A DATAFLOW APPROACH TO EFFICIENT CHANGE DETECTION OF HTML/XML DOCUMENTS IN WEBVIGIL

The members of the Committee approve the master's
thesis of Anoop Sanka

Sharma Chakravarthy
Supervising Professor ______________________________

Leonidas Fegaras ______________________________

Alp Aslandogan ______________________________

A DATAFLOW APPROACH TO EFFICIENT CHANGE DETECTION OF HTML/XML DOCUMENTS IN WEBVIGIL

by

ANOOP SANKA

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

December 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. Leonidas Fegaras and Dr. Alp Aslandogan for serving on my committee.

I am grateful to Ramji Beera, Raman Adaikkalavan, Hari Prasad Yalamanchali, Naveen Pandrangi, and Jyoti Jacob for their invaluable help and advice during the implementation of this work. I would like to thank all my friends in the ITLAB 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 (grant IIS-0123730) for this research work.

I am thankful to my parents and brother for their constant support and encouragement throughout my academic career without which I would not have reached this position.

August 12, 2003

ABSTRACT

A DATAFLOW APPROACH TO EFFICIENT CHANGE DETECTION OF HTML/XML DOCUMENTS IN WEBVIGIL

Publication No. ______

Anoop Sanka, MS

The University of Texas at Arlington, 2003

Supervising Professor: Supervising Professor Name

Data on the web is constantly increasing. Many a times, users are interested in specific changes to the data on the web. Currently, in order to detect changes of interest, users have to poll the pages periodically and check for the changes they are interested in. Efficient and effective change detection and notification is critical in many environments where a lot of resources are wasted in monitoring changes to the web manually. WebVigiL is a change monitoring system, which efficiently monitors changes to the page on behalf of the user and notifies the changes in a timely manner. It is a general-purpose, server based information monitoring and notification system.

This thesis investigates how active capability (ECA Rules) has been adapted for change monitoring. WebVigiL supports several types of changes such as keywords, phrases, links, images, and any change. A change detector, which facilitates monitoring primitive (above types) and composite (combinations of above types) changes to HTML/XML pages has been designed and implemented. Algorithms for detecting composite changes are discussed. Grouping techniques for efficient change detection are also discussed.

TABLE OF CONTENTS

LIST OF ILLUSTRATIONS

LIST OF TABLES

CHAPTER 1

INTRODUCTION

The World Wide Web has become one of the most important media for sharing information resources and continues to grow at an alarming rate. Users surfing the web, may either be searching for specific information or simply browsing the web. Different users may be interested in knowing changes to specific web pages and their contents (or even combinations there-of), and want to know when those changes take place. For example, tech-savvy users may want to monitor new technologies and new research results from engineering fields and business information of competitors. Such information would be essential for maintaining the competitive edge. Students may want to know when the web contents of the courses (they have registered for) change; users may want to know when news items are posted in a specific context (appearance of key words, phrases etc.) they are interested in. This needs periodic polling of the web (i.e., retrieval of one or more pages) to see whether the information has changed. Generally, to discover information of interest the users need to constantly monitor certain web sites and web pages. This is a drain on the bandwidth as well as labor intensive.

In large software development projects, there exist a number of documents, such as requirements analysis, design specification, detailed design document, and implementation documents. The life cycle of such projects is in years (and some in decades) and changes to various documents of the project take place throughout the life cycle. Typically, a large number of people are working on the project and managers need to be aware of the changes to any one of the documents to make sure the changes are propagated properly to other relevant documents and appropriate actions are taken. Large software developments happen in distributed environments.

Today, information retrieval is mostly done using the pull paradigm, where the user is responsible for posing the appropriate query (or queries or initiating the search) to retrieve needed information. The burden of knowing changes to the contents of pages in interested web sites is on the user, rather than on the system. Although there are a number of applications (airlines, for example) that selectively send interested information periodically, the approach typically uses a mailing list to send the same information to *all* users. Other tools that provide real-time updates in the web context (e.g., stock updates) are customized for a specific purpose and have to be running continuously and underneath still uses a naïve pull paradigm to refresh the screen periodically. In general, the ability to specify changes to arbitrary documents and get notified according to user-preferred ways will be useful for reducing/avoiding the wasteful navigation of web in this information age. In other words, users are interested in a variety of information from different sources and there is a real need for systems to be developed to support the task of automatically identifying changes and notifying the changes to the users in a timely and effective manner. The proposed system – WebVigiL provides a powerful way to disseminate information efficiently without sending unnecessary or irrelevant information to the users. It also frees the user from having to constantly monitor for changes using the pull paradigm.

Active rules have been proposed as a paradigm to satisfy the needs of many database and other applications that require a timely response to situations. Event–Condition–Action (or ECA) rules are used to capture the active capability in a system. The utility and functionality of active capability (ECA rules) has been well established in the context of databases. In order for the active capability to be useful for a large class of advanced applications, it is necessary to go beyond what has been proposed/developed in the context of databases.

In the case of large-scale network centric environments such as web, users might be interested in monitoring changes to a particular page or a part of the page such as images, links, keywords and etc. In most cases, user's interest may not pertain to images, links and keywords only, but to a combination of them. Web pages that are monitored for detecting the changes may be of type HTML or XML. Changes to pages and changes in images, links, keywords and etc., act as the primitive events themselves when mapped to the ECA rules and their combination form composite events. Thus, some of the techniques developed for active databases, when extended appropriately will provide a solution to detect changes in the web. This thesis focuses on developing a framework and to provide a selective propagation approach to detect changes that are of interest to the users in the web and other large-scale network-centric environments by adapting and extending the existing active technology.

This thesis is organized as follows. Chapter 2 explains the existing change detection and notification systems and explains how WebVigiL differs from the rest. Chapter 3 presents an overview of the current WebVigiL architecture. Chapter 4 explains the ECA Rule generation and how Local Event Detector (LED) is used to facilitate the process. Chapter 5 discusses the various approaches and the architecture of the Change Detection Graph used for detecting primitive and composite changes. Chapter 6 explains the Storage and Retrieval of pages used for storing the pages fetched for monitoring. Chapter 7 extends the above to explain the implementation details. Chapter 8 has conclusions and future work.

CHAPTER 2

RELATED WORK

Several tools are available to assist users to track when the web pages of interest have changed. Most of the tools offer service from a centralized server or a client's machine. Client-based tools focus mainly *when* to fetch the pages of interest rather than *how* the pages have changed. This is because of the complexity involved in keeping track of changes to the content for numerous versions. If specific changes to a page have to be detected, a differencing tool has to run on the client machine. In spite of having such a tool, when the user wants to track composite changes (for example, links AND images change on a page) additional information has to be maintained. And as the requirements grow, the complexity at the client end also increases. This gave way to the server-based systems. Server based tools track pages that are previously registered or submitted by users and notifies them via email or over the web upon request. The following are some of the server-based tools developed for change monitoring to web pages.

2.1 AIDE (AT&T Internet Difference Engine)

AIDE [1] is both client and server based. It is a collection of tools. The tools consist of: w3newer, which detects changes to pages; snapshot, which permits a user to store a copy of the page and to compare any subsequent versions of the page; HtmlDiff, the differencing tool used to compute the changes between two pages. W3newer runs on the client machine and when observes a change on a page, it informs the snapshot to save a copy. Snapshot is an external service that archives the versions of the page and whenever a new version of a page arrives to the system through w3newer, it invokes the HtmlDiff. When the user requests the changes, the snapshot is contacted. The user could obtain the difference between any versions of the page. Because of its architecture the necessity of grouping users who monitor the same page does not arise. The drawbacks of this system are that the user cannot specify customized changes (links, images, keywords) or composite change (links AND images) on a page. Changes to XML pages are not supported.

2.2 WebGUIDE

WebGUIDE is an extension to AIDE. It consists of the following tools: AIDE and Ciao [2]. Ciao is a graphical navigator that allows users to query and browse structural connections embedded in a document repository. The same drawbacks described for AIDE apply to this system as well.

2.3 NetMind

NetMind [3] formerly known as URL-minder provides keyword or text-based change detection and notification service over web pages. NetMind detects changes to links, images, keywords and phrases in an HTML page. The medium of notification to users about the change is via e-mail or mobile phone. The user might be interested in a change to a page but not when there are changes to articles or some set of words the user is not interested. Such change detection request cannot be specified. There is no support for composite changes (when both links AND images change) on a page. There is no provision for the user to come back later and view the last changes that have been detected. The frequency of when to poll the page is predefined. The user cannot explicitly specify when to poll the page for change detection and on what versions of page the change should be computed. Since the implementation is hidden behind a CGI interface, how changes are detected is not known. Change detection to XML pages is not supported.

2.4 WebMon

WebMon [4] is proposed for tracking information change over the Internet. The user can specify the web page to be monitored, select the monitoring function and state the monitoring frequency. The monitoring function can be any customized change such as change in the time stamp, links, images or phrases. The change detection is based on the structure of the page. The assumption is that, HTML pages have stable structure. Issues such as grouping users having overlapping monitoring requests are not considered. Change detection to XML pages is not supported. A combination of changes on a page is not supported.

2.5 WebCQ

WebCQ [5] is a prototype system for large-scale web information monitoring and delivery, which makes use of the structure present in hypertext and the concept of continuous queries. WebCQ is designed to discover and detect changes to the web pages and to provide a personalized notification of the changes to the users. Users monitoring requests are modeled as continuous queries on the web. WebCQ change detection robot is responsible for discovering and detecting changes to web pages. The authors specify that composite changes can be detected, but currently the system does not seem to support them. WebCQ lacks a fine grouping strategy. For example, the change is computed more than once for two users having the same set of keywords. The grouping is based on a single keyword rather than a set of keywords. Only change detection to HTML documents is supported. The user has to set the polling frequency explicitly, the system does not tune to the change frequency of the page i.e., the system does not learn from the polling patterns. The input specification language is limited. The user cannot specify the monitoring request to be dependent on the status of other monitoring requests. Change is always computed between the successive versions of the page. The user cannot specify *what* versions (window concept) of the page should participate in change computation.

2.6 Xyleme

In [6], the authors present a Dynamic Warehouse for Web -- Xyleme, which monitors the flow of incoming documents. The flow of documents consists of XML pages and HTML pages. The authors present a subscription language for specifying the pages to be monitored. Depending on type of information requested by the user the pages are monitored using either monitoring or continuous query. For query of type monitoring, changes to a page are discovered when the system reads the page and for continuous queries changes are discovered by regularly asking the same query. Composite changes are also supported. The user cannot specify the monitoring request to be dependent on the status of other monitoring requests. The users also cannot specify what versions of the page should participate in change computation.

The following are some of the other distinct characteristics of WebVigiL:

1) Properties of monitoring requests can be inherited: The user has the option of specifying the monitoring request to be dependent on the status of other monitoring requests. One can specify the start/end of a request to be the start/end of another request.

