) \leq f$ for some parameter f such that $0 \leq f \leq 1$.
2. For internal nodes, if x is the internal node in T1 and y is the internal node in T2, then $(x,y) \in M$ if $l(x) = l(y)$ and $|\text{common}(x,y)|/\max(|x|,|y|) > t$ for some t satisfying $½ \leq t \leq 1$. common(x,y) is defined as: $\{(w,z) \in M |$ x contains w and y contains $z\}$.

To ensure that the tree is aligned using minimum number of moves, the longest common subsequence (LCS) method is used. The LCS given by [22] is used. The LCS is given three inputs: two sequences S1 and S2 to be compared and an equal(x,y) function to compare the sequence. This algorithm successfully detects inserts, deletes, updates and move on the trees.

This algorithm works for hierarchical structured documents such as latex. But the assumption may not hold good for XML documents as they contain duplicate nodes and sub-trees.

3.2.2 Fast Change Detection for unordered XML documents

X-diff [23] is a fast change detection algorithm for XML, which takes advantage of the structure and label given to all XML element nodes. X-diff detects edit operations such as insert, delete and update on parsed unordered label tree of XML documents.

Given two unordered tree T_1 and T_2, X-diff finds the equivalent second level sub-trees by comparing their XHash values. XHash is computed using a general hash function such as MD5 [24] or SHA [25] on the entire sub-tree. For unmatched sub-trees, signature is computed for every node. Signature of a node is obtained by concatenating the names of all its ancestors with its own name and type. Using dynamic programming, the nodes are compared for matches and matching nodes are extracted. This algorithm detects inserts, deletes and updates to a node.

For this algorithm, the basis assumption is that XML is unordered tree but according to the definition [18], XML is an ordered tree except the attributes, which are unordered. For a tree rooted at R with children p_i to p_m, if a node along with the complete path is moved from j where i<=j<=m to k in tree T2 , where k≠ j, such changes cannot be detected in an unordered tree.

3.2.3 Change Detection in ordered XML documents

Algorithms have also been proposed for XML documents represented as ordered labeled trees. Cobena et. al. [26] have formulated a change detection algorithm called Xydiff to detect changes between two XML documents for the Xyleme project. XyDiff algorithm computes the changes by computing the four phases:

1. Compute Signature and order sub-trees by weight: Given two ordered tree T_1 and T_2, a signature is computed. The signature is computed by computing the node's content and its children. Thus, the weight is calculated by taking the size of its text nodes and sum of its weight of its children for element nodes. The weight of the text node is: 1+ log(length(text)) and weight of the element node is : 1+ sum(weight(children)). A priority queue is used and each sub-tree is inserted by its weight, the heaviest being the first.

2. Trying to find matching starting from heaviest nodes: Starting from the heaviest nodes, the sub-trees are matched by their signature. The best match is the one in which the nodes match and the parent matches the reference node's parent. When a

candidate is accepted as matched, the pair of sub-trees and ancestors is matched as long as they have the same label. If there are no matches and the node is an element, the children are added to the queue.

3. Use structure to propagate matching: The tree is traversed top-down and the nodes are matched from the old and new documents as long as the nodes have the same label and their parents match.

4. Compute the delta: All the non-matching nodes are denoted as inserts/deletes respectively depending whether they are in the new documents or the old documents. All nodes, which are matching but with non-matching parents are termed as moves. For the matched sub-trees, the additional moves are detected by computing the largest order preserving subsequence.

The change detection is done at the node level. Hence, when changes have to be detected on part of the node or when content to be monitored exceed a node, this method cannot be used without modifications.

3.3 Other approaches for XML change detection

XMLTreeDiff [27], a tool developed by IBM, is a set of Javabeans and does ordered tree to tree comparison to detect changes between XML documents. It generates a hash values using DOMHash [28] function and filters the identical sub-trees based on equal values of the DOMHash. XMLTreeDiff uses an optimal tree-differentiating algorithm together with a fast subtree matching procedure to detect changes between given two XML DOM trees.

XMLDiff [29] is another tool for detecting changes between XML documents, developed by the Logilabs. It is a python based tool for detecting differences between similar XML documents and was developed as part for the Narval project. It computes both structural changes and changes to the leaf nodes and detects changes in terms of insert-after, rename, move, update, append etc.

Though most of the described tools detect changes between two XML documents, these algorithms rely on the fact that either an entire node can change or there is no change. WebVigiL is a system, which monitors changes to sections of the content of a page, which in terms of tree can be part of the node or can exist as part of more than one node. Hence, the given algorithms cannot be mapped directly to satisfy the monitoring requirements of WebVigiL.

3.4 Approaches for user specification

Present day users are interested in monitoring changes to pages and want to be notified based on his/her profile. Hence, an expressive language is necessary to specify user-intent on fetching, monitoring and propagating changes.

WebCQ [15] detects customized changes between two given HTML pages and provides an expressive language called sentinel for the user to specify his/her interests. But WebCQ only supports changes between the last two pages of interest. Hence compare options is not provided to the user. The semantic correctness for the fetch and notification specifications only need to be verified unlike in WebVigiL, which needs to verify the correctness of the combination of fetch, notification and compare options. In addition, inheritance of properties or lifespan of correlated sentinels are not supported as in WebVigiL, which is able to support these properties because of the use of active paradigm.

In [30], the authors allow the user to submit monitoring requests and continuous queries on the XML documents stored in the Xyleme repository. WebVigiL supports only time- based monitoring request and not continuous queries. To the best of our knowledge, customized changes, inheritance, different reference selection or correlated specifications cannot be specified in the monitoring request on Xyleme.

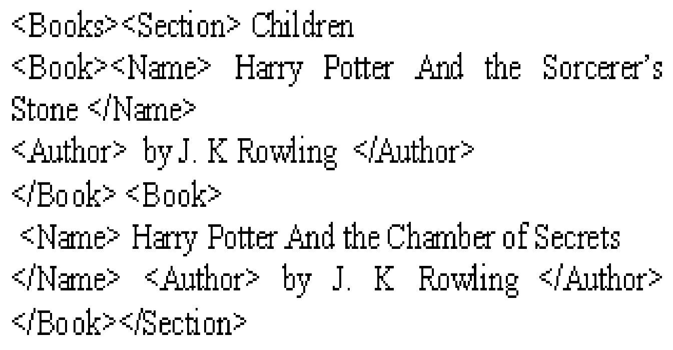
3.5 Summary

WebVigiL supports customized changes based on user-intent. Hence, the existing methods cannot be mapped for WebVigiL. In addition, as WebVigiL provides lot more flexibility to the user, an expressive semantic language needs to be devised.

CHAPTER 4

CHANGE DETECTION IN XML DOCUMENTS

WebVigiL is targeted for large network-centric environment such as the Internet. We support customized change detection to Hypertext Markup Language (HTML) and eXtensible Markup Language (XML), which are the standard formats for electronic publishing on the web.



```
<Books><Section> Children
<Book><Name> Harry Potter And the Sorcerer's
Stone </Name>
<Author> by J. K Rowling </Author>
</Book> <Book>
 <Name> Harry Potter And the Chamber of Secrets
</Name> <Author> by J. K Rowling </Author>
</Book></Section>
```

Figure 4. 1 Part of an XML document

In section 1.3, we introduced and explained the eXtensible Markup Language (XML). XML is semi-structured and differentiates between presentation and content unlike HTML which is unstructured. The tool designed to detect changes to XML must consider the semi-structured nature of XML documents and be able to detect customized changes to the document based on user-intent.

As explained in section 1.3, the user-defined tags known as elements, define the context in XML. Elements consist of attributes and the text value, which describe the properties and value of the element. Hence, for detecting changes to XML documents, the nodes such as elements, attributes and text nodes are of interest. As shown in Figure 4. 1, the content "Harry Potter And the Chamber of Secrets" is defined as the name of a book in the books section by the elements 'Books', 'Section', 'Book', and 'Name'. From the above example and the definition given in [18], it is clear that XML documents are semi-structured.

Section 4.1 describes the ordered labeled XML tree. In section 4.2, we will describe why customized change detection should be supported followed by the problems faced in detecting these changes in XML in section 4.3. Section 4.4 discusses the various design alternatives. In section 4.5, we define changes and the change detection modules will be discussed in detail in section 4.6.

4.1 XML as an ordered labeled tree

A document written in XML can be mapped into a hierarchal tree. In XML, the elements are ordered while the attributes are unordered [18]. Each node x in XML consists of a label l(x) and content denoted as value v(x). The ordered XML tree for the text shown in Figure 4. 1 is shown in Figure 4. 2.

As shown in the Figure 4. 2, the root (section 1.3) of the tree is 'Books'. The root is divided into elements, which define the main context of the documents. The parent element node connects with its corresponding child node, which could be elements, attribute or text nodes. The text and attribute are the leaf nodes of the tree. The text node contains the content/value but have a common label as 'Text'. The attributes nodes contain both label and value. According to [18], the attributes are considered unordered in XML. But as the elements and text nodes are considered ordered for the tree, for consistency in change detection, the attributes are also assumed to be ordered.

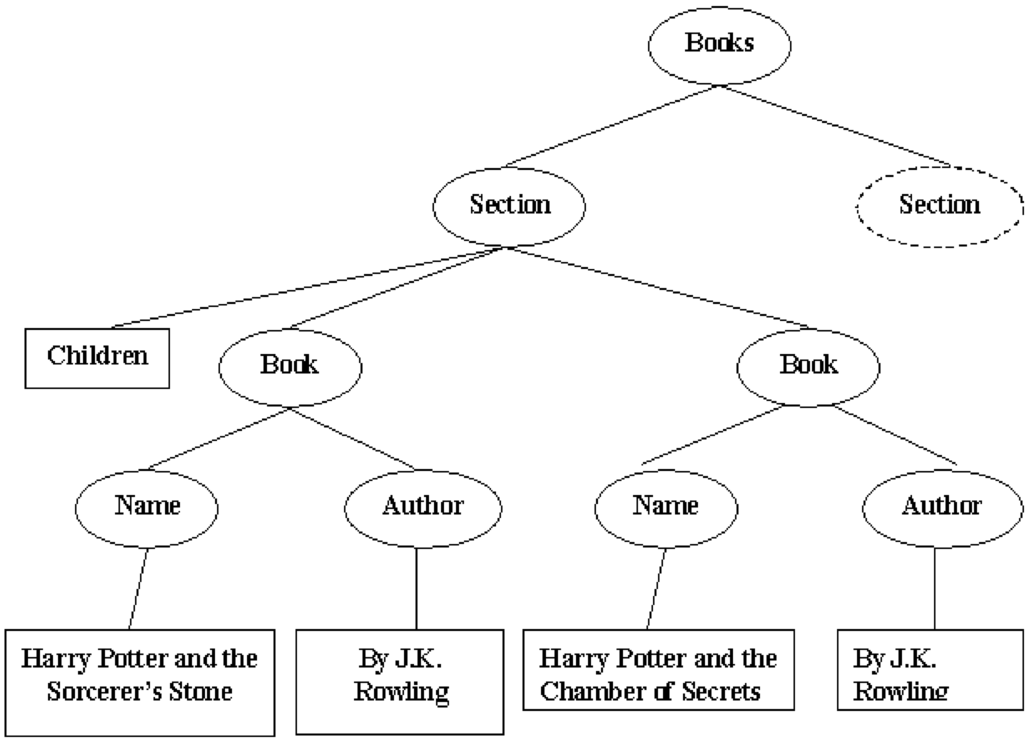


Figure 4. 2 Ordered Labeled XML tree

4.2 Importance of Customized Change Detection

The Internet is evolving as a repository of information, and the user's interest has expanded from querying information to monitoring the evolution of the pages. The ability to specify changes to arbitrary documents should be coupled with dissemination of information properly without resulting in wasteful computation and waste of resources due to manual polling or querying the information.

The emphasis is on selective change detection, as the users are typically not interested in changes to the entire page but to a particular portion or section. For example, a student may be interested in knowing whether a particular course is offered in the semester and would like to monitor changes on the keyword ‘coursename’. Similarly users may be interested in monitoring a particular new item (e.g. keywords, phrases) in the news web page or particular stocks in the stock web page. Change detection on the entire page will be an overkill and computationally costly in this case. Hence, the change mechanism should be capable of not only detecting changes to the entire page but to objects of user interest such as keywords, phrases etc. WebVigiL monitors customized changes such as changes to a phrase and keywords on web pages based on user-intent.

In addition, it has been observed that the user’s interest pertains to the content and not to the context (structure), which is hidden from the user. Detecting changes to the structure may not only increase the computational cost but may not achieve any purpose for the end user. Hence, WebVigiL supports changes only to the content of a document. But the context (structure/tags) information can be used for efficient change detection as the context defines the content. Hence, to detect changes, the content in a particular context should be matched with similar content in the same context.

4.2.1 Types of changes supported

WebVigiL supports customized changes such as any change, all words, link change, image change, table change, list change, keyword change, phrase change and regular expression. These changes can be divided into word-based changes and presentation based changes. Changes such as any change, all words, keyword change and phrase change constitute the word-based change while the rest can be defined as the presentation based changes. As XML documents separate the presentation from the content, the presentation-based changes can only be detected by additional information from the style sheet or other presentation documents or the document type definition (DTD). At present, WebVigiL supports changes on only a single page level. Hence, currently the presentation-based changes are not detected for XML pages.

The change detection approach for XML called the CX-Diff supports changes to the content such as any change, phrase change and keyword change. CX-Diff does not detect structural change to the document but only detects changes to the content. In an XML tree, the leaf nodes represent the content. Hence changes to the leaf nodes are of interest. The element information is extracted to take advantage of the semi-structural nature of the XML documents and detect efficient changes with respect to position and context. As XML is ordered, the changes detected are position dependent.

Taking the above definitions into consideration, the changes can be defined at the page-level and tree level as:

Keyword change: Appearance or disappearance of unique word(s) of user's interest, in an XML page. In terms of tree definition, the word could be the leaf node or part of the leaf node. For example, as shown in Figure 4. 2, keyword 'Harry' is part of the entire node 'Harry Potter and the Chamber of Secrets'.

Phrase: Appearance or disappearance of contiguous set of objects defined by the user, which could be a leaf node, part of a leaf node or can span across multiple node. For example, as shown in Figure 4. 2, the phrase 'the Sorcerer's stone by J.K. Rowling' spans across multiple nodes.

Any Change: Appearance, disappearance or move (defined in section 4.5) of any of the leaf nodes (i.e. complete content of a node) are tracked and detected as a change.

Any Change	Appearance/disappearance /realignment of leaf nodes
Phrase Change	Appearance/disappearance /update of a given phrase
Keyword Change	Appearance/ disappearance of given keyword

Figure 4. 3 Synopsis of changes

The synopsis of these changes is shown in Figure 4. 3. As apparent from the synopsis, move is only detected for any change as move defines realignment of complete nodes between given two trees.

4.3 Problem Overview

The changes need to be captured on contents constituting part of the node or spanning several nodes and considering the context and order of occurrence of the leaf node in the page in an ordered XML labeled tree. Change detection for semi-structured, ordered XML tree is complex because of the following issues:

Duplicate Nodes: XML contains duplicate nodes. By duplicate nodes, we mean similar leaf nodes containing the same context. As shown in Figure 4. 2, the node 'J K Rowling' appears twice in the tree for the same context (i.e. 'Books-Section-Book-Author'). Duplicate sub-trees defined for the same context are also possible in XML. Order becomes very critical for such duplicate nodes as a node *n*, existing at position p_i in the old tree should be compared to the node existing in the equivalent i^{th} position in the new tree with respect to their siblings

Realignment of nodes: Two XML documents may contain the same content having the same structure but the nodes may be realigned in different sub-trees or with respect to the siblings. For example, for a tree T_1 rooted at R with children p_i to p_m, a node along with its structural information can be moved from j where $i \leq j \leq m$ in T_1 to position k in T_2 where $j \neq k$, when considered with respect to the siblings. The change mechanism developed should be capable of detecting such move operations.

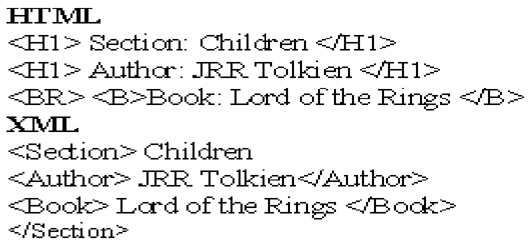
Customized changes: WebVigiL supports customized change detection to the contents, such as phrase and keyword change. Keywords and phrases can be part of the node or can span multiple nodes. Detecting changes to contents constituting part of the node or spanning several nodes complicates the extraction of the content and change detection. Hence, the algorithm should be capable of extracting the required content of interest and detect changes.

The change detection tools discussed earlier in section 3.2 for XML documents do not handle customized changes. The proposed algorithms for HTML are for unordered trees. Hence, an algorithm is proposed, taking into consideration an ordered, labeled XML tree and the position of occurrence of the node with respect to its sibling.