2) Flexible specification of versions: All the above systems compute changes between two successive pages. In WebVigiL the user can explicitly specify the pages that can participate in change detection.

None of the above systems except Xyleme use the ECA (Event-Condition-Action) paradigm for monitoring the web. ECA Rules help in adding new functionality to the system seamlessly.

CHAPTER 3

WEBVIGIL ARCHITECTURE

WebVigiL is a change detection and notification system, which can monitor and detect changes to unstructured 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. Figure 3.1 summarizes the high level 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 stored in a data/knowledge base (currently Oracle) and is used by the other modules in the system. The functionality of each module in the architecture shown in Figure 3.1 is described briefly in the following sections.

3.1 Sentinel

WebVigiL provides an expressive language with well-defined semantics for specifying the monitoring requirements pertaining to the Web. Each monitoring request is termed a *Sentinel*. Briefly, the specification language supports the following features:

- A suite of change types at appropriate levels of granularity that are of interest to a large class of users. For example, changes only at the level of a page may be overkill in many cases. One may be looking for changes to keywords or phrases of interest. Also multiple or composite changes (for example, changes to links and changes to images) to a page are supported.

- Ability to monitor a page based on the actual change frequency, or at a user-specified frequency. 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. A learning algorithm (based on history) is used for this purpose.
- Multiple notification paradigms such as e-mail, fax, dashboard etc.
- Multiple ways to compare changes (e.g., pair-wise, every n, or moving n).
- Specification of a sentinel in terms of previously defined sentinels. Also, start and stopping of a sentinel may be based on other sentinels. This provides a mechanism for tracking correlated changes.

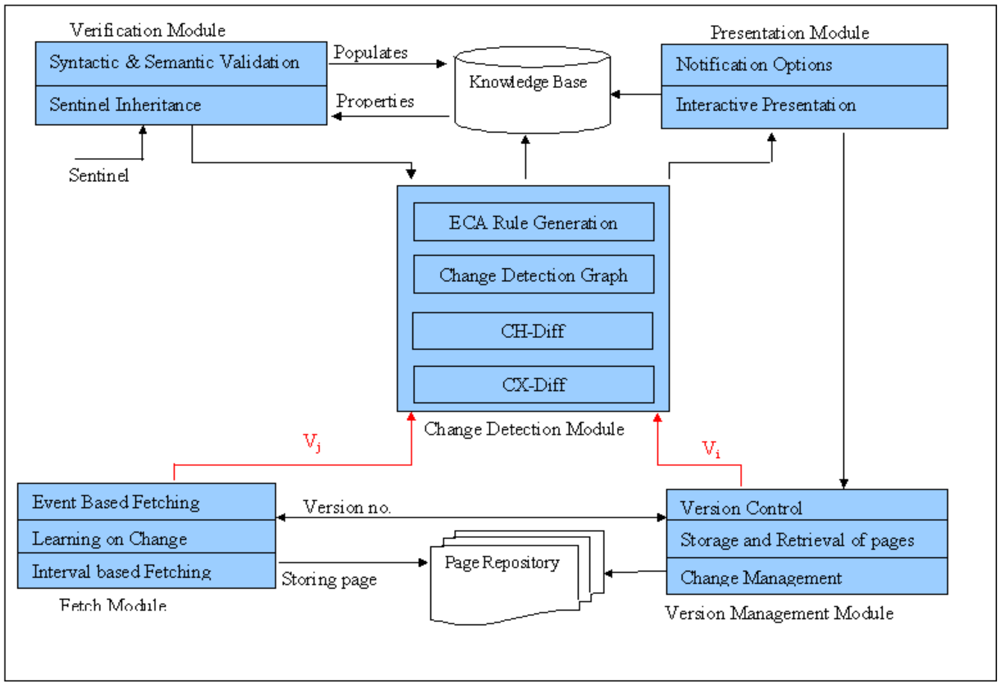


Figure 3.1 WebVigiL Architecture

For example consider the Scenario: Jill wants to be notified daily by e-mail about changes to links and images to the page “http://www.cnn.com” starting from December 2, 2002 to January 2, 2003. The sentinel generated for the above scenario is as follows:

```
Create Sentinel s1 Using http://www.cnn.com
Monitor all links AND all images
Fetch every 2 days
From 12/02/02   To 01/02/03
Notify By email jill@aol.com  Every 4 day
Compare pairwise
```

A detailed explanation is given in [7].

3.2 Verification Module

Verification module provides the required communication interface between the system and the user for specification of sentinels. User requests (sentinels) are processed for syntactic and semantic correctness. Valid sentinels are populated in Knowledge base (Oracle is used currently) and a notification of the valid sentinels is sent to the change detection module. In general the functionality of verification module can be summarized as

- Load balancing of syntactic validation between client and server, thereby reducing excessive communication between the client and the server (e.g., validating start date set to a date in past at the client’s end).
- Semantic validation of sentinels at the server, as the dependency information specified in the sentinel is available at the server. For example if the start of a sentinel s1 was specified on the end of another sentinel s2, and at the time of specification if s2 had already expired an error should be thrown to the user.

3.3 Knowledge Base:

Knowledge Base is a persistent repository containing meta-data about each user, number and names of sentinels set by each user, and details of the contents of the sentinel (frequency of notification, change type etc.). The details of a sentinel need to be stored (in a persistent and recoverable manner) as several modules use this information at run time. 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 installation 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.

To satisfy all the above requirements, the metadata (the WebVigiL Knowledge Base) generated and used by different modules is stored in a relational DBMS. The monitoring request is parsed and sentinel properties are extracted, validated, and stored in the KB. For example, the following parameters are stored for notification: the frequency of notification and the mechanism to notify the user. In addition, important run time parameters computed by different modules, such as the status of the created sentinels and parameters of the change detection module are also persisted in the KB. Finally, relational database provides mechanisms to extract the required information in a convenient manner in the form of queries or using the JDBC Bridge.

3.4 Change Detection Module

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 ECA rules [8] to perform some of these operations. This module is responsible for:

1. Activating and deactivating sentinels
2. Constructing and Maintaining Change Detection Graph
3. Generating Fetch rules.

This will be detailed in later chapters of this thesis.

3.4.1 Detection algorithms

A detection algorithm associated with each change type computes changes between two versions of a page with respect to that change type. For a change to be detected, the object of interest is extracted from the available versions of the page depending upon the change type. Change detection algorithms have been developed to detect different types of changes to HTML and XML pages. The change types currently supported are: links, images, all words, keywords and phrase. Change to links, images, words and keyword(s) is captured in terms of insertion or deletion. For phrases in addition to insertion/deletion updates are also detected. Refer [7, 9] for more detail.

3.5 Fetch Module

The Fetch Module [9] of WebVigiL is responsible for retrieving the pages registered with it and thus serves as a local wrapper for the task of fetching pages depending upon the user set fetching policy i.e., fetching a page after a specified interval (set by the user) or fetching the page on change (the system determines the frequency of fetching based on actual change frequency of the pages). The Fetch module informs the version controller of every version it fetches, stores it in the page repository and notifies the change detection graph (or CDG) of a successful fetch. The wrapper fetches the page only when there is change in the properties of the pages. By properties, we mean the size of the page and the last modified time stamp. When there is a change in time stamp of the page with an increase or decrease in page size, the wrapper fetches and caches the page. In cases where time stamp is modified, but the page size remains the same, the wrapper fetches and calculates the checksum of the page. This version of the page is cached only if the calculated checksum differs from the checksum of the cached (previous) version of this page.

3.6 Version Management

An important feature of WebVigiL architecture is its server-based repository service (Version controller) 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, thereby 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.

The repository service reduces network traffic and latency for obtaining the web page because WebVigiL can obtain the “Target Web Pages” from the cache instead of having to request the page directly from the remote server. The quality of service for the repository service includes managing multiple versions of pages without excessive storage overhead.

3.7 Presentation Module

The principal functionality of this module is to clearly present the detected differences between two web pages to the user. Therefore, computing and displaying the detected differences is very important.

3.7.1 Change Presentation

Different methods of displaying changes used by the existing tools are: i) merging two documents, ii) displaying only the changes and iii) highlighting the differences in both the pages. Summarizing the common and changed data into a single merged document has the advantage of displaying the common portions only once. The disadvantage of this approach is that it is difficult for the user to view the changes when they are large in number. Displaying only the computed differences is a better option when the user is interested in tracking changes to multiple pages or when the number of changes is large. But, highlighting the differences by displaying both the pages side-by-side is preferable for changes like “any change” and “phrase change”. In this case, the detected differences can be perceived better if the change in the new page is shown relative to the old page.

Because WebVigiL will track multiple types of changes on a web page, and eventually notify using different media (email, PDA, laptop etc.), combination of all presentation styles discussed above will be relevant, as the information to be notified will vary depending on factors such as notification method, number of detected differences and type of changes.

3.7.2 Change Notification

Users need to be notified of detected changes. The mechanism selected for notification is important especially when multiple types of devices with varying capabilities are involved. What, when and how to notify are three important issues for notification.

3.7.2.1 Presentation Content

Presentation content should be concise and lucid. Users should be able to clearly perceive the computed differences in the context of his/her predefined specification. The notification report could contain the following basic information:

- The change detected in the latest page relative to the reference page
- User specified type of change like “any change”, “all words” etc.
- URL for which the change detection module is invoked.
- Small summary explaining the detected changes.

This could include status of changes such as insert, delete and changed for certain type of user-defined types of changes such as “images”, “all links” and “keywords” and/or the different timestamps indicating the modification, polling, change detection and notification date. The size of the notification report will depend upon the maximum information that can be sent to a user by satisfying the network quality of service requirements.

3.7.2.2 Notification frequency

A detected change can be notified in two ways: i) notify immediately when the change is detected or ii) notify after a fixed time interval. The user may want to be notified immediately of changes on particular pages. In such cases, immediate notification should be sent to the user. Alternatively, frequency of change detection will be very high for web pages that are modified frequently. Since frequent notification of these detected changes may become a bottleneck on the network, it is preferable to send the notification periodically. The notification has to be sent to the user taking into consideration the QoS constraints. The system should incorporate the flexibility to allow users to specify the desired frequency of notification. For example, in sentinel s1, Jill wants to be notified every 4 days, irrespective of when the changes are detected.

3.7.2.3 Notification methods

Different notify options such as email, fax, PDA and interactive, can be used for notification. Interactive is a retrieval-based notification approach where the user retrieves the detected changes as and when needed. A dashboard will be provided to the user to view and query the changes generated by his/her sentinels.

WebVigiL architecture shown in the Figure 3.1 has five modules and many components within them. This thesis deals mainly with change detection module along with fetch and version management modules. In the change detection module, ECA rule generation and change detection graph components are addressed. In the fetch module, event based fetching and in the version management module, storage and retrieval of pages components are addressed. All the modules along with their components are discussed in the following chapters.

CHAPTER 4

ECA RULE GENERATION

Every valid user request arriving at WebVigiL initiates a series of operations that occur at different points in time. Some of these operations are: activating a sentinel (based on start time), monitoring the requested page, detecting changes of interest, notifying the change to the user(s), and deactivating a sentinel (based on end time). In WebVigiL, for every sentinel, the ECA rule generation module generates ECA rules [8, 10] 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 paradigm has been used for monitoring the database state in active databases and as a stand-alone concept for monitoring objects in applications (both centralized and distributed [11]). As part of the Active Object-oriented system [12, 13], a local event detector (LED) has been developed as a library that can be used to declare events and associate rules to be executed when events occur in a seamless manner. It is actually an event detector that has been implemented to detect events in java applications and execute rules defined on them. Primitive events (as method executions) and temporal events (both absolute and relative time), as well as composite events are supported in LED. The existing event specification language "SNOOP" [14] is used for specifying composite events. WebVigiL uses Periodic event operator for change detection and PLUS operator for activation and deactivation of sentinels. ECA rules provide an elegant mechanism for supporting asynchronous executions based on events (temporal or otherwise).

This chapter provides an overview of the tools/components used in this module and discusses how ECA rules are used for: i) activation and deactivation of sentinels, and ii) for generating fetch rules for retrieving pages.

4.1 LED (Local Event Detector)

4.1.1 Primitive Event

An event is an occurrence of interest at a specific point in time. Primitive events are the elementary occurrences and are classified into domain-specific events (e.g., database, oodb, WebVigiL), temporal, and explicit events. Domain-specific events are specific to a domain and are associated with the manipulation of data in that domain (such as the creation, deletion, or insertion that are executed over a period of time in an RDBMS). Event modifiers (begin and end) were introduced to transform operations that take an interval into an instantaneous event. In other words, the event modifiers (begin and end) are used to map the logical events at the conceptual level to physical events. The begin event modifier denotes the starting point of an event and the end event modifier denotes the ending point. Temporal events correspond to absolute and relative temporal events. The absolute temporal event is an event associated with an absolute value of time. For example, 4 P.M. on July 4, 1999 is an absolute event. The relative temporal event is an event corresponding to a specific point on the time line, which is an offset from another time point (specified either as absolute or as an event). Explicit (also termed abstract) events are explicitly defined in an application, but their occurrences are either detected outside of the application or conveyed to the application or the application explicitly raises those events.

4.1.2 Composite Events

A composite event is an event that is composed of primitive events and/or other composite events by applying Snoop [14] event operators such as OR, AND, SEQUENCE, NOT. In order words, the constituent events of the composite event can be primitive events and/or previously defined composite events.

4.1.3 Snoop Event Operators

The event operators are used to construct composite events. Some of these event operators and its *point* semantics [14] are described briefly in the following section. The upper case letter E, which represents an event type, is a function from the time domain on the Boolean values. The function is given by

E (t) = True if an event type E occurs at time point t

False otherwise

- **Conjunction: AND (?)**

Conjunction of two events E_1 and E_2, denoted by E_1 ? E_2 is applied when E_1 occurs and E_2 occurs in any arbitrary order. Formally,

$(E_1 ? E_2) (t) = (E_1 (t1) ? E_2 (t)) ? ((E_1 (t) ? E_2 (t1))$

and $t1 \leq t$

- **Sequence (;)**

The sequence of two events E_1 and E_2, denoted by $E_1 ; E_2$ occurs when E_1 happens before E_2. The timestamp of occurrence of E_1 is less the timestamp of occurrence E_2. Formally,

$(E_1; E_1) (t) = E_1 (t1) ? E_2 (t) \text{ and } t1 < t$

- **Periodic Operator (P)**

The periodic operator, denoted by P (E_1, [t], E_3) is used to express a periodic event that repeats itself within a constant and finite amount of time. The event P is signaled for every amount of time t in the half-open interval (E_1, E_3]. Formally,

$P (E_1, [TI], E_3) (t) = (E_1(t1) ? \sim E_3(t2))$

and $t1 < t2$ and $t1 + x * TI = t$ for some $0 < x < t$ and $t2 \leq t$

where TI is a time specification.

- **Plus (+)**

The plus operator denoted by E_1+ [T] is applied when T time units are elapsed after E_1 occurs.

4.1.4 Parameter Context

Four parameter contexts — recent, chronicle, continuous, and cumulative — were introduced to provide a mechanism for capturing meaningful application semantics and reduce the space and computation overhead for the detection of composite events using the point semantics described above. The contexts are defined by using the notions of initiator and terminator for events. An event that initiates the occurrence of a composite event is termed the initiator of the composite event. An event that completes the detection of a composite event is denoted as the terminator of the composite event. For example, a composite event (E1 ? E2 ? E3) has E1 as initiator and E3 as terminator.

- **Recent**:

  In the recent context, only the most recent occurrence of the initiator (when there are multiple instances of the same event) for any event that has started the detection of that event is used. When the event occurs, all the occurrences of events, those are used in the parameter relation and cannot be initiators of that event in the future, are deleted. In this context, not all occurrences of a constituent event will be used in detecting a composite event. Furthermore, an initiator of an event will continue to initiate new event occurrences until a new initiator occurs.

- **Chronicle**:

  In the chronicle context, the initiator-terminator pair is unique for an event occurrence. The oldest initiator is paired with the oldest terminator for each event. When event occurs, the occurrences of the events are deleted. The event occurrence can be used at most once for computing the parameters of the composite event.

- **Continuous**:

  In the continuous context, each initiator of an event starts a separate detection of that event. A terminator event occurrence may detect one or more occurrences of the same event. The initiator and terminator are discarded after an event is detected.

- **Cumulative**:

In the cumulative context, all occurrences of an event type are accumulated as instances of that event until the event is detected. When the event occurs, all the occurrences that are used for detecting are discarded.

4.1.5 Coupling Modes

In early systems such as POSTGRES [15], condition evaluation and action execution were done immediately after the event was detected. However, in some situations this is too restrictive. For integrity checks, condition evaluation and action execution need to be done at the end of a transaction before it commits. Coupling modes were introduced [16] to specify a relative point in time where condition evaluation and action execution should take place after the event is detected, with the constraint that the action will be performed only when the condition is satisfied. There are three coupling modes:

- **Immediate**:

  When an event is detected, the transaction is suspended, and the condition associated with the event is evaluated immediately. If the condition evaluates to true, the action is executed. The execution of the triggering transaction is suspended while the condition evaluation and action execution are completed.

- **Deferred**:

  The triggering transaction is continued after an event is detected. Condition evaluation and action execution are done at the end of the triggering transaction before it commits.

- **Detached (or decoupled)**:

Condition evaluation and action execution are done in a separate transaction (or triggered transaction) from the triggering transaction. The detached mode can be classified into two types (totally independent and causally dependent). When two transactions are totally independent, the triggered transaction is executed regardless of whether the triggering transaction commits or aborts. On the other hands, the triggered transaction can commit only after the triggering transaction commits for the causally dependent mode.

4.1.6 Rule Priority

In addition to the parameter context, coupling mode associated with a rule, there is also a priority assigned to each rule. The default priority of a rule is a priority of 1. The priority increases with the increase in the numerical value i.e., 2 has a higher priority than 1, 3 is a higher priority than 2 and so on. Rules of the same priority are executed concurrently and rules of a higher priority are always executed before rules of a lower priority. It is possible that a rule raises events that in turn could fire more rules and so on. This results in a cascaded rule execution. Furthermore, rules can be specified either in the immediate coupling mode or the deferred coupling mode. Both the priority and coupling mode of a rule have to be taken into account for scheduling the rule for execution.

We use the Java LED (Java Local Event Detector) for WebVigiL. LED is a library designed to provide support for primitive and composite events, and rules in Java applications in a *seamless* manner. It is actually an event detector that was implemented to detect events in Java applications and executes rules defined on them. Primitive event detection as well as composite event detection in various parameter contexts and coupling modes has been implemented in LED. Also, the application developer has to explicitly put the raiseBeginEvent and raiseEndEvent calls inside the methods that have been defined as primitive events [17].

4.2 Activation/Deactivation

A sentinel's life span is specified by the start and end time. During its lifespan, a sentinel is active and participates in change detection. A sentinel is enabled (participates in change detection) by default at its start time, and can be disabled explicitly by the user during its lifespan. The start/end time of a sentinel can be a point on the time line or can be an event [7] that references another sentinel's start or end time. When a sentinel's start time is *now*, it is enabled immediately. But in cases where the start is at a later time point or depends on another event that has not occurred, enabling of the sentinel is *deferred* until that time is reached or the event of interest has occurred. To facilitate this we need a triggering mechanism that will raise the event required to enable/disable a sentinel. In WebVigiL, the change detection module generates appropriate events and rules and are instantiated using the LED. The start and end events are implemented as primitive events. Start and end of a sentinel are treated as potential events as they can trigger the start or end of other sentinels. For every sentinel, start and end events are created and rules are associated.

$$\text{Event Start\_}s_i = \text{createEvent(“start\_}s_i\text{”)} \qquad (\mathbf{1})$$

$$\text{Rule Rstart\_}s_i = \text{createRule(Start\_}s_i\text{, condition\_}s_i\text{, action\_}s_i\text{)} \qquad (\mathbf{2})$$

Statement 1 shows the start event creation and statement 2 shows the rule creation for a sentinel s_1. More than one rule can be associated with an event (i.e., Rstart_s_1 .. Rstart_s_n can be associated with event Start_s_1). When the event created in statement 1 is raised the rules associated with it (statement 2) are triggered. When the rule is triggered the action is performed only when the condition is true. With enabling/disabling sentinels there are no conditions to check, thus the event is raised and the respective rule enables the sentinel. An event can be raised in the following ways.

- Absolute Time: Consider the scenario where s_1 is defined in the interval [06/02/03, 07/02/03]. At time 06/02/03 the start event associated with sentinel s_1 has to be raised for it to get enabled. Following are the events and rules that are generated to enable sentinel s_1:

  Event Start_s_1 = createEvent("start_s_1")

  Rule Rstart_s_1 = createRule(Start_s_1, condition_s_1, action_s_1)

  Event ETime$_1$ = createTemporalEvent (06/02/03)

  Rule RTime$_1$ = createRule (Etime1, condition, action)

  When event ETime$_1$ is raised at the specified time point, rule RTime$_1$ is triggered. The action associated with rule Rtime$_1$ in turn raises the event Start_s_1. When Start_s_1 is invoked the action (action_s_1) associated with rule (Rstart_s_1) enables sentinel s_1.

- Relative: The start/end of a sentinel may depend on the start/end of other sentinels. Sentinel s_2 should be enabled, if it is defined over the interval [start (s_1), end (s_1)], when s_1 is enabled. Another rule that raises the event Start_s_2 is associated with event Start_s_1.