4.4 Design Issues in XML

1) *Adopting approaches for HTML*: There are two fundamental differences between XML and HTML as apparent from Figure 4. 4:

- Separation of form and content: HTML mostly consists of tags defining the appearance of text. Hence, usage of tags in HTML is for presentation purpose. In XML, the tags generally define the structure and content of the data. A specific application or an associated style sheet specifies the presentation for XML. Hence, the content and presentation for XML is separate.
- XML is extensible: In XML, users or organizations can define tags.



```
HTML
<H1> Section: Children </H1>
<H1> Author: JRR Tolkien </H1>
<BR> <B>Book: Lord of the Rings </B>
XML
<Section> Children
<Author> JRR Tolkien</Author>
<Book> Lord of the Rings </Book>
</Section>
```

Figure 4. 4 Difference in XML and HTML

These differences can be clearly deduced from the example given in Figure 4. 4. The documents contain information about a children's book. The tags in XML such as section, author etc. defines the context of the contents. But in HTML, the tags such as H1, BR etc. define the presentation.

As the format and representation of both HTML and XML documents are different, the proposed approaches for HTML change detection cannot be adopted for XML.

2) *Considering XML as unordered tree*: As per the definition of XML, the elements in XML are ordered. [18]. In addition as described in section 4.3, XML documents contain duplicate nodes. The assumption that the tree format of XML as unordered tree is not correct. In addition, only by considering XML as an ordered tree, moves (section 4.3) can be efficiently detected. Hence, a change detection algorithm considering XML as an ordered tree needs to be proposed.

3) *Customized changes to content rather than structure*: WebVigiL detects customized changes to contents such as keyword change, phrase change etc. as discussed in section 4.2. Hence the proposed change detection algorithm must be capable of detecting changes not only to the content leaf nodes but also to content that are part of the node or contents spanning multiple nodes. In this case, detecting changes

to the structure will result in unnecessary computation, which will make the algorithm computationally expensive.

4) *Assigning unique Ids to the leaf nodes*: Each leaf node can be assigned a unique id depending on the position of occurrence and nodes can be compared on basis of the basis of unique ids. But between two given XML documents, due to inserts and deletes, comparing two nodes based on their ids will not be sufficient. Hence, the position of occurrence of the leaf nodes needs to be considered relative to the value of the sibling instead of just position.

4.5 Change Operations

The change detection for XML defined as CX-Diff, detects changes between two ordered labeled XML trees. The changes are detected by considering the change operation, which transform a tree T_1 to T_2. Given two ordered XML trees T_1 and T_2, consider the change operations from the set E = {insert, delete, move} which when applied to T_1 transforms it into a new tree T_2. To detect the change operations, the structure (element information) is also taken into consideration. The content of a leaf node is defined as its *value* and is denoted as v(x) where x is a leaf node. All the element nodes pertaining/associated to a leaf node define the context of the leaf node. All the element nodes, which constitute the path of the leaf node, are extracted and this additional structural information is used for effective change detection.

The structural information denoted as path or signature is defined as:

Definition 1: *Signature*: - The ancestral path of a leaf node from the parent to the root, denoted by path(*x*) for node *x*. For attributes, the label of the attribute also becomes a part of the signature. For example in Figure 4. 2, the signature for the node

"Harry Potter and the Chamber of Secrets" is Books-Section-Book-Name. For Figure 4. 5, the signature of the node having value 'D' is 'R-S-P'.

Given two trees T1 and T2, if leaf node x ∈ T1 and leaf node y ∈ T2, v(x), v(y) denotes the value of the node x and y respectively. The labels of the leaf nodes x and y are represented as l(x) and l(y) and signatures are denoted as path(x) and path(y). Given these properties, the change operations can be defined as follows:

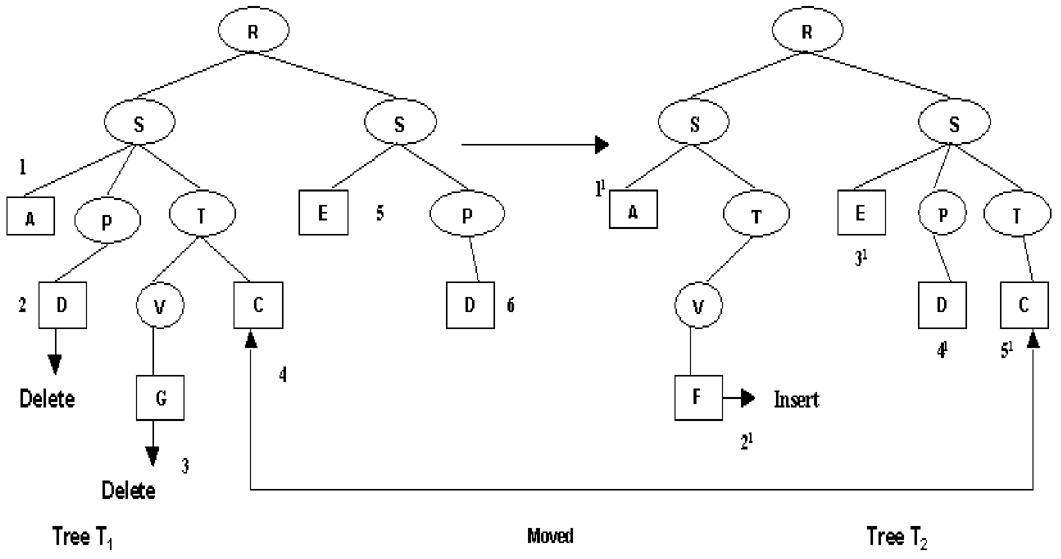


Figure 4. 5 Change Operations on trees T1 and T2

Definition 2: Insert:- Insertion of a new leaf node at the ith position is denoted by insert (v(x),i). If $n_1 \ldots n_m$ are the leaf nodes in T_1 and for $1 < k < m$, if $n_1 \ldots n_{k-1}, x, n_k \ldots n_m$ are the leaf nodes in T_2, then the node x is considered inserted. As structure defines the context for the content in XML, a node of the same value but different signature is considered inserted. Insert of a keyword is defined as the appearance of a keyword k in the *i*th leaf node x of the tree T_1, denoted by insert_keyword (k,x,i) where the keyword can be part of the leaf node x or the node itself. Insert of a phrase is defined as appearance of a complete phrase at position *i* in the tree T_1, denoted by (p,i).

Definition 3: Delete: - The deletion of an old leaf node at the i^{th} position is denoted by delete(v(x),i). A leaf node *x* having the value v(x) is deleted from the i^{th} position in tree T_1 rooted at root R. Given two ordered XML trees T_1 and T_2, T_1 will be same as T_2 except that it will not contain x. Delete of a keyword is defined as the disappearance of the keyword k in the i^{th} leaf node x of the tree T_1, denoted by delete_keyword (k,x,i). Delete of a phrase is defined as the disappearance of a phrase *p* at i^{th} position in the tree T_1, denoted by (p,i).

Definition 4: Move:- For the tree T_1 , containing leaf nodes from n_1 to n_m, a leaf node x containing signature s is shifted from position j in T_1 to position k in the new tree T_2 where $1<=j<=m$ and $j \neq k$ with respect to the siblings. Move is denoted as move (x,j ,k) where x is the leaf node moved from position j to position k. Move is only applicable to a complete node. Keyword and phrase changes are changes detected to part of the node or on the contents of more than one node. Hence, move is not applicable to keyword and phrase change but only for any change on the leaf nodes.

As shown in Figure 4. 5, leaf nodes having value ‘D’ and ‘G’ are deleted in tree T_1 in position 2 and 3 respectively and leaf node with value ‘F’ is inserted in tree T_2. Leaf node ‘C’ is moved from position 4 in Tree T_1 to position 5 in T_2.

4.6 CX-Diff: Customized Change Detection for ordered XML documents

XML contain different kinds of nodes as described in section 1.3. As we are interested in changes to only the content, the leaf nodes consisting of text and attributes are of interest. But these changes have to be detected in terms of the correct context. Hence element nodes are also important for structural information for efficient detection. We process only these three kinds of nodes and do not currently process DTD, CData, Entity and Processing Instructions nodes.

In XML, the elements are considered ordered. Hence the content or value of the element represented by text nodes is also ordered. Though according to the XML definition, the attributes are not considered ordered, but as the changes are detected considering the content to be ordered, the attributes are assumed ordered for the proposed change detection algorithm. Attributes defining ID and IDREFS are also considered as simple ordered attributes.

The matching nodes in an ordered tree, should satisfy the following conditions:

- Ancestor Order preservation
- Must satisfy some defined function equal
- Sibling order preservation

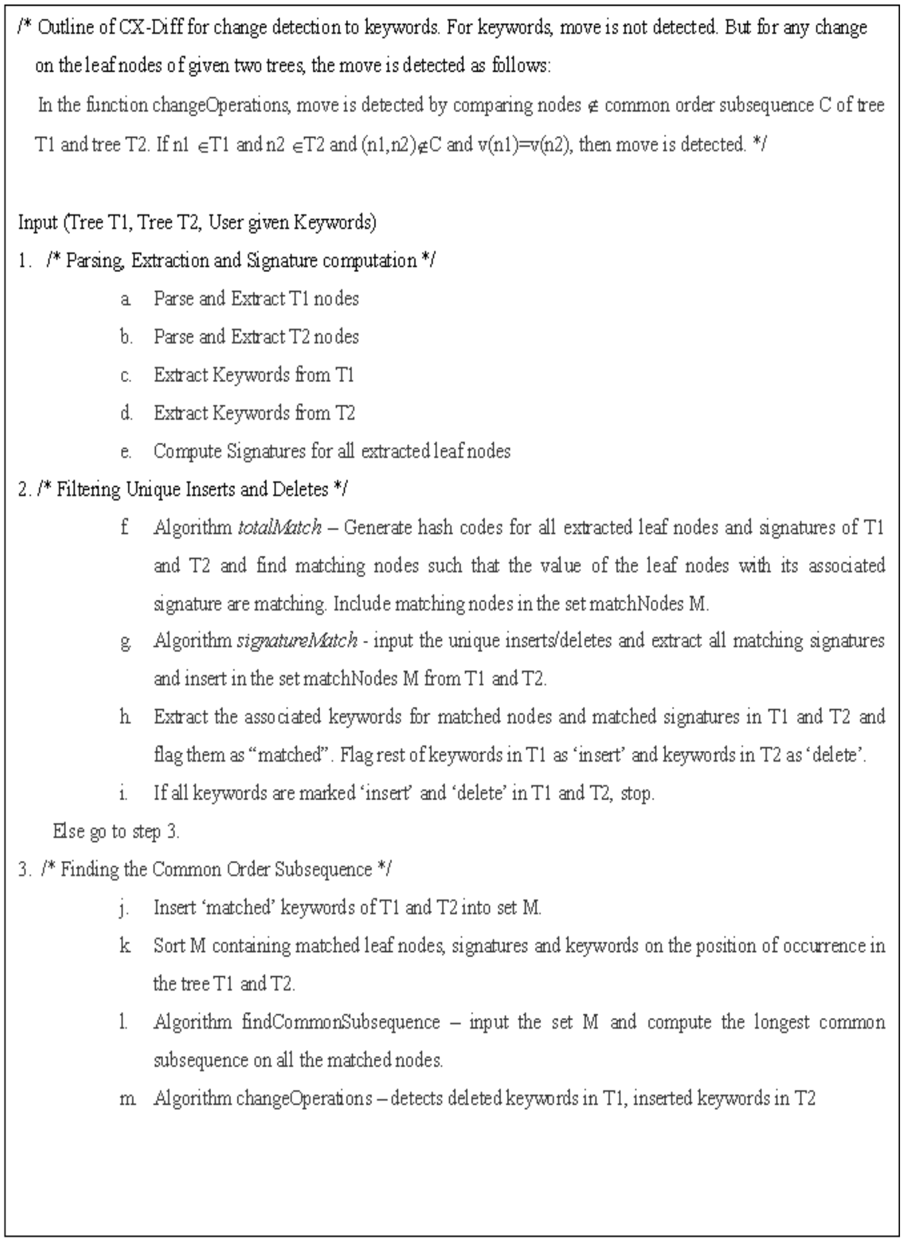
Definition 5: For an ordered, semi-structured tree, the best match for ordered leaf nodes are the ones satisfying the following:

1. For $\forall(x,y) \in M$, ,if $x_p \in$ path(x) and $y_p \in$ path(y) where p is the position in the signature list, then x_p is the ancestor of x iff y_p is the ancestor of y and $x_p = y_p$. (ancestor order preservation)
2. For $\forall(x,y) \in M$, iff $v(x) = v(y)$
3. For $\forall(x1, y1) \in M$, iff $x1, y1 \in$ common order subsequence L such that x1 has the same order of occurrence in L as y1.

According to [3, 4]in a document object model (DOM), by which an XML document is converted into a tree, the root contains all the other nodes in the tree. Based on this and definition 5, the following assumption is considered valid:

Assumption 1: For given trees T1 and T2, if Root(T1) $\neq$ Root(T2), the match set $M=\phi$

The assumption holds true because if the root of the tree do not match, then the first condition of definition 5 will never be satisfied.



```
/* Outline of CX-Diff for change detection to keywords. For keywords, move is not detected. But for any change
   on the leaf nodes of given two trees, the move is detected as follows:
   In the function changeOperations, move is detected by comparing nodes ∉ common order subsequence C of tree
   T1 and tree T2. If n1 ∈T1 and n2 ∈T2 and (n1,n2)∉C and v(n1)=v(n2), then move is detected. */

Input (Tree T1, Tree T2, User given Keywords)
1.  /* Parsing, Extraction and Signature computation */
        a. Parse and Extract T1 nodes
        b. Parse and Extract T2 nodes
        c. Extract Keywords from T1
        d. Extract Keywords from T2
        e. Compute Signatures for all extracted leaf nodes
2. /* Filtering Unique Inserts and Deletes */
        f. Algorithm totalMatch – Generate hash codes for all extracted leaf nodes and signatures of T1
           and T2 and find matching nodes such that the value of the leaf nodes with its associated
           signature are matching. Include matching nodes in the set matchNodes M.
        g. Algorithm signatureMatch - input the unique inserts/deletes and extract all matching signatures
           and insert in the set matchNodes M from T1 and T2.
        h. Extract the associated keywords for matched nodes and matched signatures in T1 and T2 and
           flag them as "matched". Flag rest of keywords in T1 as 'insert' and keywords in T2 as 'delete'.
        i. If all keywords are marked 'insert' and 'delete' in T1 and T2, stop.
   Else go to step 3.
3.  /* Finding the Common Order Subsequence */
        j. Insert 'matched' keywords of T1 and T2 into set M.
        k. Sort M containing matched leaf nodes, signatures and keywords on the position of occurrence in
           the tree T1 and T2.
        l. Algorithm findCommonSubsequence – input the set M and compute the longest common
           subsequence on all the matched nodes.
        m. Algorithm changeOperations – detects deleted keywords in T1, inserted keywords in T2
```

Figure 4. 6 Outline of the algorithm for keywords

For customized change detection based on user-intent, extraction of the objects of interest such as keywords and phrases is necessary to detect changes to a page. Signature is computed for each extracted leaf node. To detect change operations between given trees T_1 and T_2, the unique inserts/deletes are filtered and matching nodes and signatures are extracted. The common order subsequence is detected on the extracted matching nodes to detect move and insert/deletes to duplicate nodes.

The algorithm consists of following steps: i) object extraction and signature computation, ii) filtering of unique inserts/deletes and iii) finding the common order subsequence between the leaf nodes of the given trees. For reducing the computational time for detecting changes, an optimization is also proposed.

4.6.1 Object Extraction and Signature Computation

Based on the user-intent, the object of interest needs to be extracted from the contents of the XML document and the structural information derived by computing the signature. To access the content and extract the structure of the XML document, it is first transformed into a Document Object Model (DOM) and the value of the leaf nodes is extracted and their signature is computed from the element information.

For keyword extraction, the keywords are matched with the value of the leaf node. If the keyword is found, it is extracted along with the order of occurrence and its associated signature is computed.

Phrase can span multiple nodes. Hence, once the phrase is detected, the part of the nodes containing the phrase is extracted as a new tree. The old tree is realigned and the new tree is attached as a sub-tree in the original tree at the proper position. In Figure 4. 7, for the phrase “the Sorcerer’s Stone by J.K.Rowling’, the part of the phrase is extracted and realigned.

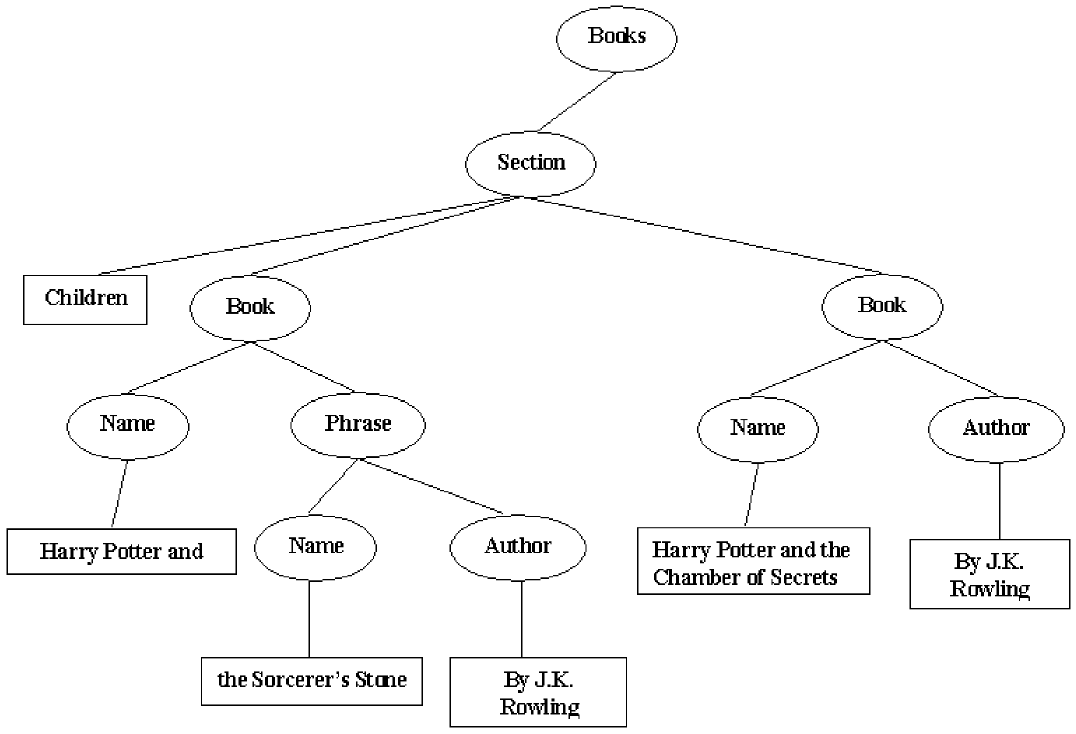


Figure 4. 7 Extracted phrase

As shown in Figure 4. 2,"the Sorcerer's Stone" is part of the node value "Harry Potter and the Sorcerer's stone" while "by J.K. Rowling" is a separate node. Once the string-matching algorithm detects the phrase, the range is set. The start index in this case will be the 4th word of the node at position 2 i.e. "Harry Potter and the Sorcerer's stone ". The end index is placed at the end of the node at position 3. The appropriate part of the nodes are extracted and inserted as new node in the tree with inserted "phrase" node differentiating it as a phrase. The rest of the nodes are realigned as shown in Figure 4. 7. The signature for the extracted phrase "the Sorcerer's stone by J.K.Rowling" will be 'Books-Section-Book-Name-Author'.

For the given tree T_1 and T_2 in Figure 4. 5, the tree is traversed and the leaf nodes and their associated signatures are extracted and added to the T_1set and T_2set respectively as shown in Figure 4. 8.

4.6.2 Filtering Unique Inserts/Deletes

In a given tree T, a node x containing value v(x) can be distinct or can have multiple occurrences. Insertion/Deletion of distinct nodes can result in *unique insert/delete* unless they are moved, and can be detected on an unordered tree. Similarly, leaf nodes containing non-matching signature can also be considered as unique inserts/deletes as the signature define the context.

Definition 6: For each leaf node x in tree T1, if there is no matching node y in tree T2 such that v(x) = v(y) or path(x) = path(y), then x is a unique insert. For a set matching M for old tree T1 and new tree T2:

$M(x,\emptyset)$ where $x \in T1$ = Unique insert

$M(\emptyset,y)$ where $y \in T2$ = Unique delete

To reduce the computation cost of finding the common order subsequence between two ordered trees, by considering all the leaf nodes, the unique inserts/deletes are filtered out and matching nodes extracted by the defined functions totalMatch and signatureMatch.

totalMatch algorithm: For each extracted node, the function totalMatch(old_tree, new_tree) extracts the set of best matches denoted as M such that for the given trees T1 and T2 and leaf node x in T1 and leaf node y in T2, $(x, y) \in M$ if v(x) =v(y) and path(x) = path(y).

The value of the node along with the signature is mapped to a unique value for both the trees and compared. If a match is found, then it is inserted into the matched set.

For a node n, if no match is found, then according to the definition 6, it is flagged as ‘insert’ or ‘delete’. For phrase change, the associated phrase for each node is also marked as ‘insert/delete’. To determine inserts/deletes for keywords, further processing is needed. For some cases, though the value of the leaf node do not match, as keyword can be part of the leaf node, instances of the keyword in the leaf node may be matching. In order to detect such matching, we compare the signature of the node. If the signature matches, the instances of the keyword in the leaf node are also considered matched. To detect nodes containing common signature, the signatureMatch algorithm (defined below) is used.

In addition, as XML is well-defined document, it can be assumed that the structure is generally stable. Hence, many times, though the contents change, the structure remains the same and this information can be included for optimal detection of common order subsequence between two trees. The non-matching nodes of totalMatch algorithm are given to the function signatureMatch, to extract common signatures.

signatureMatch algorithm: All the matching signatures in the old and new tree, containing non-matching leaf nodes are included in the set M. For leaf node x in tree T_1 and y in tree T_2, if path (x) = path (y) and $v(x) \neq v(y)$, then path(x) and path(y) are included in the match set M.

The distinct leaf node having value ‘G’ in Tree T_1 and node having value ‘F’ in Tree T_2 in Figure 4. 5 are detected as deleted and inserted after computation of the function totalMatch as shown in Figure 4. 8. Though value of leaf nodes ‘G’ in T_1 and ‘F’ in T_2 do not match but their signatures match i.e. path(G)=path(F). Hence, for efficient computation of common order subsequence, the common signature information is extracted by the function signatureMatch and included as elements in the matchedT1set and matchedT2set. As shown in Figure 4. 8, at the end of phase I, all unique inserts i.e. ‘F’ and unique deletes i.e. ‘G’ are detected and common structural information of such unique inserts/deletes are extracted.

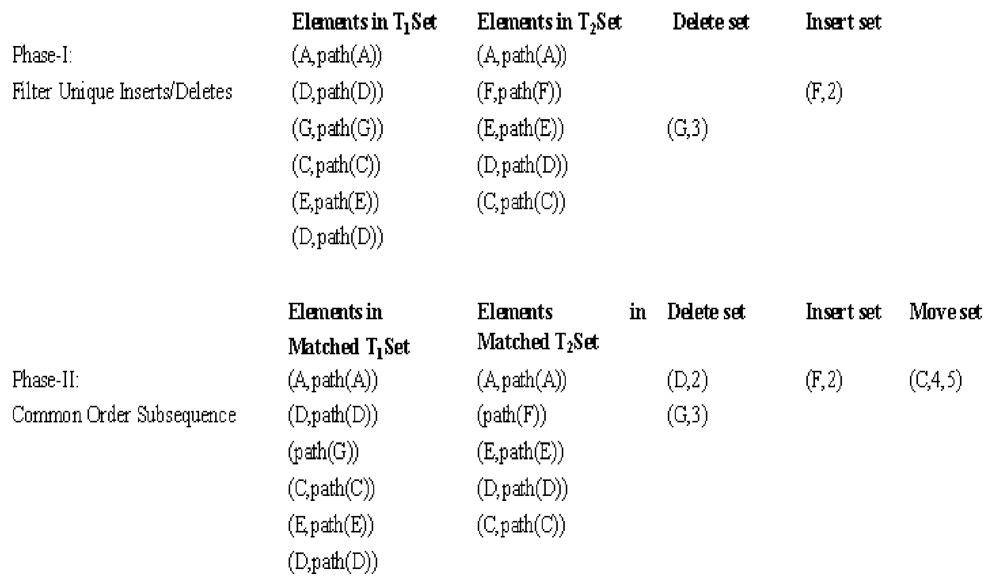
For keywords and phrase change, if all the extracted keywords and phrases result in unique insert/delete, then the computation can be considered complete at this stage.

4.6.3 Finding the Common Order Subsequence

For change detection to multiple occurrences of a node with common signatures and for moved nodes, it is necessary to consider an ordered tree. As per definition 5, apart from signature match and equal values of the leaf nodes, the best match should satisfy the 3rd condition i.e. the node must belong to the common order subsequence extracted from the two trees. Due to realignment of the node and inserts and deletes, the order of occurrence needs to be considered with respect to the value of the sibling. The common order subsequence is computed by finding the Longest Common Subsequence (LCS) [17] between the matched nodes of both the trees.

All the matched nodes are aligned in the order of occurrence. For keyword change, the extracted keywords, which are part of the leaf node, are also aligned with its matching leaf node and signature. For the matching signatures extracted from signatureMatch, the instances of the keyword/s along with the signature is aligned for LCS. As explained in section 4.6.1, the phrase is inserted as a text node and hence is treated as a complete leaf node. For detecting LCS, each node is mapped into its equivalent hash code and the nodes resulting in the common order subsequence are extracted.

The nodes, which do not constitute the common order subsequence between the given two trees, are differentiated as inserts, deletes or moves. At the end of this phase, all the moved nodes and the duplicate inserts/deletes will be detected. For example, in Figure 4. 2, if ”J.K Rowling” at 3rd position is deleted, the delete will be detected for correct position. Similarly, at the end of the LCS computation on the matched nodes in Figure 4. 5, the deletion of the node ‘D’ at position 2 in T_1 as well as the move of node ‘C’ from position 4 in T_1 to position 5 in T_2 can be detected. Hence, this algorithm detects customized changes such as keywords, phrases etc based on user-intent. In addition, changes to duplicate leaf nodes containing common structural information and moves are accurately detected.



	Elements in T_1Set	**Elements in T_2Set**	**Delete set**	**Insert set**
Phase-I:	(A,path(A))	(A,path(A))		
Filter Unique Inserts/Deletes	(D,path(D))	(F,path(F))		(F,2)
	(G,path(G))	(E,path(E))	(G,3)	
	(C,path(C))	(D,path(D))		
	(E,path(E))	(C,path(C))		
	(D,path(D))			

	Elements in Matched T_1Set	**Elements in Matched T_2Set**	**Delete set**	**Insert set**	**Move set**
Phase-II:	(A,path(A))	(A,path(A))	(D,2)	(F,2)	(C,4,5)
Common Order Subsequence	(D,path(D))	(path(F))	(G,3)		
	(path(G))	(E,path(E))			
	(C,path(C))	(D,path(D))			
	(E,path(E))	(C,path(C))			
	(D,path(D))				

Figure 4. 8 Phases of CX-Diff algorithm

For phrase change, update to a phrase can also be detected. After computing the common order subsequence, if a phrase in tree T_1 is not matched, then using the signature information, the leaf node values of the element constituting similar signature in tree T_2 can be extracted. Using LCS, the common subsequence can be extracted from the node values. A value *f* can be defined such that if the common subsequence is greater than or equal to *f*, then the phrase can be considered updated and non-matching value in the phrase can be extracted. If the common subsequence is less than the value *f*, then the phrase can be termed as deleted. The user or the system can determine the value of *f*.

4.6.4 Optimization

To improve the time taken by the above algorithm, an additional phase of eliminating common second level sub-tree is introduced. Sub-trees are computed at the second level as the second level defines the main context of the contents in the document. For given trees T_1 and T_2, the second level element node is denoted as *l(s)* where l is the label of node s. if l(s1) is the second level node of T_1 and its equivalent node in T_2 is l(s2), the sub-trees of T_1 and T_2 are considered matched if l(s1) = l(s2) and all the leaf nodes along with the signature in T_1 is equal to the leaf nodes and their associated signature in T_2 in the same order of occurrence. All the nodes of the matched sub-trees are removed from the matched set M. Hence, the size of M for LCS is reduced and the cost of computation is improved. But accurate results cannot be achieved if the sibling information is lost. Hence this optimization trades computation time for accuracy. Our experience has indicated that doing LCS at the 2^{nd} level does not affect the accuracy of change detection except in very rare cases. Currently, WebVigiL includes optimization as default for change detection to XML documents. In future, we plan to allow the user to decide whether accuracy of change detection or time is important. Based on the user policy, the decision to utilize the optimization technique will be made.

4.7 Summary

CX-Diff is an approach proposed for detecting customized changes to ordered, semi-structured documents such as the XML. Though this algorithm has been applied to XML documents, this approach can be effectively used for other semi-structured or hierarchically structured documents, which are ordered or for which position dependent changes need to be detected.

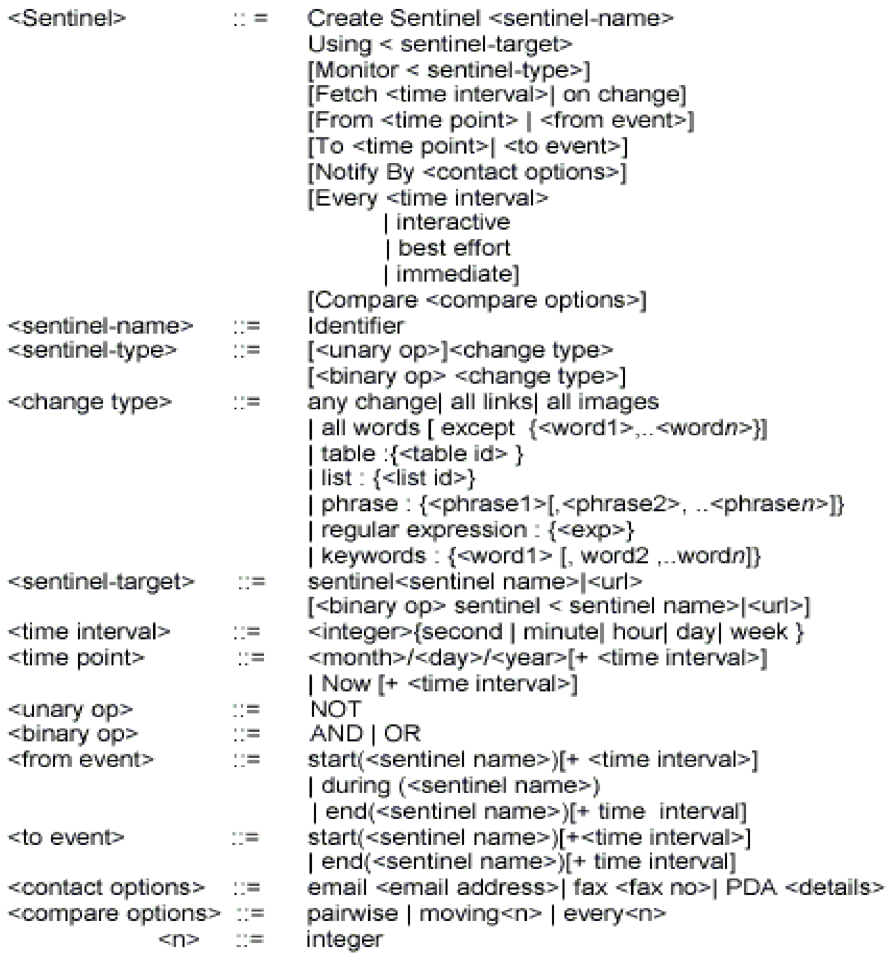
CHAPTER 5

CHANGE SPECIFICATION LANGUAGE

The Internet has evolved as an indispensable repository of information. The present day web user's interest has extended from mere retrieval of information to monitoring the sequence of changes (to web pages) that are of interest. As the web pages are distributed over multiple large repositories, the emphasis is on selective and timely propagation of information/changes. Changes need to be notified to the user in different ways based on user preferences as specified in their profiles/policies. In addition, the notification of these changes may have to be sent to different devices that have different storage and communication bandwidths. The language for establishing the user policies should be able to accommodate the requirements of a heterogeneous distributed large network centric environment. Hence, there is a need to define an expressive and extensible specification language wherein the user can specify his policies to specify details, such as the web page(s) to be monitored, the type of change (keywords, phrases etc.), and the interval for comparing occurrence of changes. The user should also be able to specify how, when, and where to be notified, taking into consideration the quality of service factors (timeliness, size vs. quality of notification) for example). The user should be allowed to specify the type of comparison between two pages from a set of compare alternatives. In addition, the user should be allowed to specify a reference page with which a new page is compared for changes.