The following are the events and rules that are generated in order to enable s_2 when s_1 is enabled.

Event Start_s_2 = createEvent("start_s_2")

Rule Rstart_s_2 = createRule(Start_s_2, condition_s_2, action_s_2)

Rule Relative_Start_s_2 = createRule (Start_s_1, condition, action)

When a sentinel s_3 specifies its time interval as [start (s_2)+ 1 day, end (s_2)+ 1 week], s_3 should be enabled after 1 day since s_2 is enabled. Composite event PLUS is used to achieve the above. The events and rules generated in order to enable s_3 are as follows.

Event Start_s_3 = createEvent("start_s_3")

Rule Rstart_s_3 = createRule(Start_s_3, condition_s_3, action_s_3)

Event S_2Plus_1day = createPlusEvent (Start_s_2, 1 day)

Rule R_Start_s_3 = createRule (S_2Plus_1day, condition, action)

The event Start_s_3 is raised by the rule R_Start_s_3 when the plus event S_2Plus_1day is raised. And finally sentinel s_3 is enabled. Each sentinel generates a start and an end event. Rules are associated with these events to trigger the start or end of other sentinels.

All the above examples are based on sentinels that are relative to start of other sentinels. A sentinel's start can also depend on the end status of other sentinels and vice versa. Generalized event and rule definition is as follows.

Event e_i = createEvent("event_e_i") **(3)**

Rule R_e_i = createRule(e_i, condition_e_i, action_e_i) **(4)**

In the above statements e_i can be the relative event (start/end) or absolute time based event (*t*) or a plus event (e_i + *t*).

4.3 Fetching

In order to monitor the page targeted by the sentinel, it has to be fetched using the specified periodicity. In WebVigiL, we use the PERIODIC event to achieve this fetch in an asynchronous manner. A periodic event is an event that repeats itself within a constant and finite amount of time. The initiator and terminator are the start and end events of a sentinel and *t* is the interval with which the page should be monitored. The actual fetch of the page is performed by the rule associated with the periodic event. The sentinels in WebVigiL can be classified into two categories.

- Fixed fetch-Interval: In this case, the interval for polling is explicitly specified by the user. For example, the user knows that the page changes several times a day but he is interested in changes happening each hour. In WebVigiL for every sentinel belonging to this category, a unique periodic event with an associated fetch rule is generated. For sentinel s_1 whose periodicity is defined as 2 days, the periodic event generated is as follows.

  Event FetchEvent_s_1 = createPeriodicEvent (Start_s_1, 2 days, End_s_1)

  Rule FetchRule_s_1 = createRule (FetchEvent_s_1, condition, action)

  Figure 4.1 shows the graphical interpretation of the periodic event FetchEvent_s_1.

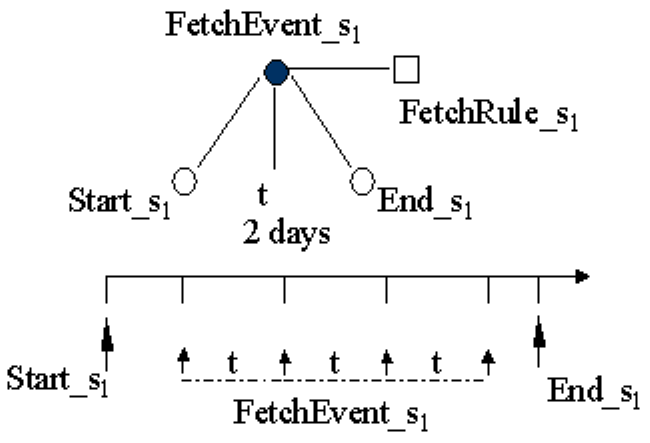


Figure 4.1: Periodic Event

- On-Change: This specification requests the WebVigiL system to detect the changes as soon as it takes place. In order to do so, the system has to fetch the page at some pre-determined frequency and learn to fetch the page as its modification history is collected. The initial interval of polling the page is set by the system to a predefined value. This is useful in scenarios where the user is interested in every change occurring to the page but has no clue when the page changes. Currently the system initially starts with a small interval (1 hour) and learns from the previous change intervals [9]. All sentinels monitoring the same page belonging to this category *share* a common fetch rule. Consider the scenario where sentinel s_2 is defined in the interval [06/02/03, 07/02/03] on $page_i$ and sentinel s3 defined in the interval [06/01/03, 07/01/03] on the same $page_i$ with on-change specification. Initially when s_2 registers with the system the periodic event generated is

  Event $FetchEvent\_Page_i$ = createPeriodicEvent ($Start\_s_2$, *t*, End_s_2)

Rule FetchRule_Page$_i$ = createRule (FetchEvent_Page$_i$, condition, action)

When sentinel s_3 registers with the system, as both s_2 and s_3 are requesting monitoring of the same page and fall under the same category, the previous fetch rule (FetchRule_Page$_i$) generated is shared. Since s_3 starts at an earlier time than s_2, the polling should initiate when event Start_s_3 is raised, and terminate when End_s_2 is raised (end time of s2 is later than end time of s3). Thus the earlier initiator and terminator of the periodic event generated should be replaced (FetchEvent_Page$_i$) with Start_s3 and End_s2 respectively. The periodic event reflecting the changes is

Event FetchEvent_Page$_i$ = createPeriodicEvent (Start_s_3, *t*, End_s_2)

If all the succeeding sentinels registering with the system belong to the same category of s2 and s3, the periodic event initiators and terminators have to be determined at runtime and if needed, be replaced at runtime. Furthermore, the user can also explicitly disable a sentinel before it has started which results in computing the initiators and terminators. In order to avoid this computation each time a sentinel belonging to the same category is registered, dummy initiator and dummy terminator are created. The periodic event generated would be

Event FetchEvent_Page$_i$ = createPeriodicEvent (Start_dummy, *t*, End_dummy)

When the start of a sentinel is raised the rule associated with it checks on the status of the periodic event (i.e., initialized). If initialized it does not raise the Start_dummy event. Similarly when the end of the sentinel is raised it checks whether the Fetch rule is servicing other sentinels. Based on this information the End_dummy event is raised.

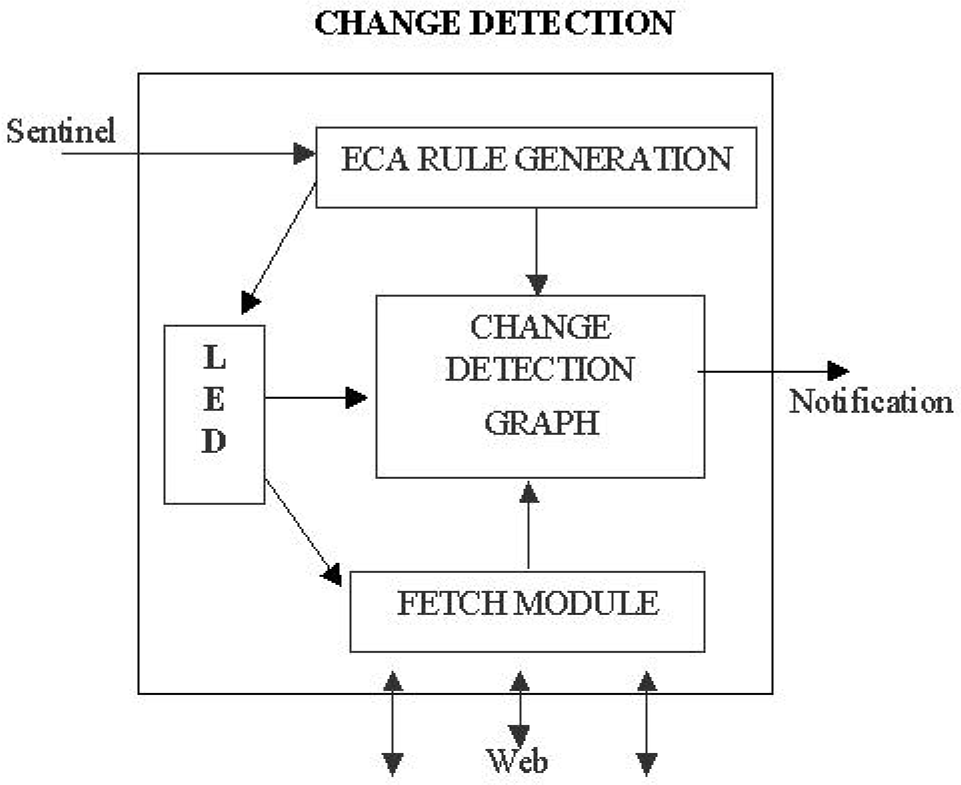


Figure 4.2: Change Detection Module

Since the initiator and terminator are raised only once, the contexts (section 4.1.4) do not apply. Hence, the default RECENT context is used. The rules associated with all the events (absolute, relative, plus and periodic) generated are executed in the immediate coupling mode. Currently, the priority of all the rules is assumed to be the same. In this manner, ECA rules are used to asynchronously activate (enable) and deactivate (disable) 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 the next chapter. The following Figure 4.2 shows the individual modules in the Change Detection module. The sentinel is received as input to the ECA Rule generation that creates the rules necessary for monitoring using the LED. The rules, upon firing, inform the change detection graph for enabling/disabling the sentinels. The fetch module fetches the pages (Fetch Rules) and propagates them to the change detection graph for participation in change computation and detection.

CHAPTER 5

CHANGE DETECTION GRAPH

5.1 Introduction

For each document/page (HTML/XML) of interest, when the page is fetched the change is detected and reported to all the sentinels interested in that change. Change is detected between the current page and the previously cached page for the same URL. Change detection algorithms CH-DIFF and CX-DIFF [7, 9, 18] have been developed to support the change types (links, images, keywords, phrases, all-words) for HTML and XML pages, respectively. The framework for monitoring is based upon the use of events. Fetching of a page is considered an event that starts the process of change detection. Each type of detected change is considered an event that is propagated to detect composite events. The WebVigiL system detects these events for each document (page) on which a sentinel is set. The system should also be able to detect composite events. A composite event is an event expression comprising a set of events related through one or more event operators such as NOT, AND, OR [6].

As we are assuming a large number of users setting sentinels on URL's for different types of changes, we are likely to have overlaps among URL's, types of changes, frequency of access etc. One of the goals of WebVigiL is to process sentinels efficiently and be able to scale to a very large number of sentinels. Version control is also an important issue, as versions of a page should be maintained (depending upon the window specification). This thesis addresses the efficient evaluation (i.e., change detection) of sentinels. Some of the issues that need to be addressed are

- Mapping of URL's to an internal representation
- Raising an event when a page is fetched
- Detecting changes for ALL sentinels (with different types of changes) efficiently
- Detecting composite events
- Passing changes detected to the notification module

The following sections discuss some of the approaches for handling the above issues.