WebVigiL provides an expressive language with well-defined semantics for specifying the monitoring requirements of a user, pertaining to the Web. The Sentinel

Specification language developed for this purpose allows the user to create a monitoring request based on his requirements. Each monitoring request is termed a Sentinel.
The semantics of the sentinel specification language for WebVigiL has been formalized. The syntax of the language is shown in Figure 5.1.



```
<Sentinel>            :: =   Create Sentinel <sentinel-name>
                             Using < sentinel-target>
                             [Monitor < sentinel-type>]
                             [Fetch <time interval>| on change]
                             [From <time point> | <from event>]
                             [To <time point>| <to event>]
                             [Notify By <contact options>]
                             [Every <time interval>
                                   | interactive
                                   | best effort
                                   | immediate]
                             [Compare <compare options>]
<sentinel-name>       ::=    Identifier
<sentinel-type>       ::=    [<unary op>]<change type>
                             [<binary op> <change type>]
<change type>         ::=    any change| all links| all images
                             | all words [ except {<word1>,..<wordn>}]
                             | table :{<table id> }
                             | list : {<list id>}
                             | phrase : {<phrase1>[,<phrase2>, ..<phrasen>]}
                             | regular expression : {<exp>}
                             | keywords : {<word1> [, word2 ,..wordn]}
<sentinel-target>     ::=    sentinel<sentinel name>|<url>
                             [<binary op> sentinel < sentinel name>|<url>]
<time interval>       ::=    <integer>{second | minute| hour| day| week }
<time point>          ::=    <month>/<day>/<year>[+ <time interval>]
                             | Now [+ <time interval>]
<unary op>            ::=    NOT
<binary op>           ::=    AND | OR
<from event>          ::=    start(<sentinel name>)[+ <time interval>]
                             | during (<sentinel name>)
                              | end(<sentinel name>)[+ time  interval]
<to event>            ::=    start(<sentinel name>)[+<time interval>]
                             | end(<sentinel name>)[+ time interval]
<contact options>     ::=    email <email address>| fax <fax no>| PDA <details>
<compare options>     ::=    pairwise | moving<n> | every<n>
                  <n> ::=    integer
```

Figure 5. 1 Sentinel Syntax

Following are some scenarios and their representation using the sentinel specification language.

Scenario1: Jill wants to monitor the UTA class schedule website "http:// w w w. u t a.edu/class schedule" for the keywords "5331,5324 " to take a decision for registering for these courses for a particular semester. The sentinel starts from December 2, 2002 to January 2, 2003 and she wants to be notified as soon as possible when the change is detected. The sentinel (s1) for the above scenario is as follows:

Create Sentinel s1 Using http:// w w w. u t a.edu/class schedule

Monitor keyword (5331, 5324)

Fetch 1 day

From 12/02/02 To 01/02/03

Notify By email jill@aol.com Every best effort

Compare pairwise

Scenario2: Jill wants to be notified for any change to the page "http://www2.uta.edu/sharma/courses/cse5331And4331/
Spring2003/General/course_sc hedule.htm" as this may affect her decision for registering for the course 5331. As it is correlated with sentinel s1, the duration is specified between the start of s1 and the end of s1. The sentinel (s2) for the above scenario is:

Create Sentinel s2

Using www2.uta.edu/sharma/courses/cse5331And4331/
Spring2003/General/course_schedule.htm

Monitor all words OR anychange

Fetch on change

From start (s1) To end (s1)

Notify By email jill@aol.com Every 1 day

Compare pairwise

Scenario3: Jill decides to monitor another course "Optical Network" on the class schedule web page "http:// w w w. u t a.edu/class schedule". As she has already specified a sentinel s1 on the same link and requires similar properties for this sentinel too, she can specify a new sentinel s3 using her previously defined sentinel s1. The sentinel (s3) for the above scenario is:

Create Sentinel s3

Using s1

Monitor phrase ("Optical Network")

The rest of this section elaborates on the semantics of a sentinel, using the above scenarios.

5.1 Sentinel Name

Create Sentinel <sentinel name>

For every sentinel, the system generates a unique identifier. In addition, the system also allows the user to specify a sentinel name of his choice. This is to facilitate the user, when he wants to specify another sentinel in terms of his previously defined sentinels, as it would be difficult for the user to track the system defined sentinel name. The user is required to specify a distinct name for all his sentinels.

5.2 Sentinel Target

The user should be allowed to specify the reference web page, which is of interest and needs to be monitored for changes. In addition, the user should be allowed to specify the monitoring target based on previously defined sentinels, which are correlated. Sentinels are correlated if they inherit properties during runtime like start and end of a sentinel. Otherwise, they merely inherit static properties (e.g. URL, name etc. of the sentinel)

Using <sentinel- target>

The *sentinel-target* could be either a URL or a previously defined sentinel s_i. If the new sentinel s_n, specifies the sentinel target as s_i, then s_n inherits its properties from s_i , unless the user overrides those properties in the current specification. Consider the scenario 4, where the user requires monitoring changes to the same web page based on the same policies defined in a previously defined sentinel s3. But he wants to monitor the links on the page instead of the images. The language provides the user the facility to inherit his/her properties from the user's previous sentinel unless overwritten by the user. Hence, in the given scenario 3, all the properties of s1 will be inherited except the sentinel type, which will be replaced from "keywords" to "phrase".

When a sentinel is defined in terms of a previously defined sentinel, the validation module first validates the inherited properties before the created monitoring request is started.

5.3 Sentinel Type

Meaningful change monitoring can be provided only if changes are detected at a finer level of granularity. The user may be interested in selective changes (e.g., phrases, keywords etc) on the content of the page rather than changes to the entire page. Consider scenario 1 where the student Jill wants to monitor the college schedule of class to find whether a particular course (cse5331) is offered or not for that semester. Hence, she has to monitor the web page for that particular keyword i.e., cse5331. In such cases, detecting changes to the complete page is: i) not meaningful to Jill ii) disseminates irrelevant information, and iii) leads to wasteful computation. WebVigiL allows the detection of customized changes in the form of sentinel type and provides explicit semantics for the user to specify his desired type of change. The semantics of sentinel type is given as:

Monitor <sentinel-type>

sentinel-type=[<unary op>]<change type> [<binary op> <change type>]

Change is computed between two versions of the same page with respect to the type of change specified. The *sentinel-type* is the change type *t* selected from the set *T* = {any change, all links, all images, all words except <set of words>, phrase:<set of phrases>, keywords:<set of words>, table: <table id>, list :<list id>, regular expression: <exp> }. The user is interested in the contents of a web page. The contents can be defined as a set of words, links and images. From the set of words, the user may be interested in monitoring changes to some particular keywords or a sequence of words such as phrase. Similarly, the user's object of interest may be links and images also. Thus a page can be defined as a set of objects of user's interest, which can be classified as, Key Words, Phrases, All Words, Links and Images. Based on this, the content of a page can be classified into the following:

Keywords: Corresponds to a set of unique words from the page. A change is flagged when any of the keyword appears or disappears in a page with respect to the previous version of the same page.

Phrase: Corresponds to a set of contiguous words from the page. A change is flagged on the appearance or disappearance of a given phrase in a page with respect to the previous version of the same page. Update to a phrase is also flagged depending on the percentage of words that has been modified in a phrase. If the number of words changed exceeds above a threshold, it is deemed as a delete (or disappearance).

Links: Corresponds to a set of hypertext references. Links are presentation-based object. In HTML, links are represented between the hypertext tag (<A href=".">). Given two versions of a page, if any of the old links are deleted in the new version or new links are inserted, then a change is flagged.

Images: Corresponds to a set of image references extracted from the image source. Images are also presentation based objects and in HTML are represented by the image source tag (IMG src=".">). The changes detected are similar to the links except that the images are monitored instead of the links.

Table: Corresponds to the content of the page represented in a tabular format. Though the table is a presentation object, the changes are tracked on the contents of the table. Hence, whenever the table contents are changed, it is flagged as a table change. Currently changes to the content of a table are not supported in WebVigiL but we intend to support this type of change in the future.

List: corresponds to the contents of a page represented in a list format. The list format can be bullets or numbered. Any change detected on the set of words represented in a list format is flagged as a change. Currently changes to the content of a table are not supported in WebVigiL.

*Regular expressio*n <exp>: Expressed as valid regular expression syntax for querying and extracting specific information from the document data. Currently, we do not support regular expression but intend to do so in the future.

All words: A page can be divided into a set of words, links and images. Any change to the set of words between two versions of the same page is detected as a all words change.

Anychange: Anychange encompasses all the above given types of changes. Changes to any of the defined set (i.e. all words, all links and all images) are flagged as anychange. Hence, the granularity is limited to a page for anychange.

Any change is the superset of all changes. All words encompass phrases, keywords and words in the table and list. While considering changes to all words, the presentation objects such as table and list are not considered and only the content in these presentation objects are taken into consideration.

If V_1 and V_2 are two different versions of the same page, then Change *C* on V_2 with reference to V_1, is defined as:

$C_{\mathbf{t}}$ (V_1, V_2) = *True* if the change type *t* is detected as insert in V_2 or delete in V_1 or update in V_2.

False otherwise

Primitive change is the detection of a single type of change C between two versions of the same page. For keyword change, the user must specify a set of words. For phrase change, a set of phrases is specified. For regular expression, a valid regular expression is given. In some cases, the user may want to detect the non-occurrence of a type of change on a page. For example, the user may be interested in monitoring a page only as long as it contains the required links or a particular keyword. To facilitate detection of such changes, the unary operator NOT is specified in the SSL.

NOT: NOT is a unary operator, which detects the non-occurrence of the given change type *t* on version V_2 with reference to version V_1 of the same page.

$(\text{NOT } C_{\mathbf{t}})(V_1,V_2) = {\sim}C_{\mathbf{t}}\ (V_1,V_2)$

Many times, the user may be interested in detecting more than one type of change on a given page. For example, in the given scenario 2, Jill wants to be notified of a change if any words from the set of words in a page change or anychange type are flagged. Hence, the user should be allowed to specify more than one type of change on a page at a finer granularity. WebVigiL detects such types of combination of change and the change specification language (CSL) defines it as a composite change. Composite change comprises of a combination of *distinct* primitive change(s) specified on the same page, using one of the binary operators AND and OR. The semantics of composite change formed by the use of an operator can be defined as follows (Note that Λ , V, and ~ are Boolean AND, OR, and NOT operators, respectively):

OR: Disjunction, denoted by C_t^1 OR C_t^2, of two primitive changes C_t^1 and C_t^2 specified on version V_2 with reference to version V_1 of the same page, is detected if either C_t^1 is detected or C_t^2 is detected. Formally,

$(C_t^1 \text{ OR } C_t^2)(V_1,V_2) = C_t^1(V_1,V_2) \vee C_t^2(V_1,V_2)$

where t1, t2 are the types of changes and t1<>t2

AND: Conjunction, denoted by C_t^1 AND C_t^2 of two primitive changes C_t^1 and C_t^2 specified on version V_2 with reference to version V_1 of the same page, is detected when both C_t^1 and C_t^2 are detected. Formally,

$(C_t^1 \text{ AND } C_t^2)(V_1,V_2) = C_t^1(V_1,V_2) \wedge C_t^1(V_1,V_2)$

where t1, t2 are types of changes and t1 <>t2

The unary operator NOT can be used to specify a constituent primitive change in a composite change. For example, for a page containing the list of fiction books, a user can specify a change type as: All words AND NOT phrase {“ Lord of the Rings”}. A change will be flagged only if given two versions of a page, at least some words may change such as insertion of a new book and author etc. but the phrase “Lord of the Rings” must not have changed. Hence, the user is interested in monitoring the arrival of new books or removal of old books, only as long as the book “Lord of the Rings” is available.

For detection of these changes as defined, the change detection module detects the type of changes on a page as insertion of the changes in the new page or deletion of object of interest from the old page. For XML type of documents, the changes are detected based on the order of the user's object of interest in the page. Hence, the change detection definition is extended to include 'move' of a user's object of interest also. By 'move', we detect the movement of change from position n in old page to position m in new page where $n \neq m$.

The change monitoring of all the created sentinels is done by the change detection graph (CDG) as discussed in section 2.5, which uses the active capability. The CDG fetches the pages of interest when a change needs to be detected and based upon whether it is a primitive change or a composite change and the type of change, takes the appropriate actions to detect the changes. As HTML is unstructured and XML is structured, different methods of change computation are performed for both the kinds of documents. The CDG, depending upon the type of document and change, calls the appropriate mechanism for change detection.

5.4 Fetch

To detect changes on a given web page, a new version of the page has to be fetched. Ideally, the new version should be fetched only when certain page properties change. Page properties such as last modified date for static pages or checksum for dynamic pages denoted as meta-data, define whether a page has been modified. WebVigiL's fetch module relies on a learning algorithm based on the meta-data properties and history of changes to the page to determine an effective interval with which a page should be fetched. In some cases, the user may be aware of the change pattern of the page or may be interested in detecting changes only at a particular interval. For example, user Don knows that the project status page gets updated every morning at 9 am and wants to monitor changes at that time. Hence, the user may be interested in monitoring changes only with a fixed frequency.

The fetch attribute allows a user to specify how often a new page should be fetched:

Fetch <time interval>| on change

On change: Indicates that the system is responsible for fetching the page as soon as it is modified. The specification of the actual change frequency relieves the user of knowing when the page changes and requests the system to do its best effort to fetch the page on modification. The interval with which a page should be fetched, is determined by a heuristics-based fetch algorithm called Best Effort Algorithm [9], a learning algorithm dependent on change history and meta-data of the page.

<time interval> t_d : is a fixed user-defined fetch interval when a page is fetched by the system . t_d can be in terms of minutes, hours, days or weeks and is a non-negative integer.

In WebVigiL, we are able to support both "on change" and "interval" based changes because we use periodic events [8]. To actually fetch the page, we associate a fetch rule with this periodic event. Hence whenever a periodic event occurs, the rule associated with it is triggered. The rule could either be an interval based rule (section 2.6) or a best effort rule (section 2.6). In interval based rule, the page is fetched at the given user defined fetch frequency (i.e. time interval) which is associated with all pages which has to be fetched at fixed frequency. For changes with a fetch frequency of "On change", the best effort rule is associated, which uses the Best Effort algorithm [9].

5.5 Sentinel Duration

WebVigiL is not a continuous query processing system. It uses the duration of the sentinel as the interval for monitoring. The duration defines the lifespan of a sentinel and is the closed interval formed by the start time and end time of the sentinel, during which the change is monitored. This is defined as:

From <timepoint>| <from event>

To <timepoint>|<to event>

Let *time line* be a equidistant discreet time domain having "0" as the origin and each time point as a positive integer as defined in [5]. Defining it in terms of the timeline, occurrences of the created Sentinel *S* are specific points on the time line and the duration (lifespan) defines the closed interval within which *S* occurs. The *From* modifier denotes the start of a sentinel S and the *To* modifier denotes the end of S. The start and end times of a sentinel can be a specific time or can depend upon the attributes of other correlated sentinels. Users can specify the duration as one of the following: (a) Now (b) Absolute time (c) Relative time (d) Event-based time

Now: A system-defined variable that keeps track of the current time.

Absolute time: Denoted as time point T, it can be specified as a definite point on the time line. The format for specifying the time point is MM/DD/YYYY.

Relative time: It is defined as an offset from a time point (either absolute or event-based). The offset can be specified by the time interval t_d defined in section 5.4.

Event-based time: Events, such as the start and end of a sentinel can be mapped to specific time points and can be used to trigger the start or end of a new sentinel. Start of a sentinel can also be dependent on the active state of another sentinel and is specified by the event 'during'. During s_i defines that a sentinel should be started in the closed interval of s_i and the start should be mapped to *Now*. Start s_i defines that the sentinel should be started/ended as soon as s_i starts. End s_i defines that the sentinel should be started/ended as soon as s_i ends.

The sentinel is activated at the start time and is deactivated at the end time by the events and rules defined by the ECA Rule Generator module, using the rules defined in [5, 6]. This is explained in detail in section 2.3. If the duration is not specified correctly, the sentinel will never start or continue forever without ending which WebVigiL currently does not support. For example, in scenario 2, if sentinel s2 is dependent on sentinel s1 to start it's monitoring. But if s1 has already started before the creation of s2, then s2 will never start. Hence, the semantic check should capture such errors and notify the user. When a sentinel is inherited (as in scenario 3), the properties of that sentinel are inherited. If a sentinel inherits from another sentinel having a start time of *Now*, as the properties are inherited, the time of the current sentinel will be mapped to the current time. Similarly, if the sentinel duration is defined as a relative time on an event and if the event has already occurred, the validation module is required to map it into the appropriate absolute valid time.

5.6 Notification

Users need to be notified of detected changes. How, When and Where to notify is an important criterion for notification and should be resolved by the change specification semantics. The mechanism selected for notification is important especially when multiple types of devices with varying capabilities are involved. The semantics for specifying the notification mechanism is given by:

Notify By <contact options>

The <contact *options*> allows the users to select the appropriate mechanism for notification from a set of options *O* = {email, fax, PDA}. The default mechanism is email. The user will also provide the appropriate contact address e.g. email id etc in the contact options. This contact id will also be the userId of the user and all the sentinels of the particular user will be tracked based on the contact address. The notification module has to ensure that the detected changes are presented to the user at the user specified frequency. The system should incorporate the flexibility to allow users to specify the desired frequency of notification. For example, in sentinel s2, Jill wants to be notified once per days, irrespective of when the changes are detected. The semantics of notification frequency has been defined as:

Every best effort | immediate | interactive| <*time interval*> where

<*time interval*> is as defined in section 5.4.

immediate means to be notified immediately on change detection.

best effort is defined as notify ASAP (as soon as possible) after change detection. Hence, best effort is equivalent to immediate but will have lesser priority than immediate for notification.

Interactive is a navigational style notification approach where the user visits the WebVigiL dashboard to retrieve the detected changes at his/her convenience. A WebVigiL dashboard will be provided to the user to view and query the changes generated by his sentinels.

In the given scenario, for sentinel s2, the user wants to be notified when the change is detected but can tolerate some delay and hence has specified a notification frequency of best effort. Depending upon the changes detected and the mechanism to transmit the changes, the presentation module (section 2.8) will select the appropriate presentation method and notify the user.

5.7 Compare Options

Changes are detected between two versions of the same page. Each fetch of the same page is given a version number. The first version of the page will be the first page fetched after a sentinel starts. Given a sequence of versions V_1, V_2 ......V_n, of the same page, the user may be interested in knowing changes with respect to different references. By default, the page previous (based on user-defined fetch interval where appropriate) to the current page is used for change detection. For example, a user may want to monitor changes between every n versions of the page. In order to facilitate this, the compare option can be selected from a set *P* = {pairwise, moving n, every n} and is specified by: *Compare <compare options>*

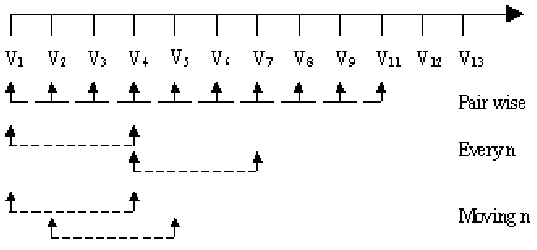


Figure 5. 2 Compare Methods

Pairwise: The default is pairwise, which will allow change comparison between two chronologically adjacent versions (Figure 5. 2).

Every n: This compare option allows a user to detect changes between versions V_i and V_{i+n}. For the next comparison, the nth page becomes the reference page. For example if a user wants to detect changes between every 4 versions of the page, the versions for comparing will be selected as shown in Figure 5. 2. This option is useful if the user is aware of the changes occurring on a page such as a web developer or administrator and can assume that the cumulative changes between only *n* versions are important.

Moving n: This is a moving window concept for tracking changes. For moving n, the first reference page is compared with the nth page. For the next comparison, the subsequent version is taken as the reference page and is compared with its nth page, starting from the reference page. In the given scenario, for sentinel s3, Jill specifies the compare option of moving n where n=4. As shown in Figure 5. 2, V_1 will be the reference page for V_4. The next comparison will be between V_2 and V_5. The moving window concept is useful for tracking changes to stock pages or other pages where meaningful change detection is only possible between particular set of pages occurring in a moving window.

The user-given compare type is used by the change detection graph (section 2.5) to select the appropriate versions of the page for change detection. Most of the tools developed for change detection (chapter 3) only give the user the option for pairwise change detection. WebVigiL emphasizes on giving the users more flexibility and options for change detection and hence have incorporated several compare options for efficient change detection.

5.8 Summary

The change specification language gives the flexibility to the user to specify his/her intent WebVigiL is one of the few change monitoring system, which allows specification and detection of more than one type of change (composite) on the web page. In addition, choice of selection of reference page, inheritance and event-based duration is novel to WebVigiL system.

CHAPTER 6

DESIGN ISSUES FOR KNOWLEDGEBASE

The sentinel defined by a user is a monitoring request, which contains the policies provided by the user for monitoring and notification of selective changes on a page. Many modules use the sentinel details at runtime. For example, the change detection module detects changes based on sentinel information such as the URL to be monitored, the change and compare specifications, and the start and end of a sentinel. The fetch module fetches the pages based on the user specified fetch policy. The notification module requires appropriate contact information and notification mechanism to notify the changes. User information, such as the sentinel creation date, and the page versions for change detection and storage path of detected changes also need to be stored to allow a user to keep track of his/her sentinels. Hence, there is a need to extract the sentinel information known as meta-data and store it in a persistent and recoverable manner. WebVigiL stores the sentinel information in a persistent repository called the Knowledgebase (KB).

A central repository containing meta-data that is accessible to the different modules of the system should contain the following properties:

- The relevant information should be collected and stored in a manner, which can be easily accessed by all the modules of WebVigiL.
- The information should be persistent and recoverable.
- The repository should be robust and scalable to store large amount of data.

- Efficient data retrieval mechanism is very important when large amount of data is stored and many modules need to retrieve this information.
- There should be a mechanism to store historical information. This separates the historical information from the main data and yet provides a way to retrieve it when needed.

Relational database have in-built robust features for persistence and recovery. They can store large amounts of data as well. In addition, relational databases provide mechanisms to extract the required information in a convenient manner in form of queries or through the JDBC Bridge. Hence, WebVigiL system uses a relational database (Oracle 9i on a Linux platform) for KB.

6.1 Meta-data

The monitoring request is parsed and sentinel properties are extracted, validated and stored in the KB. These sentinel properties; useful to different modules at run-time is referred to as meta-data. In addition, certain run-time meta-data of different modules is also stored in KB. As shown in Figure 6. 1, the various modules access the KB to extract the required information and also to store data, which will be required by different modules or by itself for later use.

User Interface: A sentinel submitted by a user contains user policies. These policies such as the web page to monitor, the change detection interval and version of pages, notification options etc are disseminated from the sentinel information, validated and stored in the KB. User properties such as the userId of the user, user's contact address and login access information are stored in the KB. In addition, the sentinel creation time is also persisted.

The user can specify a sentinel based on his/her previously created sentinels. In that case, all properties of the previous sentinel are inherited unless overwritten by the user in the current sentinel. On request from the user interface, these properties are retrieved from the KB and sent to the user interface.

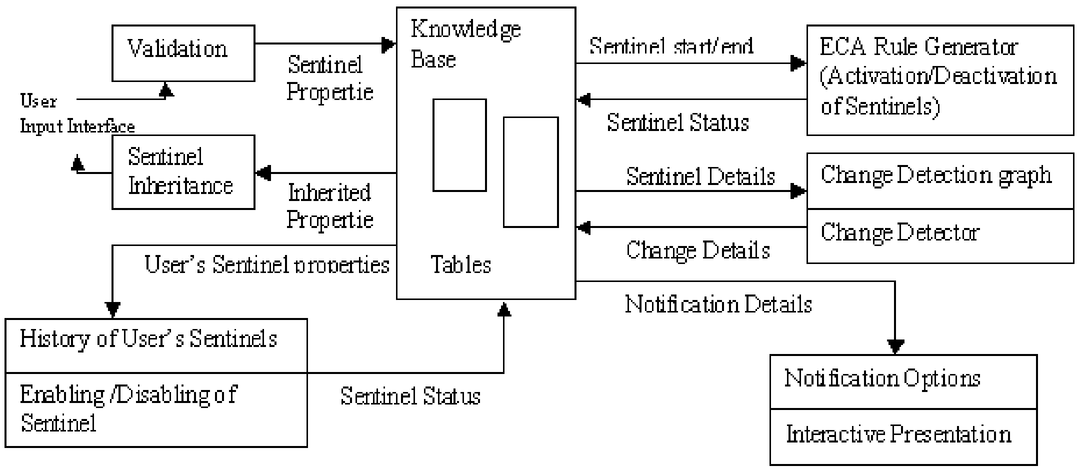


Figure 6. 1 Modules accessing Knowledgebase

ECA Rule Generator: When a sentinel is created, the ECA Rule Generator (ECARG) is notified. The ECARG retrieves the start and end time of a sentinel from the KB. Based on this information, a periodic event [8] for activation and deactivation of the sentinel is created. Additional information such as the target webpage, the fetch details etc. are also extracted by the ECARG. The fetch details indicate whether the user wants to fetch the page at a particular frequency irrespective of the time of change or wants to fetch the pages, upon a change. Based on these details, the interval based rule or the best effort rule (section 2.6) is associated with the periodic event to fetch the pages and detect changes.

Once the sentinel is activated/deactivated, the status of the sentinel is updated in the KB by the ECARG. The validation module retrieves this information for validation for other dependent sentinels.

Change Detection Module: The Change Detection modules extract information such as the web page to monitor, the type of change (i.e. primitive or composite) (section 5.3) and the compare options for change comparison.

Once the change is detected, change details such as the time of change detection and the location, where the changes are stored are updated in the KB. This information will be utilized by the user dashboard to display the sentinel history and also by the presentation module to notify the user.

Presentation Module: The presentation module extracts the change details such as type of changes and the location where the changes are stored, from the KB. Based on this information, the presentation module selects the type of presentation from alternative presentation choices.

The notification details such as user's notification frequency selected from the set {time interval, interactive, best effort, immediate} and the contact information of the user to propagate the changes are extracted from the KB by the notification module (part of the presentation module) to notify the user of the detected changes.

User Dashboard: Information such as the status (i.e., enabled, disabled etc) of the user's sentinels, the sentinel creation details and the change details are retrieved from the KB and displayed to the user.

During the lifespan of the sentinel, the user is allowed to enable or disable, the user's sentinel. This status change of the sentinel is updated in the KB. Based on this information, the ECA Rule Generator enable/disables the rules (section 2.3) associated with the sentinels. The user is also allowed to delete the sentinel during its lifespan. In such cases, all information pertaining to that sentinel is removed from the KB.

6.2 Knowledgebase Schema

The Knowledgebase (KB) is implemented in a relational database and the meta-data is stored in different relational tables. As shown in Figure 6. 1, different modules access the KB to extract and store the required information, which are useful to other modules. The schema is designed in such a way that all the modules can access and retrieve their required information with ease. Similarly, the information is stored also in an efficient way. Updates to certain meta-data information that affect other meta-data information are captured using referential integrity. For example, the lifespan of a sentinel can be defined on another previously defined sentinel. The referential integrity ensures the validity of the previously defined sentinel. Similarly as WebVigiL supports only interval-based sentinels and not continuous queries, each sentinel should have a valid start and end time. Such referential details are incorporated in the KB schema. If these validations are not satisfied, a database error is declared.

Figure 6.2 shows the UML conceptual schema for the KB. As shown in the figure, the main sentinel details and user related details are obtained from the wv_sentinel table. The user interface allows only authenticated users to create sentinels. For this the user has to provide a correct username (email id) and password. The user authentication information is stored in the wv_userauthentication table. The lifespan details of a sentinel are stored in the wv_sentinelstart and wv_sentinelend tables. The notification details are stored in the wv_notify table and the fetch details in the wv_fetch table. The composite and primitive types of change details are specified in the wv_composite and wv_primtive tables respectively. The status of the sentinel (start, enable, disable, end) is stored in the wv_sentinelstatus table.

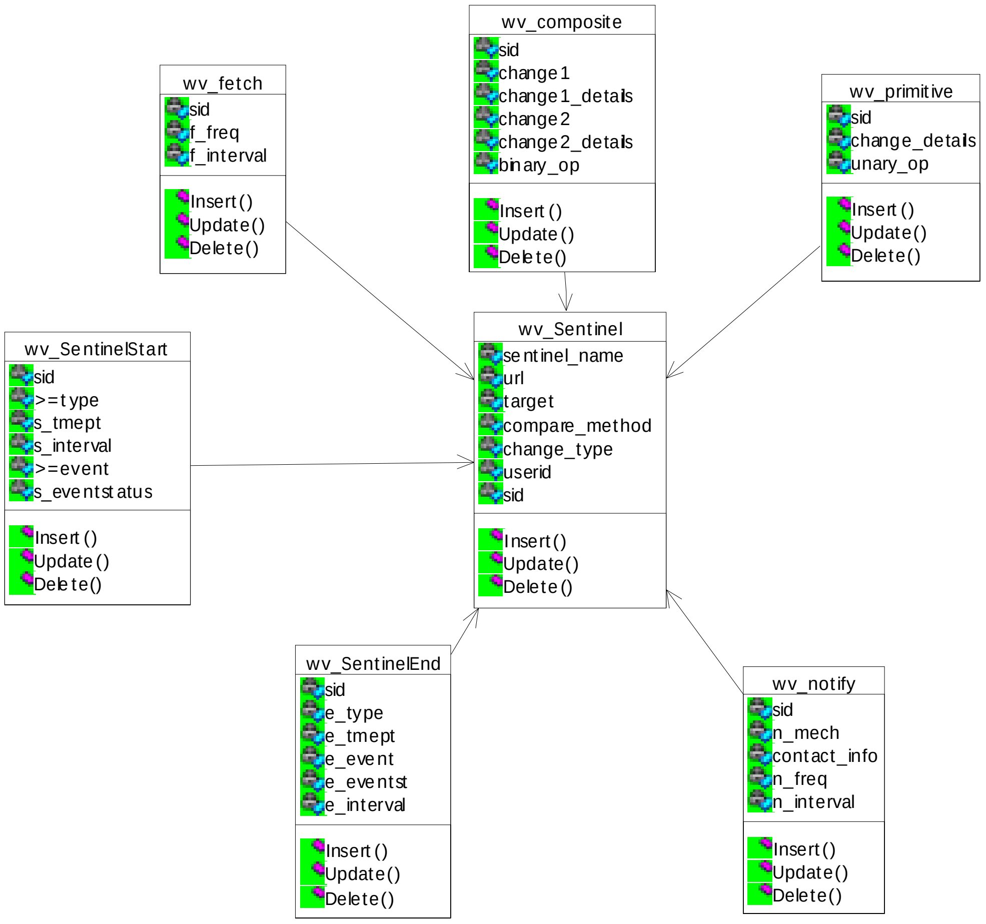


Figure 6. 2 Schema diagram for Knowledgebase

6.3 Validation Module

WebVigiL allows selective monitoring of web pages and notification based on user-defined policies. The system manages and propagates the changes using the active paradigm depending upon the user policies extracted from the sentinel. Hence, it is very important that the information extracted from the sentinel is correct syntactically and semantically. Each monitoring request is checked for correctness before being persisted in the KB.

The validation module does three types of validation: syntax, semantics and database. Syntax validation checks the correctness of sentinel in terms of the defined grammar of the change specification language (CSL). The syntax and semantics of CSL is described in chapter 5. The user specification is parsed and checked for correctness in terms of the defined grammar and the data types of the value. For example, the value of time interval has to be specified in either minutes, hours, days or weeks. Time interval specified in any other type will not be accepted. As described in (section 5.5), the start and end time can be specified as an event in terms of previous defined sentinel status. The event status "during" cannot be specified for the end time of a sentinel in the CSL and a syntax error will be thrown if specified. Similarly, the format for specifying the time point is mm/dd/yyyy i.e. month/ day/ year in four digits. Time point specified in any other format will not be accepted.

6.4 Semantic Validation

The accepted sentinel should be semantically correct for the policies to be meaningful and used by different modules of WebVigiL. Semantic validation is done on the content of the monitoring request.

Start/ End time: WebVigiL supports interval-based monitoring request. The activation and deactivation of a sentinel depends upon the start and end time defined for the sentinel. The ECA rules for fetching the pages and detecting changes will also be enabled depending on the start and end time. Hence, the semantic correctness of the specified start and end time of a sentinel has a bearing on the correct output of the different modules. If the duration is not specified correctly, the sentinel will never start or will continue forever without ending. This is unacceptable, as WebVigiL does not support continuous queries.

As explained in section 5.5, the start and end time can be defined as either *Now* (i.e. current time), absolute time, relative time point or as an event. If the start is defined as time point or event-based and if the specified time has already lapsed before sentinel creation, the semantic check will throw an error. In case of relative time, during sentinel creation, the absolute time/event may have lapsed but not the interval/offset. In such cases, the appropriate total absolute time (absolute time + interval) needs to be computed by the validation module. Conversion to absolute time is a requirement for the ECA generator also. An event, which is in future, can be scheduled to be fired but not an event, which has already, lapsed. This problem, defined as the *time lapse* problem, would result in malfunctioning of the system as certain events will never be fired and hence the associated sentinels will never be active. Hence, these values should be converted in to the appropriate absolute time before the ECA agent retrieves the information.

If end time is specified as *Now*, it is syntactically valid but semantically invalid, as the sentinel will end as soon as it starts. However *Now* + time interval can be allowed.

For event-based time specification, semantic validation is very critical. The event status specified should be satisfied for acceptance. As defined in section 5.5, the following cases are allowed for a sentinel s1 whose start or end are event-based and specified on event s:

- If the given status is "start" of another sentinel s, then s should not have started.

- If the status is "end" of s, then s can be active but should not have ended.
- If the status is "during" of s1, then the lifespan of s starts during the lifespan of s1. s1 should have already started but should not have ended. The end time of a sentinel cannot be specified as "during" of a previously defined sentinel.

For event-based events, the defined events' start/end time can also be an event, creating many dependencies and making validation more complex. For example start of sentinel s2 may be defined on end of sentinel s1 and the end of s1 can be start of s plus some time interval. The *time lapse* problem explained above can also affect these cases. For example, when s1 was defined, s may not have started but when s2 was created, s had already started. But s1 will not end, as the time interval has not lapsed. To validate such cases, the validation modules have to convert the events into absolute time based on either their activation time or the specified time for activation and validate according.

Sentinel name: Each Sentinel has to be identified by a unique name. The user can denote a name for his created sentinels but the validation module should ensure that the sentinel name is unique.

Change Type: The specified change type should be semantically meaningful. For example change type: "all links and not any change" is not valid as link change would trigger the change type "any change".