5.2 Naïve Approach

One approach for change detection is to maintain a hash table with the page of interest as the key and the value being a list of sentinels monitoring that particular page. When the page is fetched, the sentinel(s) on that page are extracted and change type is detected for *each* sentinel. For example, for sentinels S_1 and S_2 that are interested in links change to $Page_i$ the mapping constructed is shown in Figure 5.1. When $Page_i$ is fetched the sentinels S_1 and S_2 are extracted. For each of the sentinels the previous version of the page is fetched from the cache and the change detection algorithm for links is executed and is notified if there is a change.

This approach is rather naïve since, when there are sentinels interested on same change type on the same page, the change is computed twice. In the above example, if S_1 and S_2 are monitoring links change to "www.uta.edu"($Page_i$), then change detection is computed twice for each sentinel. In order to attain scalability more efficient approach for change detection is required.

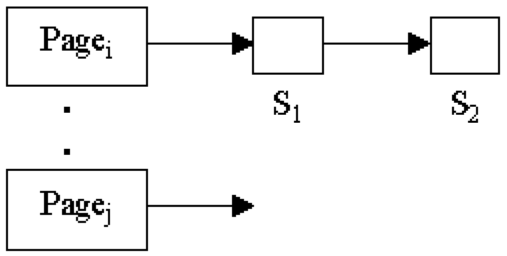


Figure 5.1 : Naïve Approach

5.3 Hash-Based Approach

In order to avoid the redundant computation when two or more sentinels subscribe to the same properties (same target page and change type) a grouping can be established. Multiple sentinels monitoring the same change type on the same page can be grouped together. In this approach a hash table is maintained with the monitoring page as the hash key. Each bucket of the hash table contains a list of groups, one group (change types) per page. Sentinels that share the same target page are hashed into the same bucket. Consider sentinels S_3 and S_4 that are interested in images change to "www.uta.edu" ($page_i$) in addition to the other sentinels discussed in the previous approach. Figure 5.2 shows the mapping structure constructed for all the sentinels.

When $Page_i$ is fetched, all the groups corresponding to the page are retrieved and the corresponding changes are detected based on the group property (change type). Hence for sentinels S_1, S_2 the change is computed once. Similarly for S_3, S_4 the change is computed only once. With this approach the redundant computation is avoided. A composite change refers to a "combination" of changes via operators. For example, changes to links AND images is a composite change, where AND is the operator. This change is detected only when both links *and* images change on the page. But, in order to achieve composite change detection on the same page or multiple pages, this approach induces lot of complexity. Either hashing or grouping can be used for composite change detection. Sentinels from different buckets need to be interlinked for composite change detection. Deletion of sentinels with composite changes will also pose problems, as the information needs to be propagated from the root sentinel node.

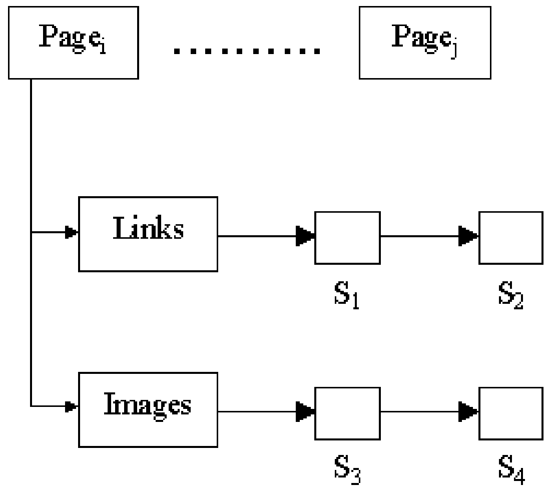


Figure 5.2 : Hash-Based Approach

5.4 Change Detection Graph

We need a data structure that will allow us to asynchronously feed fetched pages for change detection, allow parallelism where possible, optimize the computation by grouping sentinels over URL's and change types, and facilitate composite change detection using the same paradigm as primitive change detection. Deletion and propagation of delete semantics must be straightforward in the representation chosen. Although a number of data structures have been proposed in the literature for event detection, such as Petri nets [19], extended automata [20], it has been shown that event graphs [14, 21] support the requirements at the granularity and grouping that is appropriate for our problem. Hence, we have adapted and extended the event graph approach proposed for snoop [8] for detecting primitive as well as composite changes. Below, we describe the extended structure along with its advantages.

Primitive change detection involves detecting changes to links, images, keywords etc., in a page. In order to facilitate primitive change detection, grouping of sentinels, and data flow we construct a graph. This graph is referred to as the change detection graph (CDG). The graph is constructed bottom up as shown in Figure 5.3. The different types of nodes in the graph are as follows:

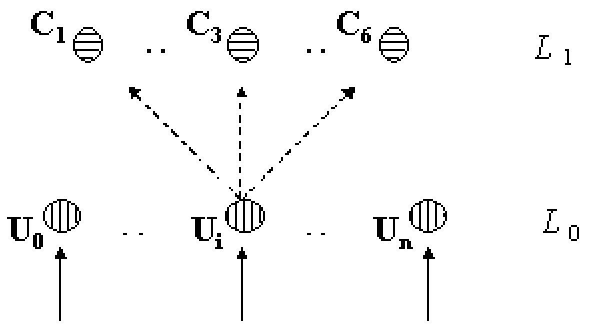


Figure 5.3: Change Detection Graph

- URL node (U_n): A URL node is a leaf node at level-0 (L_0) that denotes the *page of interest* (e.g., “www.uta.edu”). The number of URL nodes in the graph is equal to the number of *distinct* pages the system is monitoring at that particular instant of time. At this level whenever the version of a page is fetched (treated as fetch event), it is propagated to respective nodes at level-1.

- Change type node (C_n): All level-1 (L_1) nodes in the graph are change type nodes. This node represents the *type of change* on a page (links, images, keywords, phrases etc., see Table 5-1). Change detection of pages is performed at this level. The maximum number of change type nodes that are created in the system is equal to the product of the number of change types supported and the number of URL nodes present at that time instant. Currently the number of change types is equal to 6. Each URL node can be connected to at most 6 change type nodes.

Table 5-1. Change Types Supported

Change Type	**Description**
Links	All the Links in the URL given
Images	All the images in the given URL
Phrases	Changes to a particular phrase(s)
Keywords	Changes to a particular keyword(s)
Any-change	Any change in the given URL
All-words	Changes to all the words excluding user defined set of words in the given URL

In the graph, to facilitate the 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. The subscriber list at each node contains the following

- Level-0: Contains references of level-1 nodes.
- Level-1: Contains references of sentinels monitoring that change.

The change is computed for all the sentinels present in the subscriber list at the change type node (C_n). There can be more than one sentinel associated with each change type node. The arrows in the graph represent the data flow. For example, consider two sentinels, S_1 monitoring changes to links and S_3 monitoring changes to images on $page_i$ as shown in Figure 5.4. The node references are maintained at the URL node ($page_i$). When the new version of the $page_i$ is fetched, it is propagated to the links and images node. At each change type node, the previous version is retrieved from the version controller and the appropriate (links, images) change is computed. If there is change, the sentinels subscribed to it are notified, in this case it is notified to S_1 and S_3.

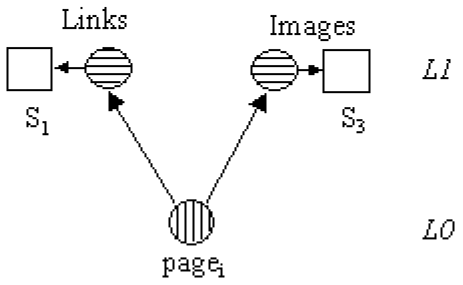


Figure 5.4: Primitive Change Example

When a sentinel reaches its end time or is explicitly disabled by the user, the sentinel no longer participates in change detection. This information is propagated to the change type node with which the sentinel is associated. Since the change type node is a subscriber to the URL node, it decides on whether to remove its subscription based on the other sentinels that are associated with it. Once this information is propagated to the URL node it removes the references of the corresponding change type node and does not send the next version of the page for change computation. For example, if sentinel S_1 shown in Figure 5.4 is disabled, the next version of the page is not propagated to the links node from URL node ($page_i$), since there are no other sentinel that are interested in links change. Graph structure suites well in these kinds of applications where flow of data is required.

5.4.1 Sentinel Grouping

System scalability and performance issues arise when there are large numbers of sentinels registered with WebVigiL. Typically, the system should not impose a restriction on the number of sentinels it can handle. This introduces the need for a technique to minimize the susceptibility of the system to attain efficiency in change computation and to guarantee the quality of service. Consider the scenario where there are *n* sentinels registered to monitor the changes to the same page. Instead of computing the change *n* times, it could be reduced to once if we could group all those sentinels together. Grouping is required not only for scalability of the system but also to reduce the I/O involved in fetching the old version of the page and high computation involved in parsing the documents and extracting information. Pages with huge content add additional burden. Sentinel grouping is an optimization technique developed to minimize change computation. Grouping of sentinels are based on change type, fetch type and compare-options. All these types of grouping are explained in the following paragraphs.

Sentinels are grouped when there is more than one sentinel interested in the same change type. For example, more than one student might be interested in knowing if there are any new questions added (links) to the course message board. In this case, all the students' sentinels are grouped together for change detection. As another example, on a university sports page where forthcoming events are listed, students might be interested in knowing if their sport of interest (keywords) appears or when there is change to the paragraph (phrase) containing venue and time of a particular game. Another scenario is where users are interested in any change to a page but each user is interested in a different set of words that should not be included in the change detection such as articles (a, an, the). Based on the semantics of the change type the corresponding nodes maintain either a set containing the union of all the words (keywords/phrases) or intersection of all the words (all-words). For example, sentinel s6 is interested in keywords {x, y} and s7 on keywords {y, z} on the same page. The change type node maintains the union of all the individual sentinel's keywords {x,y,z}.

A sentinel belongs to two categories depending on the fetch type (section 3.1). Only those sentinels belonging to on-change fetch type are considered for grouping, as the change is computed on the versions of page fetched by the common fetch rule. Sentinels belonging to fixed-interval are not grouped as each has its own version of page being fetched by their respective fetch rules. In spite of sentinel belonging to on-change the other attribute that plays a role in the grouping strategy is the compare-options (pair-wise, moving:n, every:n see Table 5-2)(section 3.1). For example, a page is updated thrice a day and a user is interested on every consecutive change (Pair-wise) whereas another user is interested in changes only once a day (every:3). Here the versions of pages involved for change computation are different and hence the sentinels cannot be grouped together. Furthermore, sentinel's belonging to the same compare-option need not belong to the same group and hence cannot be grouped together.