Inheritance: WebVigiL allows the user to inherit properties from previously defined sentinels. The properties are the specifications provided by the user in the sentinel and not the current status of the sentinel. In case of inheritance, all the properties are inherited unless overwritten by the user. If the properties are overwritten, then the validation module has to ensure that correctness of the inherited properties as well as the overwritten properties are maintained. If the start and end time of the previous sentinel is inherited, the semantic validation module has to ensure that the time has not lapsed. If the start time is *Now*, the current time is taken instead of the start time of the previous sentinel as the specification are inherited and not the status. Because of inheritance, there is number of dependency issues to be considered for the validity of the sentinel duration as explained above.

Table 6. 1 Combination of Compare, Notification and Fetch options

Compare Options / **Notification options**	**Pairwise**		**Every*n***	**Moving*n***	
Best Effort	OC	T	OC	OC	T
Immediate	OC	T	OC	OC	T
Time interval	OC	T	OC	OC	T
Interactive	OC	T	OC	OC	T

Fetch frequency: On change - OC, Time interval: - T

Other Semantic Validations: Taking the notification, fetch and compare options definition into consideration, it was verified that all combination of these three options are valid as shown in Table 6.1. But for a specification of every*n* compare method with <time interval> fetch frequency is equivalent to a specification containing pairwise compare method with <time interval> fetch frequency. For example, every3 with 10 minutes fetch frequency is equivalent to pairwise comparison with 30 minutes fetch frequency. The fetch module should take into consideration such equivalent conditions to avoid redundant fetching. Specifying a time interval for every*n* is redundant. Hence, only “on change” option is considered in the given table.

6.5 Design and Flow Diagram of the Validation Module

The flowchart diagram for validation and insertion into Knowledgebase is given in Figure 6. 3. As shown in the figure, once the user is authenticated, if the user specifies the sentinel target to be a previous sentinel, all properties of that sentinel are inherited. The user can overwrite the properties, as required. Once the user submits the request, the validation process starts. First, the system ensures that the created sentinel name is unique. The main emphasis is on duration especially if the duration is specified as event-based. This is because, if the specified properties were correct, the ECA Rule Generator would not be able to activate and deactivate the sentinel. Once the validation process is complete, the sentinel properties are inserted into the Knowledgebase.

6.6 Summary

The Knowledgebase is the repository from which all the WebVigiL modules retrieve the required information for efficient monitoring and notification. Hence, proper validation, storage and retrieval of the data stored in Knowledgebase are important.

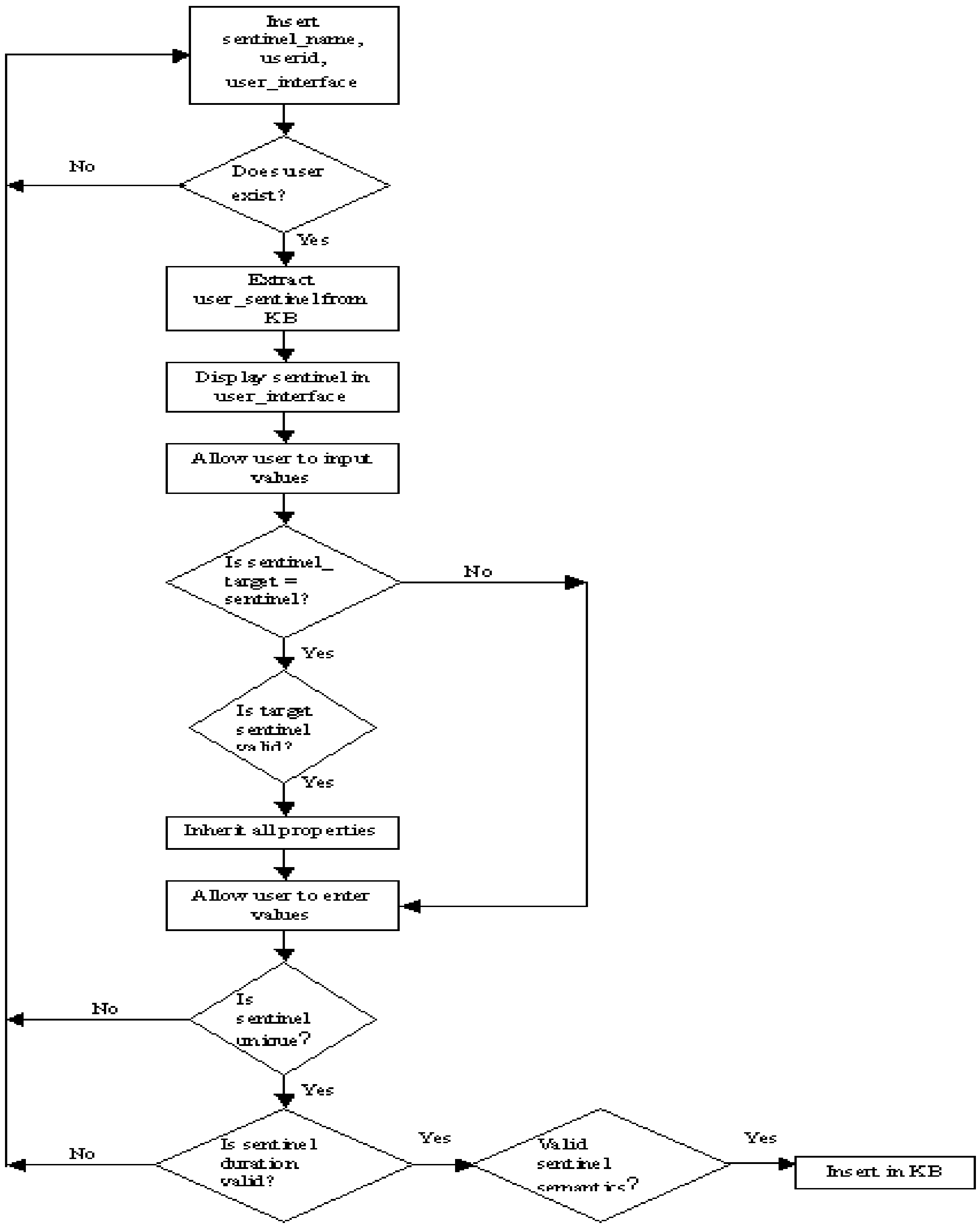

Insert sentinel_name, userid, user_interface
Does user exist?
No
Yes
Extract user_sentinel from KB
Display sentinel in user_interface
Allow user to input values
Is sentinel_target = sentinel?
No
Yes
Is target sentinel valid?
Yes
Inherit all properties
Allow user to enter values
Is sentinel unique?
No
Yes
Is sentinel duration valid?
No
Yes
Valid sentinel semantics?
Yes
Insert in KB

Figure 6. 3 Flow chart for validation and insertion of sentinel in KB

CHAPTER 7

IMPLEMENTATION

This chapter discusses issues that need to be considered for the implementation of the change specification language, the change detection tool for XML and the Knowledgebase. Section 7.1 deals with the implementation of the parser, needed for parsing the change specification language. Section 7.2 explains the semantic validation of the user specification and the flow from the user interface to Knowledgebase to various modules of WebVigiL. Section 7.3 discusses the implementation details for the various steps of XML change detection tool.

7.1 User Interface and Knowledgebase

Once the user provides his/her sentinel specification using the user interface, the flow of control for accepting the sentinel is shown in Figure 7. 1. On user login, the user is authenticated by the system. Once authenticated, sentinels created by that user are extracted and displayed. The user can define a new sentinel either by providing a target as a URL or use a previously defined sentinel. If the sentinel target is a previously defined sentinel, then all the properties of the target sentinel are extracted and displayed to the user. The user has the option to overwrite the inherited properties. Once the user submits a completed sentinel, its properties are extracted and validated semantically, and if correct, stored in the Knowledgebase. The ECA Rule Generator is notified. The ECA Rule Generator extracts the required properties from the Knowledgebase and appropriate events that would start the sentinel. The ECA Rule Generator updates the status of the sentinel such as start and end in the Knowledgebase.

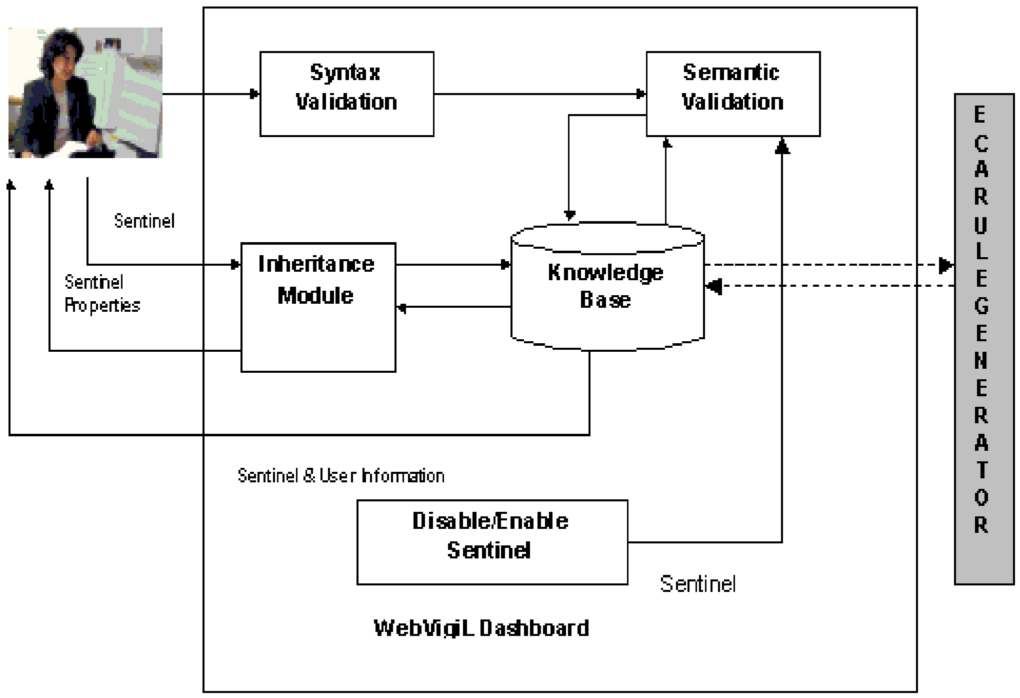


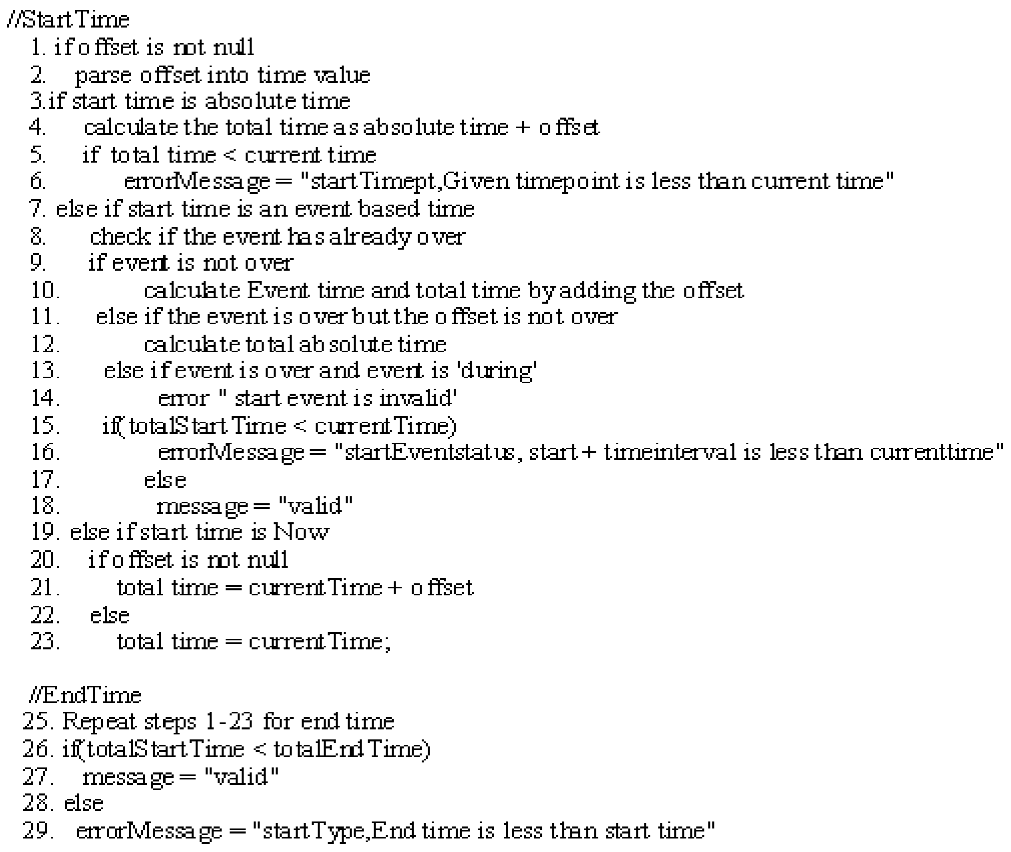
Figure 7. 1 Input Flow

7.1.1 <u>Parser for the Change Specification Language</u>

The change specification language described in chapter 4 needs to be parsed to check for correctness and extract the required information. This parser should be capable of understanding the syntax of the developed language. A parser has been implemented using JavaCC [32]. The language syntax such as ‘Create’, ‘Using’ etc., are considered as tokens and based on the token definition the input is parsed. In the sentinel syntax, only the sentinel name and target are ‘not null’ fields.

7.1.2 Semantic Validation

The required semantic validation has been discussed in section 6.4. Syntactically correct specification details are extracted from the sentinel and validated for semantic correctness as well. As the validity of duration is critical, we will discuss the implementation of the validation module for duration.



```
//StartTime
 1. if offset is not null
 2.   parse offset into time value
 3.if start time is absolute time
 4.    calculate the total time as absolute time + offset
 5.    if total time < current time
 6.        errorMessage = "startTimept,Given timepoint is less than current time"
 7. else if start time is an event based time
 8.     check if the event has already over
 9.     if event is not over
 10.         calculate Event time and total time by adding the offset
 11.     else if the event is over but the offset is not over
 12.         calculate total absolute time
 13.     else if event is over and event is 'during'
 14.          error " start event is invalid'
 15.     if(totalStartTime < currentTime)
 16.          errorMessage = "startEventstatus, start + timeinterval is less than currenttime"
 17.         else
 18.          message = "valid"
 19. else if start time is Now
 20.   if offset is not null
 21.       total time = currentTime + offset
 22.   else
 23.       total time = currentTime;

//EndTime
25. Repeat steps 1-23 for end time
26. if(totalStartTime < totalEndTime)
27.   message = "valid"
28. else
29.   errorMessage = "startType,End time is less than start time"
```

Figure 7. 2 Pseudo-code for validity of duration

For duration, the defined start and end time should be valid and furthermore the end time should be less than start time. In addition, for relative time specifications, if the reference time has lapsed but the offset is remaining, then the specified time need to be transformed into an absolute time for the ECA agent. The pseudo-code for duration validation is shown in Figure 7. 2. For event-based specification, a part of the code for calculating the event time is given in Appendix C.

7.1.3 <u>Knowledgebase</u>

The Knowledgebase is stored in Oracle database (Oracle 9i on Linux). The schema for the database is already given in section 6.2. A JDBC connection object is created for the application to access the relational database. The JDBC connect string is given as:

```
ConnectString=openConnection("oracle.jdbc.driver.OracleDriver",
                "jdbc:oracle:thin:username/password@berlin.uta.edu:1521:MAVHOME")
```

Various APIs have been implemented for efficient retrieval and insertion of values in the relational schemas by different modules. The properties of a specified sentinel, which are extracted are member variables of the *KnowledgeBase* class. Once a sentinel is submitted by the user, the API *public void generateSid()* generates a unique id for each sentinel. The API's associated with insertion are: *insertMain()* : wv_sentinel table (contains the main properties of the sentinel) , *insertSentinelStart():* wv_sentinelstart table(start time of sentinel), *insertSentinelEnd():* wv_sentinelend table(end time of sentinel), *insertNotify()* : wv_notify table (notification details) , *insertFetch()*: wv_fetch table (fetch details) , *insertPrimitiveChange():* wv_primtive table (primitive changes) and *insertCompositeChange():* wv_composite table (composite changes).

For inheritance, the appropriate sentinel specification has to be extracted from the Knowledgebase. The API's associated with inheritance are:

1. public Vector inheritAll(int refSentinel) : Given the sentinel id, it extracts all the properties of the sentinel and returns a Vector containing the properties.
2. public void inheritSentinelDuration(int refSentinel) : Given the sentinel id, extracts the duration properties i.e. start and end time of a sentinel.
3. public void inheritSentinelPropertiesminusDuration(int refSentinel) : Given the sentinel id, extracts all the properties of the sentinel except the duration.

7.2 Implementation of Change Detection algorithm

The change detection of XML documents consists of three steps:

1. Object Extraction and Signature Computation
2. Filtering the unique inserts/deletes
3. Finding the common order subsequence

The main class for XML change detection is *XMLChangeDetector*. The function for anychange is: *detectAnyChange(oldFilename, newFilename)*. It takes two XML files and detects 'anychange'. The function for detecting changes to keywords is: *detectKeywordChange(oldFilename, newFilename,keywords).* It takes two XML files and a vector containing keywords. For detecting changes to phrases, the function is: *detectPhraseChange(oldFilename, newFilename,phrases).* It takes two XML files and a vector containing phrases to be detected. The main method in *XMLChangeDetector* class is *detectChange* which calls the appropriate method for change detection The input parameters for the method *detectChange* is given as shown in Figure 7. 3.

```
private static void usage() {
System.err.println("Usage: XMLChangeDetector (oldFilename, newFilename, Vector
[<keyword>/<phrase>], Vector exceptWords,  TypeOfChange, Boolean Optimize");
System.err.println("  Type of change are: ");
System.err.println("      P = phrase change");
System.err.println("      K = keyword change");
System.err.println("       A = any change ");
System.err.println(" optimize = true will detect changes with optimization. Default is true") ; }
```

Figure 7. 3 Input parameters for the method detectChange

7.2.1 Object Extraction and Signature Computation

To access the content and extract the structure of the XML document, it is first transformed into a Document Object Model (DOM) [4]. The Xerces-J 1.4.4 java parser [33] for XML is used for this purpose. The tree is traversed and the leaf node consisting of text and attribute nodes are extracted. The signature of each node is also computed from the extracted element information. The associated API for extraction is defined in the class: *XMLObjectExtractor*. Two vectors containing old nodes and new nodes are returned containing the leaf nodes and signature in both the trees. For phrase and keyword, additional vectors containing extracted phrases and keywords are returned.

For phrase extraction, value of all the leaf nodes are divided into words and extracted in the order of occurrence. The Knuth-Morris-Pratt (KMP) string-matching algorithm is applied against the sequence of words and the start and end index of all exact matches to the given phrase are extracted. A range is set for the indexes, which defines a phrase, and using the range, the sub-tree containing the phrase along with the parent elements is extracted. A separate tree with “wv_phrase_03” (unique signature) as the root node is created for the extracted sub-tree. The old tree is realigned and the newly created tree is inserted in its correct order of occurrence into the old tree and the tree is realigned. The tree is then traversed to extract the nodes to compute the signature. When the “wv_phrase_03” node is encountered, all its child elements constitute the signature while the leaf nodes contain the value of the phrase. The pseudo-code for phrase extraction is shown in Figure 7. 4.

Phrase Extraction (Tree T1, Tree T2)

```
1. Parse T1
2. Parse T2
3. Extract all the words ∈ leaf nodes in T1
4. Extract all the words ∈ leaf nodes in T2
5. Run the Knuth-Morris-Pratt string-matching algorithm to extract the phrase in T1 and T2
6. Find all the start index and end index of all the phrases in T1 and T2
7. Extract all the leaf nodes in T1 and T2 in the document order.
8. ∀phrase detected in T1
9.     Based on the start and end index of the phrase, set the range for the phrase in T1
11.        Extract the range from T1
12.        Realign the rest of the nodes in T1.
13.        Insert a new node " wv_phrase_03" as root node for the extracted tree
14.        Insert the new sub-tree in T1
15. ∀phrase detected in T2
        repeat steps 9 –14
16. Parse modified tree T1
17. Parse modified tree T2
18.Extract the value of leaf nodes and compute signature for every leaf node in T1 and T2
19. If element node is 'wv_phrase_03', the signature will consist of all its child elements and
path to the root. The value of the leaf node will be the phrase.
```

Figure 7. 4 Pseudo-code for phrase extraction

For keyword extraction, if v(x) is the value(content) of leaf node x, the value is divided into its respective words w_1 to w_n where n is the number of words in *v*. A string compare is carried for each word w and the given keyword k. If a word w_i equal to the keyword *k* is found, the order of occurrence of the keyword in the node, value of the leaf node v(x) and its signature is extracted.

7.2.2 Filtering Unique Inserts/Deletes

totalMatch: The function is defined in the *XMLChangeDetector* class .The value *v* of the node along with the signature is mapped to the java-generated hash code[34]. The value and signature are matched and matching nodes are extracted. The nodes with the associated signature are sorted on the hash code. For phrase change and any change, every element in the new tree set with a hash code greater than the upper bound or lower than the lowerbound of the old tree set is flagged as an 'insert'. Elements that have their hash code within [lowerbound, upperbound] are searched for occurrence of similar elements in the old tree set. If not found, these elements are flagged as 'insert'. Similarly, the nodes with their associated signatures not matched in old tree set are flagged as 'delete'. For phrase change, the associated phrase for each node is also marked as 'insert/delete'. For keyword change, the matched nodes are extracted but keywords are inserted into 'insert'/'delete' vector only after applying the signatureMatch algorithm and if no match is found as discussed in section 4.6.2.

signatureMatch: The function is defined in *XMLChangeDetector* class .The matching is done in the same way as totalMatch algorithm but here only the signature is matched. This is to detect instances of keyword where the leaf node value may have changed but the structure has not. In addition, the common structural information between the two trees can be taken advantage of.

The inserts are stored in the *insert* vector and deletes in the *delete* vector. The matching nodes are stored in *matchedOldNodes* and *matchedNewNodes* vector and given as input to compute the common order subsequence. The matched nodes are sorted according to the position of occurrence in the tree by the function *public String[][] sortByPosition(String array[][]),* which takes an unsorted array and returns a sorted array by position of occurrence. For keywords, the node value, signature and all instances of keyword in the node are sent for detecting the common order subsequence (COS). For matching nodes extracted after applying signatureMatch, the signature and instances of keyword are sent for detecting the COS. In case of keyword and phrases, if all keywords and phrases result in unique insert/delete, then the algorithm stops here.

7.2.3 Finding the Common Order Subsequence

LCS given in [17] is used for detecting the common order subsequence. The function *public void LCS(String[][] oldTree, String[][] newTree)* takes two arrays containing the matched nodes of old tree and new tree and compute the LCS. The fuction:
public void printNonSeq(char[][] table, String[][] oldTree,String[][] newTree, int oldTreePos,int newTreePos) takes the output of the LCS and detects the leaf nodes, keywords and phrases which are not part of the common subsequence between given two trees. The inserts/deletes for keyword change and phrase change are stored in the *insert* and *delete* vectors.

For any change, the non-matching nodes are stored in a vector and given as input to the function: *public void detectMoves(String[][] oldMoveNodes, String[][] newMoveNodes)* for detecting moves. In this fuction, the value of the non-matching nodes is converted into unique values using hashcode, then sorted and compared to detect common values but at different position. These nodes are termed as 'moved'.

7.2.4 Optimization

The optimization methods are implemented in the *Optimization* class. For optimization, once the nodes are extracted, signature is computed and inserted into vectors, the vectors are given as input to the function: *public void matchSubtree(Vector oldTree, Vector newTree).* The hash value of each sub-tree is computed by adding the hash value of each node and its signature in the sub-tree. The Java hashCode[34] function is used for this purpose. The function: *public Vector computeSubtreeHash(String[][] Tree)* computes the hash value of each sub-tree. The generated hash values of the sub-trees are given to the function: *public void match(int[][] oldTree, int[][] newTree, Vector oldT, Vector newT)* which checks for matching sub-trees. If the hash values of two sub-trees are matched, then the function: *public boolean exactCompare(Vector oldT, Vector newT, int subtreeO, int subtreeN)* checks the correctness of the match by comparing each node and its signature of the matching sub-trees in the order of position of occurrence. If two sub-trees are matching, the function: *public Vector extractSubtree(Vector tree, int subtree)* extracts the matching sub-trees from the old and the new tree. After the removal of the matching sub-trees, the nodes of non-matching old and new tree are given as input to the second (filtering unique inserts/deletes) and third (finding longest common subsequence) phase.

7.3 Summary

The above implementation has been tested vigorously with various test cases for performance and scalability. To the best of our knowledge, WebVigiL is the only system, which supports customized change detection to XML documents. Hence, we couldn’t compare the results of our algorithm with any other system. But there is a definite performance improvement as compared to the primary algorithm by using the optimization techniques. For phrase, currently we detect inserts/deletes and implementation of phrase updates is underway.

CHAPTER 8

CONCLUSIONS AND FUTURE WORK

8.1 Conclusions

The semantics of the specification language has been defined and validated. A parser has been developed for the change specification language. For the WebVigiL system, the inheritance module, the ECA Rule Generator to activate and deactivate the sentinels, the change detection tools for XML and HTML and the fetch mechanism has been designed and implemented. The design of the change detection graph is currently underway. We are currently in the process of integration of the entire module for a complete running system.

8.1.1 Knowledgebase and Validation Module

The relational schema for storing the extracted sentinel information in the Knowledgebase has been designed and the various tables have been created. Additional tables for storing user information, status of the sentinels and detected changes have also been created. APIs for the insertion of extracted sentinel properties and for the retrieval of these properties as described in the chapter 7 has been implemented. In additions, APIs for extracting required information for inheritance have also been implemented.

The validation module has also been designed and implemented. The parser does syntax validation. APIs for the semantic validation and the various functions that validate a specification as described in chapter 7 have been implemented.

8.1.2 Change Detection for XML documents

The change detection of keywords, phrases and any change for semi-structured documents such as the XML has been designed and implemented. To the best of our knowledge, this is the first algorithm, which detects customized changes to XML documents. For phrase and keywords, insertion and deletion has been implemented. The algorithm for detecting update to a phrase has been designed and is being implemented. The implemented algorithm has been tested for various XML documents. We have also tested the proposed optimization method for different characteristics of a tree (such as size, height and number of sub-trees in a tree). From the results obtained, it can be concluded that the optimization improves the performance of the primary algorithm while maintaining the correctness of the changes detected. Though this algorithm has been tested for XML documents, the same approach is also applicable to other similar semi-structured documents, as the proposed method is a generalized approach.

8.1.2.1 Performance

A series of performance tests have been carried out to observe the effect of optimization on the performance of the primary change detection algorithm. The change detection algorithm detects changes between two ordered labeled XML trees. Hence, the effect of the optimization and performance of the algorithm had to be tested for different tree characteristics. The performance tests were carried out for the following tree characteristics a) Deep trees to understand the effect on increase of height of a tree.

In this case, the path taken from root to leaf node is increased. Hence, the number of element nodes increase considerably. b) Bushy trees to understand the effect of increase of the leaf nodes and the number of sub-trees.

An XML page generator was implemented for synthetic test data generation. The parameters given as input to the generator are the depth of the tree, the number of sub-trees and leaf nodes, the size of the leaf nodes and the number of common second level sub-trees between the two XML ordered trees. Based on these parameters, the generator randomly creates tags and value for the nodes to generate two XML documents according to the specification. Analysis of the actual XML web pages found from the existing XML repositories on the Internet such as the ACM Sigmod XML repository, indicated that the depth (height) of the tree was usually around 5 to 7 nodes (number of nodes from root to leaf node) deep. The number of second level sub-trees was usually in the range of 5 to 20 and the number of leaf nodes observed was in the range of 300-500 leaf nodes. Taking this observation into consideration, the test cases were designed.

Each test case was run 4 times using both "with optimization' and 'without optimization' options. The average of the 4 runs was taken as the final result. As the cost for parsing the old tree was high in the first run and then consistent later, the cost for second run for parsing the old tree was taken in the total time for change detection. Below, we discuss the performance results for various tree characteristics:

a. Wide trees: Wide trees means tree having more number of leaf nodes and more second level sub-trees. The dataset characteristics of the tree consisted of 1) leaf nodes: 100-400 2) Sub-trees: 10-40 3) Common second level sub-trees: n-3 where n is the number of sub-trees 4) depth: 4. The observed performance is shown in Table 8. 1. As observed from the results, the optimization improves performance by reducing the total time taken for change detection for trees having more number of leaf nodes and more number of common second level sub-trees.

Table 8. 1 Effect of increase in sub-trees of tree on optimization

	No of second level sub-trees				
	10	20	25	30	40
Without Optimization (change detection time in ms)	307	655	926	1185	2055
With Optimization (change detection time in ms)	256	632	886	1139	1655
Leaf nodes: 101-401 Sub-trees: 10-40 nodes in Sub-tree: 10 common second level sub-trees: n-3 where n is the number of sub-trees.					

b. Deep trees: To observe the effect of optimization on the increase in height of the tree, a tree containing a small number of leaf nodes was selected. The dataset characteristics consisted of 1) leaf node: 20 2) Sub-trees: 5 each having size of 4 leaf nodes 3) Common second level sub-trees: 2 (hence 8 nodes were common) and 4) depth was varied from 5 – 30. The observed performance is shown in Table 8. 2.

As can be inferred from the results, the performance improvement due to optimization is not significant for deep trees having small number of leaf nodes. It is observed that the cost after optimization increases in some cases as the number of matching leaf nodes in the common second level sub-trees is small due to the small size of the sub-tree. The increase in cost is due to additional cost is incurred for checking the common second level sub-trees, negating the small improvement due to the removal of common second level sub-trees. As the height of the tree increases, the parsing cost increases. Hence, the optimization is not effective for deep trees having small number of leaf nodes. But it has been observed that for deep trees containing large number of leaf nodes, the optimization improves the performance. Hence, we can conclude that the optimization works effectively for trees having large number of leaf nodes and more second level sub-trees.

Table 8. 2 Effect of increase in depth of tree on optimization

	Depth (height –1) of a tree			
	5	10	20	30
Without Optimization (change detection time in ms)	87	88	115	160
With Optimization (change detection time in ms)	91	84	114	166
Leaf nodes: 21 Sub-trees: 5 nodes in Sub-tree: 4 common second level sub-trees: 2				

The optimization is based on the fact that as the number of common second level sub-tree nodes are removed; the amount of time taken for computing the longest common subsequence (LCS) is reduced, decreasing the overall cost of change detection. To test this hypothesis, trees were generated with the following characteristics a) leaf nodes: 300 b) second level sub-trees: 30 c) depth: 5 d) common second level sub-trees: range 1- 27 and e) Change operations: were increased in consistent manner as the number of common sub-trees decreased. These data characteristics were selected based on the characteristics of the actual XML web pages observed on the Internet (e.g. ACM Sigmod XML repository). As the performance observed was for the same tree, the parsing cost was not considered. The observed performance on optimization with increase in matching second level sub-trees on the same tree is shown in graph given in Figure 8. 1. From the graph, it can be clearly observed that there is a considerable improvement in performance with increase in matching second level sub-trees.

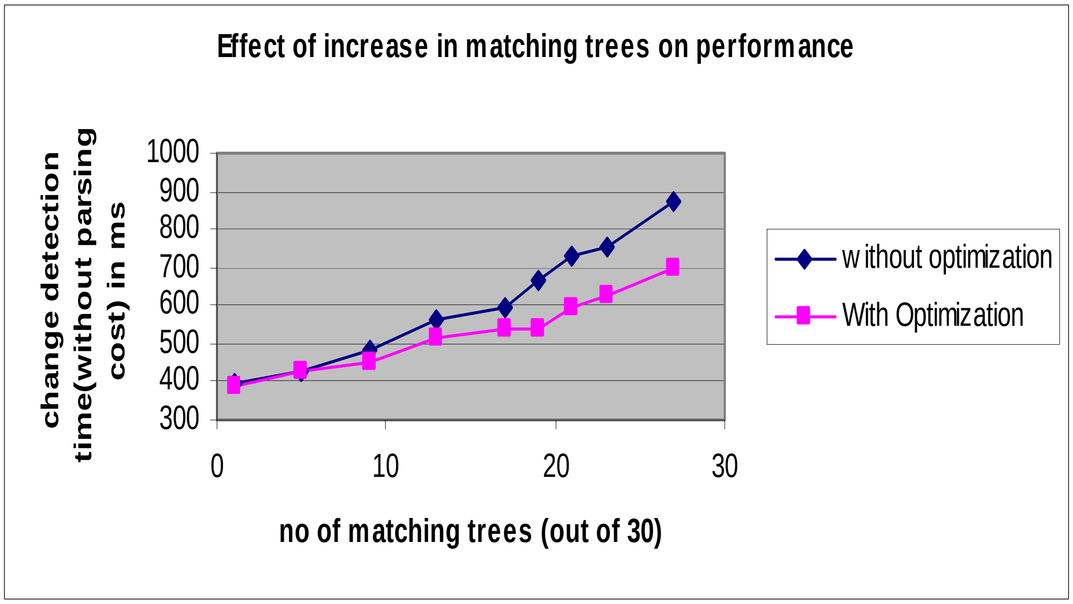


Figure 8. 1 Effect of increase of matching sub-trees on optimization

In addition to optimization, in the primary algorithm contains a phase for pruning unique inserts and deletes. This phase is intended to reduce the amount of time taken for change detection by pruning the nodes resulting in unique inserts/deletes and hence decreasing the number of nodes for LCS. The performance test was carried out for two XML trees with the following characteristics a) leaf nodes: 300 b) second level sub-trees: 30 c) depth: 5 d) common second level sub-trees: range 1- 27 and e) change operations: were increased in consistent manner as the number of common sub-trees decreased. The performance graph is shown in Figure 8. 2.