Table 5-2. Compare-options supported

Compare-options	**Descriptions**	**Point of detection**
Pair-wise	Detects changes to consecutive versions of the same page	Once the subsequent version arrives
Moving:n	Detects changes to versions "i" and (i-n+1) (where "i" is the version of the page fetched and "n" is the moving window size) of the same page	Once the n^{th} version arrives
Every:n	Detects changes to every "n" versions of the same page	Every change is detected after waiting for "n" versions. These versions are termed as *waiting versions*.

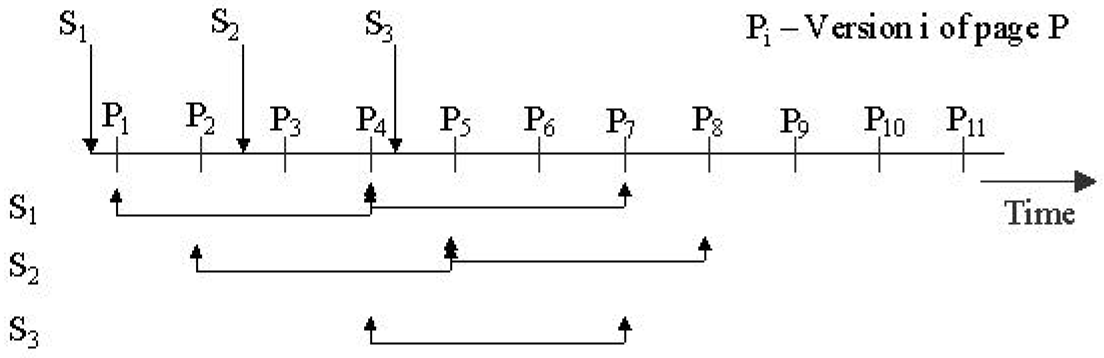


Figure 5.5: Compare-Options with Every: 4.

Consider the following example where S_1 and S_2 are on-change sentinels monitoring the same change on the same page (P) with the compare-option of "every:4" as shown in Figure 5.5. The points on the time line denote either the arrival of sentinels or the versions of page P. For S_1, change is detected between the versions (p_1, p_4) and so on. When S_2 arrives, since there is already a cached version p_2 the change is detected between (p_2, p_5) and so on. For S_3 the change is computed with (p_4, p_7). Hence S_1 and S_2 cannot be grouped together whereas S_1 and S_3 can be grouped. This behavior is applicable only to sentinels with *every* as the compare-option since only selected versions of the page participate in change detection. For S_1, p2 and p3 are not used even though fetched whereas this is not the case for other compare-options (pair-wise and moving). If there is a sentinel S_4 on "moving:4" and has arrived along with S_1, p_1 and p_4 participate in change initially, but later on (<p2, p5>, <p3, p6>…) all versions fetched will participate in change detection. Hence any other sentinel S_i interested on "moving:4" arriving at a later time will be grouped along with S_4. The same applies to sentinels having pair-wise as their compare-option.

Thus the sentinels are grouped on a combination of change type, fetch type (on-change), compare-options. When the compare-option is "every:n" then the corresponding time at which the sentinel arrives is taken into consideration.

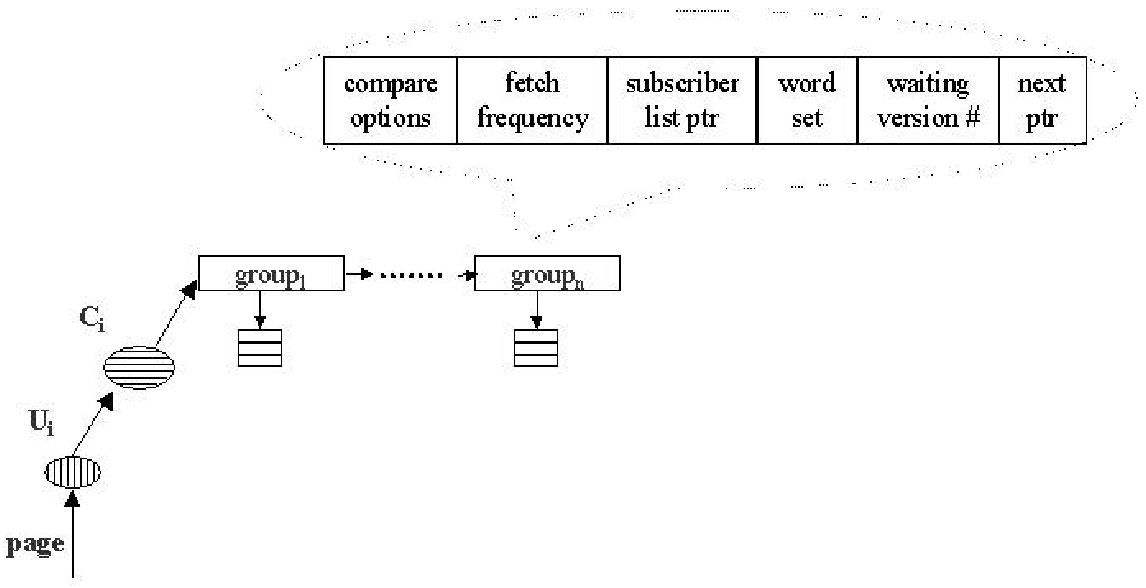


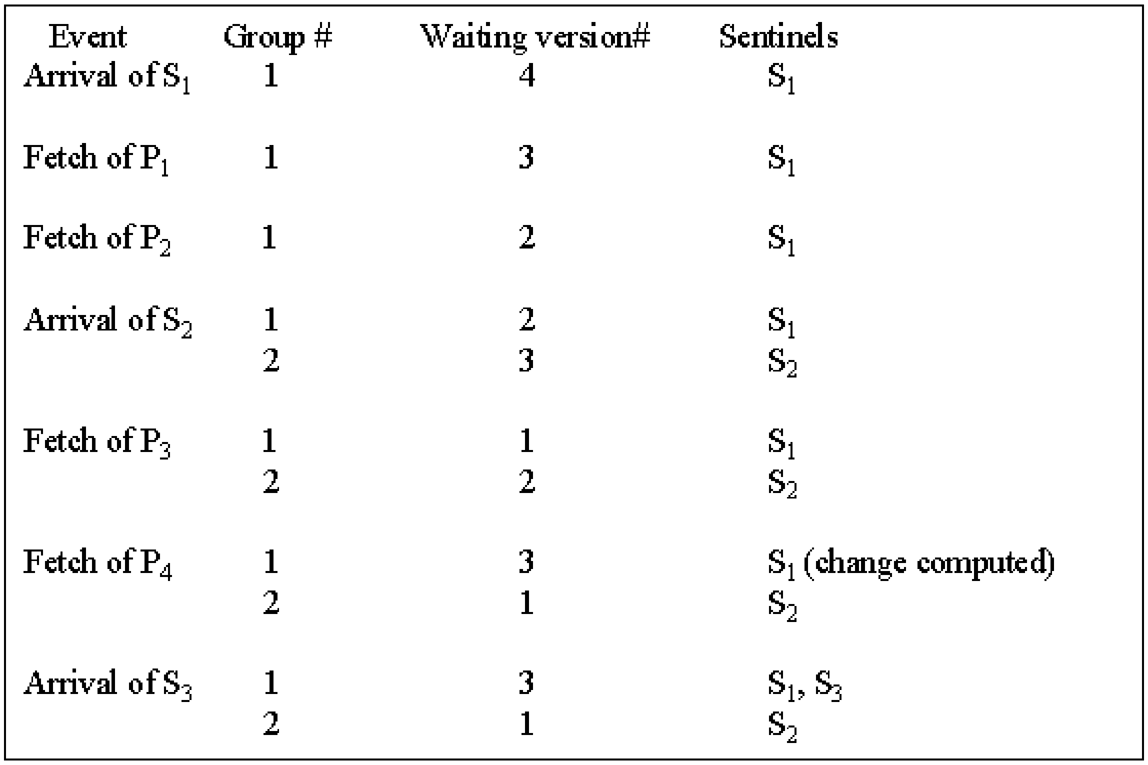
Figure 5.6: Grouping Data Structure

Figure 5.6 shows the information used for grouping sentinels based on the strategy explained above. The "subscriber list ptr" contains all sentinels that belong to the same group. The "word set" attribute is null for *links, images and any-change* or union for *keywords and phrase* or intersection for *all-words* depending on the change type. The number of waiting versions is used to differentiate sentinels with the same compare-option "*every:n*" and also to determine when to compute the change. Figure 5.7 shows the algorithm used to group sentinels and Figure 5.8 shows the various stages of the data structure based on the time line showed in Figure 5.5 as the algorithm is applied. The following algorithm is used to determine the group to which the sentinel belongs.

```
Grouping (Sentinel s, Compare-option option, Reference n)
/* Inputs: Sentinel s, Compare-option Every, Reference n */
    1.  Get all the groups with the compare options equal to option and reference equal to n
    2.  if no group exists go to step 12
    3.  switch (option)
    4.      case Pair-wise: go to step 13
    5.      case Moving: go to step 13
    6.      case Every:
    7.          for each group
    8.              get waitingversions = number of versions the group is waiting for
    9.              if there is a version of the page in the cache
    10.                 if n-1 = waitingversions go to step 13
    11.             else if n = waitingversions go to step 13
    12. Create a new group
    13. Add the s to the group
    14. End
```

Figure 5.7 : Grouping Algorithm

As shown in Figure 5.8 when sentinel S_1 arrives the number of waiting versions is set to 4 as it has to wait for four versions (no cached version) in order to compute the change. As each version of the page is fetched the number of waiting versions is decremented. When S_2 arrives it has to wait for another three versions (cached copy is available) for change computation. Hence it is not grouped with S_1. As the version P_4 is fetched the number of waiting versions for S_1 equals to zero and the change is computed. After change computation the value is set to three for the next computation. And finally when S_3 arrives since S_1 is also waiting for the same number of versions, they are grouped together.



Event	Group #	Waiting version#	Sentinels
Arrival of S_1	1	4	S_1
Fetch of P_1	1	3	S_1
Fetch of P_2	1	2	S_1
Arrival of S_2	1	2	S_1
	2	3	S_2
Fetch of P_3	1	1	S_1
	2	2	S_2
Fetch of P_4	1	3	S_1 (change computed)
	2	1	S_2
Arrival of S_3	1	3	S_1, S_3
	2	1	S_2

Figure 5.8 : Grouping Example

So far primitive change detection has been discussed. In the following paragraphs composite change detection will be discussed. A composite event is an event expression comprising a set of events connected through one or more composite event operators such as NOT, AND, OR (refer Table 5-3).

Table 5-3: Semantics of Composite Event

Operator	Semantics
NOT (unary)	Non-occurrence of a given event, ($\neg C_1$) where C_1 is the constituent event
OR	Changes to either of the constituent events, ($C_1 \vee C_2$) where C1 and C2 are constituent events.
AND	Changes to both the constituent events on the same versions of the page, ($C_1 \wedge C_2$) where C_1 and C_2 are constituent events.