As observed from the graph, a performance improvement of 55% in terms of execution time was observed. This improvement is due to the increase of unique inserts/deletes with the increase in change operations, resulting in reduction of time taken for change detection. This is because, as more number of nodes result in unique inserts/deletes, they are filtered out at the second level. Hence, the number of nodes for LCS decreases considerably leading to an overall decrease in the time taken for change detection.

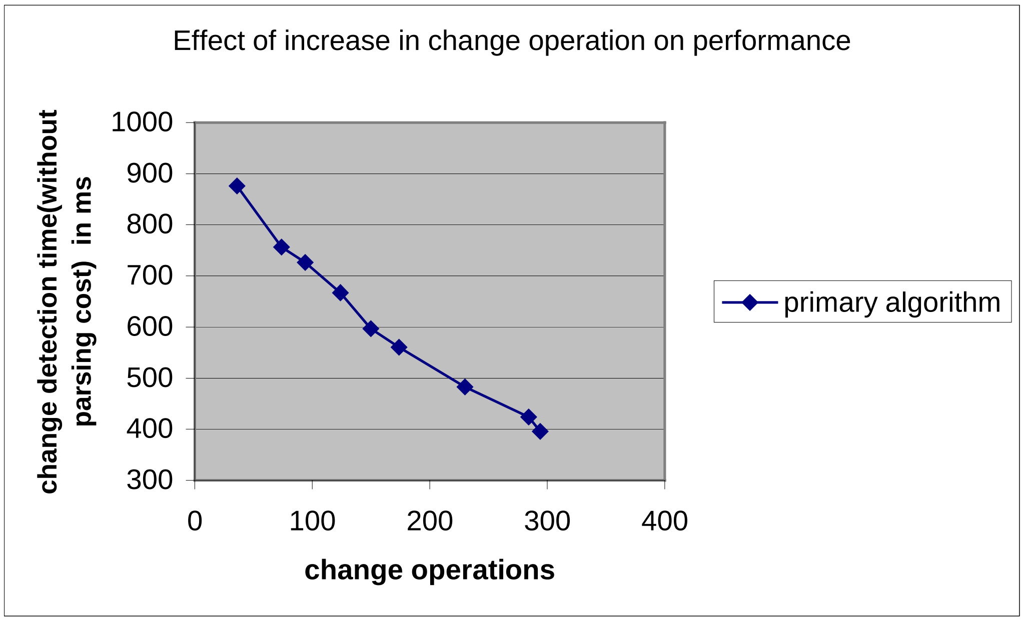


Figure 8. 2 Effect of increase of change operations

If the optimization technique is applied to this primary algorithm, then as the number of change operations increase, the time taken for change detection converges with the time taken without optimization as can be seen in the graph shown in Figure 8.3. This is because, as the number of change operations increase for a tree, the number of matching (common) second level sub-trees decrease. As the number of nodes removed due to common second level sub-trees is less, the time taken with optimization is almost equal to the time taken without optimization.

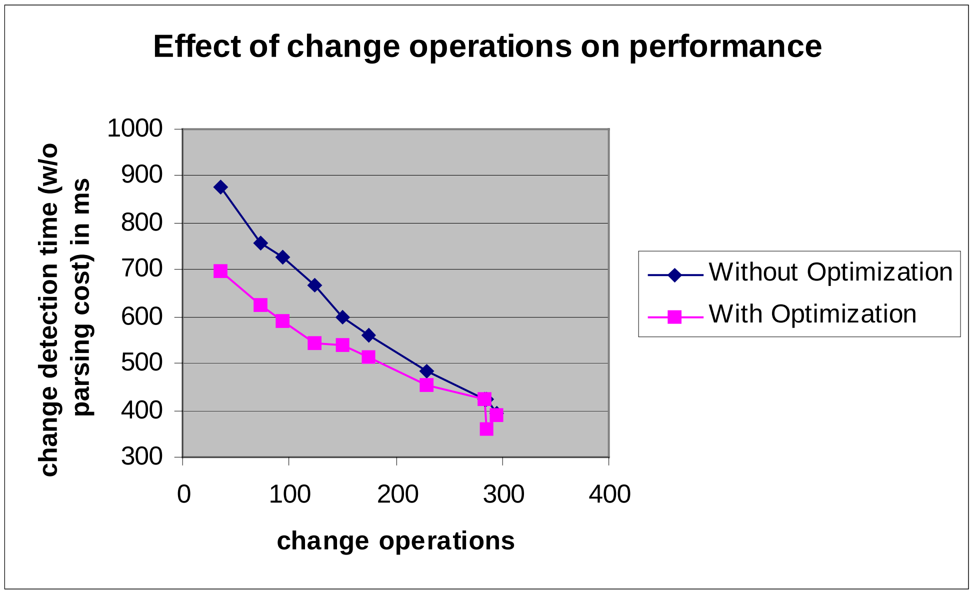


Figure 8. 3 Effect of increase of change operations with optimization

8.2 Future Work

The input to the parser for parsing the sentinel specification is a simple text file. As WebVigiL is designed for large network centric environment such as the Internet, the users need a flexible web browser enabled interface for submitting the specification. In addition, the user should not be burdened with remembering all the options that can be specified in a sentinel. Hence, a user-friendly interface is needed; using which the user can specify easily his/her policies without invalidating the syntax or semantics of the sentinel. We have designed and are developing a user-friendly web browser enabled interface using various client and server-side scripting languages. This interface will authenticate the user, extract information from the inheritance module and display to the user, if user desires to inherit the properties from previously specified sentinels and will allow the user to specify his/her policies with ease. The interface will then call the APIs of the validation modules. Once validated, the information will be stored in the Knowledgebase.

8.2.1 Change detection to XML documents

The present change detection mechanism supports changes to the contents of an XML document. This can be extended to detect changes to the presentation and content-based tag objects such as tables, list, links and images. For this purpose, additional information regarding the presentation and defined structure of the documents is needed which is available in the DTD or the presentation documents such as the style sheet or html, which embeds the XML file for presentation. In addition, if the DTD or XML schema information of a particular document is available, then queries such as "Notify me when the price of a particular stock increases or decreases by a particular ratio" on stock pages etc. can be supported.

At present, the level of granularity for change detection is a page. We support customized changes on a single document. This can be extended to detecting composite changes (more than one type of change) on multiple XML document for a particular specification. Hence, the user can specify correlated changes on multiple XML documents in a single specification.

8.2.2 WebVigiL

Most of the modules as discussed in this thesis have been implemented. The design of change detection graph is ready and will be implemented soon. We are at present finalizing the design of the presentation/notification module and a naïve version controller. We plan to have a complete running system in a couple of months.

APPENDIX A

TOTALMATCH AND SIGNATUREMATCH ALGORITHMS FOR KEYWORDS

TotalMatch algorithm

```
Input: N1= {all lead node values with position and associated signatures in tree T1}
N2 = {all leaf node values with position and associated signatures in T2}
K1 = {extracted keywords in T1},  K2 = {extracted keywords in T2}.
Output : Matching nodes set in T1 and T2
        /* Initialize the sets */
            $M_{T1}$ = {}  /* matching nodes of T1 /, $M_{T2}$ = {} / matching nodes of T2

    1.  Concatenate the value of the leaf node with its associated signature in T1 and generate its
        java hashCode.
    2.  Concatenate the value of the leaf node with its associated signature in T2 and generate its
        java hashCode.
    3.  Sort the elements in T1 and T2 based on their hashCode
    4.  lowerbound = hashCode(the first element of T1)
    5.  upperbound = hashCode(the last element of T1)
    6.  For all element in T2 such that hashCode(element) >= lowerbound || hashCode(element)
        <= upperbound
    7.      find matching elements in T1 and T2 such that their hashCode match.
            // finding all matching nodes in T1 and T2 such that their value match and //
            // associated signatures match.
    8.      $M_{T1}$=matching elements in T1
    9.      $M_{T2}$=matching elements in T2
    10. For $\forall$ element n in $M_{T1}$
    11.     k1 = extractKeyword(n,K1)
    12.     flag k1 as “matched”
    13. For $\forall$ element m in $M_{T2}$
    14.     k2 = extractKeyword(m,K2)
    15.     flag k2 as “matched”
```

SignatureMatch Algorithm

```
Input: N1= {non-matching nodes in T1 after totalMatch}, D= {non-matching nodes in T2
after totalMatch} $M_{T1}$ = {matching nodes in T1 from totalMatch},
$M_{T2}$ = {matching nodes in T2 from totalMatch}
K1 = {extracted keywords in T1}, K2 = {extracted keywords in T2}.
I = Insert set , D = Delete set.
    1.  Generate hashCode of the signature of all elements in N1
    2.  Generate hashCode of the signature of all elements in N2
    3.  Sort N1 and N2 based on their hashCode
    4.  Find matching elements in N1 and N2 such that
    5.      hashCode(path(element in N1)) = hashCode(path(element in N2))
    6.      $M_{T1}$ = matching elements in N1
    7.      $M_{T2}$ = matching elements in N2
    8.      Extract corresponding keywords of matching elements and flag as “matched”
    9.  D = Rest of keywords in K1 with their position.
    10. I = Rest of the keywords in K2 with their position
```

APPENDIX B

FINDING THE COMMON ORDER SUBSEQUENCE AND DETECTING CHANGE OPERATIONS FOR KEYWORDS

Finding the Common Order Subsequence

Input: M_{T1}= {matching nodes in T1}, M_{T2} = {matching nodes in T2},
K1 = {extracted keywords in T1}, K2 = {extracted keywords in T2},

```
/* Initialize the sets */
NoSeq1 = {} /* keywords not in the common order subsequence  in T1*/
NoSeq2 = {} /* keywords not in the common order subsequence  in T2*/

1. if ($\forall$ k1 in K1 is flagged as "deleted" || $\forall$ k1 in K2 is flagged as "inserted")
2.      Stop
3. Else
4.       Sort the signature, value and keyword/s associated with $\forall$ element in $M_{T1}$ and $M_{T2}$ in
         the order of the position of occurrence in the tree T1 and T2.
5.       LCS($M_{T1}$, $M_{T2}$)
6.       NoSeq1 = all keywords in $M_{T1}$ $\notin$ common order subsequence
7.       NoSeq2 = all keywords in $M_{T2}$ $\notin$ common order subsequence
8.       Call function changeOperation
```

ChangeOperation // Detecting inserts and deletes

Input: NoSeq1 = { keywords not in the common order subsequence in T1}
NoSeq2 = { keywords not in the common order subsequence in T2}

Output = I: Insert set, D: Delete set

```
1. For $\forall$ k1 in NoSeq1
2.       Add (k1,position(k1)) in D
3. For $\forall$ k1 in NoSeq2
         Add (k2,position(k2)) in I
```

APPENDIX C

PSEUDO-CODE FOR FINDING THE ABSOLUTE TIME OF AN EVENT

CalculateEventTime
Input: eventStatus , event , offset
Output : absolute total time   \\ absolute time + offset if offset I is not zero

```
1.  If offset ≠ null
2.  Convert the offset into appropriate date format I
3.  Check if the event has already lapsed
4.  if lapsed
5.      Extract appropriate time from wv_sentinelstatus table, calculate absolute time
        and return
6.  else if event = start(sentinel s)
7.       Extract the start properties T of s from wv_sentinelstart table
8.  else if event = end (sentinel s)
9.       Extract the end properties T of s from wv_sentinelend table
10. else if event = during
11.      Extract the sentinel creation time.
12.      Calculate absolute total time and return
13. If T is Now
14.      Extract sentinel creation time, calculate absolute total time and return
15. else if T is timepoint
16.  Calculate the absolute total time and return
17. else if T is relative
18.      Convert the offset into appropriate date format and add it to I
19.      Calculate absolute total time and return
20. else if T is event
21.   repeat steps 4 - 16
```

REFERENCES

1. Chakravarthy, S., et al. *WebVigiL: An approach to Just-In-Time Information Propagation In Large Network-Centric Environments*. in *Second International Workshop on Web Dynamics*. 2002. Hawaii.
2. Jacob, J., et al., *WebVigiL: An approach to Just-In-Time Information Propagation In Large Network-Centric Environments(to be published)*, in *Web Dynamics Book*. 2003, Springer-Verlag.
3. *Extensible Markup Language(XML).*, World Wide Web Consortium,http://www.w3.org/XML/.
4. Document Object Model., http://www.w3.org/DOM/.
5. Chakravarthy, S. and D. Mishra, *Snoop: An Expressive Event Specification Language for Active Databases.* Data and Knowledge Engineering, 1994. 14(10): p. 1--26.
6. Chakravarthy, S., et al., *Composite Events for Active Databases: Semantics, Contexts and Detection*, in *Proc. Int'l. Conf. on Very Large Data Bases VLDB.* 1994: Santiago, Chile. p. 606--617.
7. Tanpisut, W., *Design and Implementation of Event based subscription/notification paradigm for distributed environments*. 2001, The University of Texas at Arlington.
8. Krishnaprasad, V., *Event Detection for Supporting Active Capability in an OODBMS: Semantics, Architecture, and Implementation*, in *MS Thesis*. 1994, Database Systems R&D Center, CIS Department, University of Florida, Gainesville, FL 32611.

9. Chakravarthy, S., et al., *WebVigiL: Architecture and Functionality of a Web Monitoring System.*

10. Chakravarthy, S., et al., *HiPAC: A research project in active, time-constrained database management*. 1989, Tech. Report (89-02), Xerox Advanced Information Technology: Cambridge.

11. Wells, D., et al., *Architecture of an Open Object-Oriented Database Management System.* IEEE Computer, 1992. 25(10): p.74--81.

12. Hanson, E., *The Ariel Project*, in *Active Database Systems - Triggers and Rules For Advanced Database Processing*. 1996, Morgan Kaufman Publishers Inc. p. 63--86.

13. Pavan Deolasee, A.K., Ankur Panchbudhe, Kirthi Ramamritham and Prashant Shenoy. *Adaptive Push-Pull: Disseminating Dynamic Web Data*. in *Proceeding of the 10th International WWW Conference*. 2001. Hong Kong.

14. Changedetection.com, *http://www.changedetection.com*.

15. Liu, L., et al. *Information Monitoring on the Web: A Scalable Solution*. in *World Wide Web*. 2002.

16. Douglis, F., et al., *The AT&T Internet Difference Engine: Tracking and Viewing Changes on the Web*, in *World Wide Web.* 1998, Baltzer Science Publishers. p. 27-44.

17. Hirschberg, D., *Algorithms for the longest common subsequence problem.* Journal of the ACM, 1977: p. 664-675.

18. S.Abiteboul, P.Buneman, and D.Suciu, *Data on the Web: From Relations to Semistructured Data and XML*. 1999: Morgan Kaufmann.

19. K.Zhang and D.Shasha, *Simple Fast Algorithms for the Editing Distance between Trees and Related Problems.* SIAM Journal of Computing, 1989. 18(6): p. 1245-1262.

20. K.Zhang, R.Statman, and D.Shasha, *On the Editing Distance between Unordered Labeled Trees.* Information Processing Letters, 1992. 42: p. 133-139.

21. S.Chawathe, et al. *Change detection in hierarchically structured information*. in *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 1996. Montréal, Québec.

22. E.Myers, *An O(ND) difference algorithm and its variations.* Algorithmica, 1986. 1: p. 251-266.

23. Y.Wang, D.DeWitt, and J.Cai, *X-Diff: An Effective Change Detection Algorithm for XML Documents*. 2001, Technical Report, University of Wisconsin.

24. Rivest, R., *The MD5 Message-Digest Algorithm*. 1992, http://www.faqs.org/rfcs/rfc1321.html.

25. Eastlake, D. and P. Jones, *US Secure Hash Algorithm 1 (SHA1)*. 2001, http://www.faqs.org/rfcs/rfc3174.html.

26. G.Cobena, S.Abiteboul, and A.Marian, *Detecting Changes in XML Documents.* Data Engineering, 2002.

27. F.P.Curbera and D.A.Epstein, *Fast Difference and Update of XML Documents.* XTech'99, 1999.

28. H.Maruyama, K.Tamura, and R. Uramoto, *Digest values for DOM (DOMHash) proposal*. 1998, IBM Tokyo Research Laboratory.

29. XMLDiff.

30. Nguyen, B., et al. *Monitoring XML Data on the Web*. in *Proceedings of the 2001 ACM SIGMOD International Conference on Management of Data*. 2001.

31. S.Chawathe, S.Abiteboul, and J.Widom, *Representing and Querying Changes in Semistructured Data.* ICDE, 1998.

32. *Java Compiler Compiler™ (JavaCC) - The Java Parser Generator*., http://www.webgain.com/products/java_cc/.

33. Xerces-J, *http://xml.apache.org/xerces2-j/index.html*.

34. Java1.3, *http://java.sun.com/j2se/1.3/docs/api/.*

BIOGRAPHICAL INFORMATION

Jyoti Jacob received her Bachelor of Engineering degree in Computer Science and Engineering from Sardar Patel University, Gujarat, India in August 1997. In the Spring of 2001, she started her graduate studies in Computer Science and Engineering at The University of Texas, Arlington. She received her Master of Science in Computer Science and Engineering from The University of Texas at Arlington, in May 2003. Her research interests include XML, active database techniques, data on the web and tree structures.