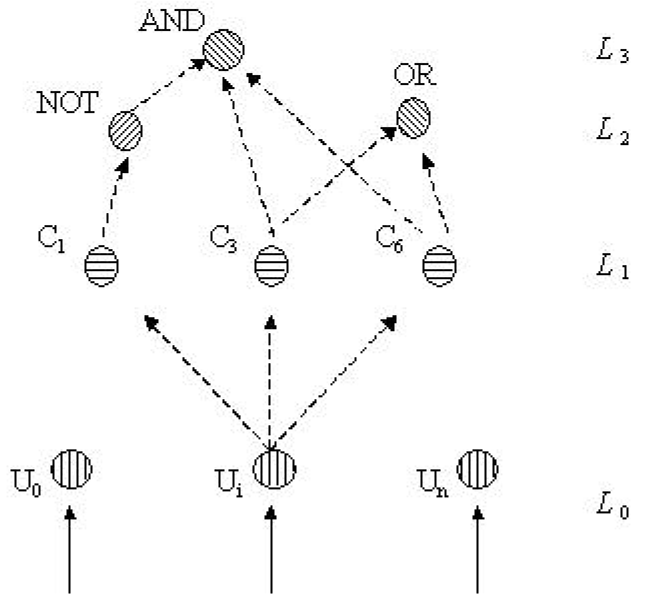


Figure 5.9. Composite Change Detection Graph

As shown in Figure 5.9 composite event nodes are in the levels L_2 and above. Composite Node represents a combination of change types through the operators NOT, AND, and OR. They can extend to any number of levels. These nodes are created for every sentinel monitoring a composite event. Level-2 and above contains references of the nodes belonging to the immediate higher level (composite event containing more than two constituents) or sentinels.

The change is computed for all the sentinels present in the subscriber list at the change type node. Hence, for sentinels monitoring composite changes, a representation at its constituent change type node is needed. This is implemented by creating proxy sentinels with the same properties of the original sentinel at each of the constituent change type node. Consider the scenario where sentinel S_5 is interested in links and images change to "$page_i$" (refer Figure 5.10). When the new version of the "$page_i$" is fetched it is propagated to the links and images node. If there is a change, sentinels subscribed to it are notified. Sentinel S_{and} acts as a proxy for S_5. When S_{and} is notified the change computed is in turn propagated to the AND node. At the AND node, S_5 is informed only when it receives notifications from both its constituent S_{and} sentinels.

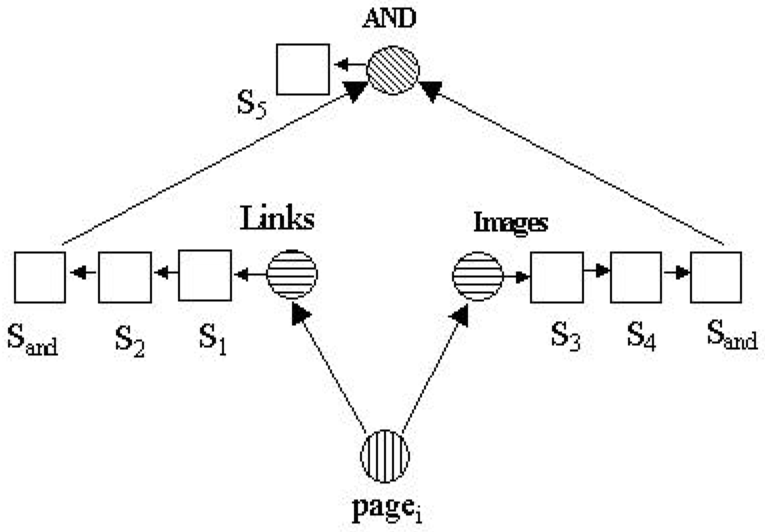


Figure 5.10: Composite Example

Following are the steps taken when a new sentinel is registered with the system:

1). The URL node corresponding to the target page of sentinel is created if there is none.

2). The change type node associated with the target change is obtained; if there is none, a new node is created.

3). The grouping structure is traversed to obtain the group to which the new sentinel belongs. If there is no such group a new group is created and the sentinel is appended to the list of sentinels pointed ("subscriber list ptr") by the grouping structure.

If the sentinel is any-change, links or images, the sentinel is added to the subscriber list of the group. If the sentinel is keywords or phrases, a union is performed between the keywords/phrases of the new sentinel and words/phrases belonging to the word set of the group. If the sentinel is all-words an intersection is performed on the word set and the words the sentinel is not interested in.

As versions of a page are fetched and are propagated through the change detection graph, the following steps are performed:

1) The event fetchedPage (URL) is raised at the node corresponding to that URL. The event is further propagated to all the subscribers at that URL node.
2) At the change type node, for each group the corresponding old page (based on the compare-option) is retrieved from the version controller and change is computed with the current page. If there is a change detected, all the subscribers in that group are notified. When there is no change then only those sentinels interested on NOT change are notified.

   In case the change type is keywords or phrases, the change is computed for all the words/phrases present in the word set against the two pages. If a change is detected then for each sentinel we check if his or her words/phrases of interest is present in the change computed and notify them accordingly.

For a composite event AND the sentinel is notified only after notifications from both the constituent events on the same set of versions are received. In case of OR it is notified when notifications from either of the constituent events is received. The notifications contain the change and the set of versions on which the change is detected. NOT is detected when the change in the notification is null.

5.4.2 Illustration of Composite Change Detection

Consider the following sentinels monitoring composite changes on page p with the same compare-option and fetch frequency.

S_1 - C_1 OR C_2

S_2 - C_1 AND C_2

S_3 - (C_1 OR C_2) AND C_3

Where C_1, C_2, C_3 represent the events (changes to links, images, keywords, phrases, all-words, any-change). The time line showing the occurrence of events and the detection process is illustrated in the following Figure 5.11. On the time line, $C_k^{i\ (x,y)}$ represents the change detection event C_k on versions x and y of page p and i denotes the ith occurrence of the event C_k. The event graph shows the nodes from the level-1 (does not include the URL node representing page p). Figure 5.11 shows the composite event detection process for each sentinel. The time line indicates the relative order of primitive events with respect to their time of occurrences. All event propagations are done in a bottom-up fashion. The leaves of the graph have no storage and hence pass the primitive events directly to the parent nodes.

The various operations at the time points where primitive events are detected are shown in Figure 5.11. The arrows pointing from the child node to its parent in the graph indicate the detection and flow of events. The OR event is straightforward and is detected whenever any of its constituent events are raised. Hence S_1 is notified at t_1, t_2 and t_3. At time t_1 when $C_1^{1\ (1,2)}$ is propagated to the AND node, it waits for the event to be raised from the other constituent event on the same versions (1,2) for S_3 to be notified. At time

t_2 it receives $C_1^{2\ (2,3)}$ that replaces $C_1^{1\ (1,2)}$. Currently the composite operators can be specified on the same page and not across different pages. So when a version "i" is

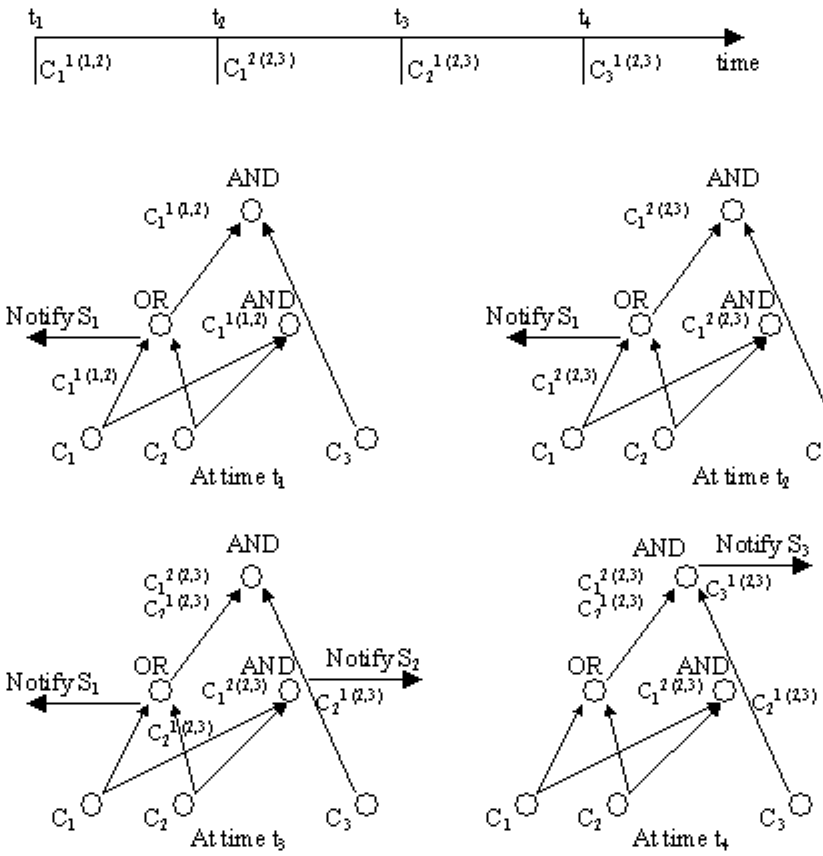


Figure 5.11: Composite Change Detection Example

fetched, all the monitored changes are computed on versions "i-1" and "i" (for pair-wise change detection). If there is a change, the corresponding primitive events are raised. Hence when the next change is computed between versions "i" and "i+1" and the composite node does not get any notification from its other constituent on versions "i" and "i-1", it means there is no change detected between "i" and "i-1". Hence all old event occurrences are replaced by the new occurrences. This is illustrated in the Figure 5.11 at time t_2. S_2 is notified at time t_3 when it receives $C_1^{2\ (2,3)}$ and $C_2^{1\ (2,3)}$. At time t_3 the AND node corresponding to S_3 saves both the versions $C_1^{2\ (2,3)}$ and $C_2^{1\ (2,3)}$ and when $C_3^{1\ (2,3)}$ is received S_3 is notified twice for the combinations ($C_1^{2\ (2,3)}$, $C_3^{1\ (2,3)}$) and ($C_2^{1\ (2,3)}$, $C_3^{1\ (2,3)}$).

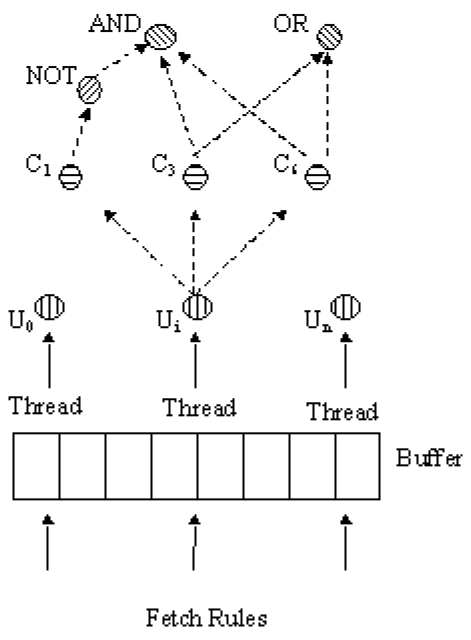


Figure 5.12. Parallelism in CDG

5.5 Parallelizing Change Detection

The Change Detection Graph (CDG) is designed to detect changes to pages that are fetched by the fetch rules. When there are large numbers of sentinels registered, for the system to be scalable, change detection for sentinels should be handled simultaneously. This can be achieved through parallel detection of changes to different URL's at the same time. As shown in Figure 5.12 the fetch rules place the versions of the pages into the buffer. As each URL is assigned a separate URL node and the change detection of different URL's do not interfere with each other. Hence change detection of different URL's can be processed concurrently. Here parallelism is achieved only between different URL's (inter-URL). Currently WebVigiL supports only inter-URL parallelism.

The implementation details are discussed in CHAPTER 7. As discussed so far, graph architecture used for change detection facilitates scalability, efficiency and parallelism.

CHAPTER 6

STORAGE AND RETRIEVAL OF PAGES

This chapter discusses different approaches for storage and retrieval of pages that are being fetched for change detection.

6.1 Introduction

An important feature of WebVigiL architecture is its demand-driven page fetching using its server-based repository service that archives and manages versions of pages. WebVigiL stores only those pages that are needed by a sentinel in the cache. 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, which in turn reduces the communication cost. All the pages are fetched from the web server based on the page properties. For static pages last modified time (LMT) is retrieved before the page is fetched. For dynamic pages, as the LMT is not available the page is fetched and checksum is computed. When a page fetch is initiated, the repository service checks for the existence of the latest version of the page in its cache and if present, the latest version of the page in the cache is returned. In the case of static pages, if the LMT of the page is equal to the LMT of the cached copy it is not fetched. In the case of dynamic page, the page is fetched and the checksum of old version in the cache is compared with the checksum of fetched page. If the checksums are equal then the page is not stored in the cache to avoid duplication of the versions. In case of a cache miss, the repository service requests that the page be fetched from the appropriate web server. In order to detect changes to a particular page, versions of that page have to be stored in the cache. To maintain these versions in the cache, each URL has to be mapped to a unique directory. The complete URL cannot be used as a directory name since the length of the URL is very large in many cases. Two approaches to establish the required mapping for the directory structure is discussed below.

6.2 Hash-based Approach

In this approach, each unique URL is inserted into a hash table with URL as the key. Mapping is generated for each key (i.e., unique URL). This mapping represents the directory where the corresponding versions of the unique URL are to be stored. Consider two URL's “x/y/z/i.htm” and “x/y/z/j.htm”, which have common path “x/y/z”. Instead of generating the mapping for each of these URL’s, a mapping is generated for the common path once and is reused whenever required. In the above example, when a version of the first URL (“x/y/z/i.htm”) is fetched, it is saved in the directory U1F1, where U1 is the mapping generated for “x/y/z” and F1 is the mapping generated for “i.htm”. U1 and F1 are maintained as values with the corresponding URL path as the key in the hash table so that they can be reused. When a version of second URL (“x/y/z/j.htm”) is fetched it is stored in the directory U1F2, where the mapping for “x/y/z” is got from the hash table (i.e., U1) and F2 is the mapping generated for “j.htm”. All versions pertaining to these URL’s are stored under U1F1 and U1F2 respectively.

6.3 Directory-based approach

In this approach the path of the URL is replicated for the directory structure. For example, for the URL “x/y/z/i.htm”, the directory structure can be “x\y\z\i.htm”. But, in case of dynamic pages the filename (i.e., “i.htm”) can be very large. As the underlying operating system imposes a restriction on the length of the directory name, we cannot create directories for dynamic pages. Hence we use hash-based approach to generate a unique mapping for the file name. Thus, the directory structure is a concatenation of URL up to the filename (i.e., “x/y/z/”) with the mapping generated. For the above example the file “i.htm” is mapped to F1 and all versions of the page are stored under the directory “x\y\z\F1”.

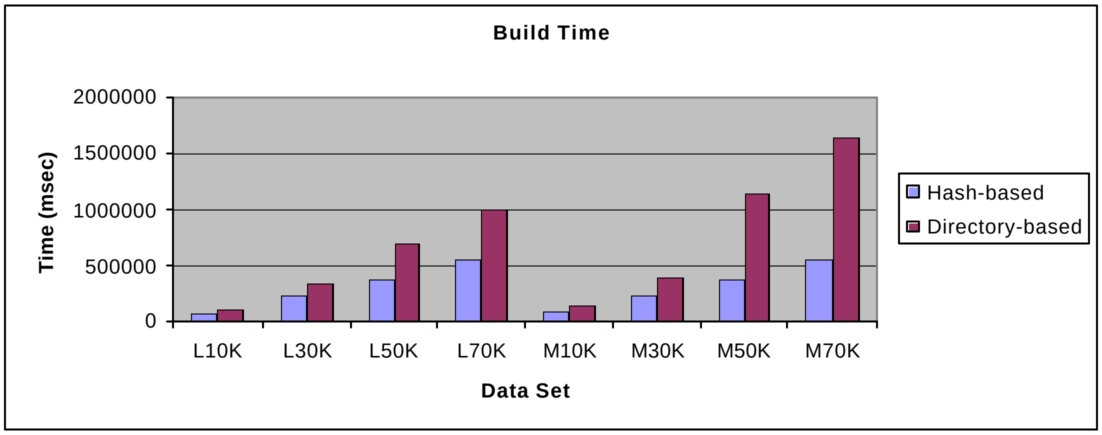


Figure 6.1: Build Time Analysis

6.4 Experimental evaluation

Two approaches explained above were evaluated based on two criteria’s i) time taken to construct the mapping, and ii) time taken to reconstruct the mapping (in case of recovery) plus time taken to retrieve the page given its URL. In these evaluations filename in the URL is not considered, since both the approaches uses the same hash based functions provided in Java 1.4 for generating the mapping. Thus selection of data sets for the experiments were based on the length of the path (excluding the file name) of the URL denoted as depth. For example, for the URL “x/y/z/i.htm” depth is 3. The data sets are represented as “L#” where “#” denotes the number of URL's with depth varying between 1 and 3 and “M#” with depth varying between 4 and 6. For example, L10K, represents 10K URL’s with depth range 1 to 3. Based on an experiment where 30,000

URL’s were extracted from the web, the maximum depth was limited to 6. The experiments were run on a single processor Intel Pentium (700Mhz) machine with 256MB RAM, loaded with Windows 2000 operating system.

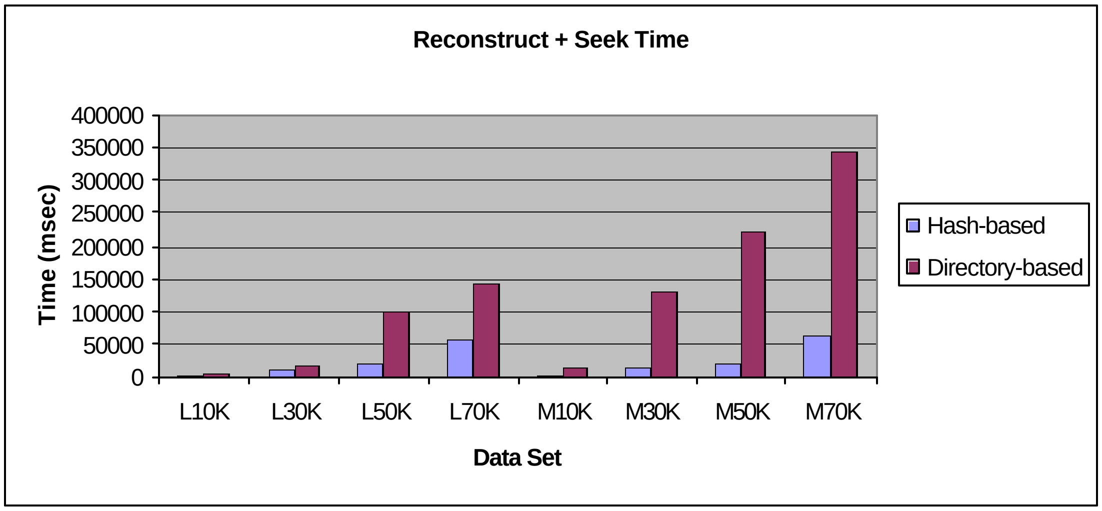


Figure 6.2: Reconstruction + Seek Time

Figure 6.1 shows the time taken to build the mapping for each data set. As it is shown in the Figure 6.1 for L/M10K, L/M30K directory based approach takes more time than hash-based approach, but for M50K, M70K it is nearly three times. The directory-based approach takes more time as more I/O is involved. The complete directory representing the URL was created using a single *mkdir* command. Figure 6.2 shows the time taken to rebuild the mapping plus time taken to retrieve the page given its URL. Mapping is persisted as and when unique URL's are hashed (during the build time). During reconstruction, unique mappings for persisted URL's are not regenerated, thus saving time. From Figure 6.2 it is observed that, as the depth increases, more number of directories has to be traversed (one *cd* command was issued for each directory), thus increasing the time for directory-based approach even though there is no reconstruction required. Based on the performance results, hashed-based approach was selected for caching the pages. In the hash-based approach, some of the different hash functions available in the literature [22] have been tested against the hash function provided by Java 1.4. It has been observed that the URL's are more uniformly distributed when hashed using the java hash function. The variation in the number of entries in each bucket was not large when java hash is used. The pseudo code for the hash functions is shown in Appendix A.

CHAPTER 7

IMPLEMENTATION

This chapter discusses the implementation details of the ECA Rule generation module, Change detection and issues involved in parallelizing the computation over the change detection graph and the whole system WebVigiL. In addition, synchronization issues and the selection criteria for synchronization primitives are also discussed.

7.1 Implementation of ECA Rule Generation

For every user request (sentinel) registered with the system a sentinel object is instantiated. The sentinel object captures all the properties of the sentinel such as, status (enabled/disabled), change type (primitive/composite), fetch type (on-change/fixed-interval), compare-options, and etc. All the sentinels are stored in a hash table called the "sentinelList". Sentinel id forms the key for the hash table and the corresponding sentinel object is stored as the value. The hash function gives the handle to the sentinel object, which is used when the user wants to explicitly enable/disable a sentinel. This hash table eliminates the need for traversing the graph to acquire a handle on the sentinel object. Local Event Detector (LED) [16] is used for generating the start and end events. The methods of sentinel are used to raise the events to achieve the desired functionality.

ECAAgent class in LED contains the API to be used for generating events. The ECAAgent instance stores the names of all events and their associated event handles. The handle to ECAAgent is obtained as shown below:

```
import sentinel.led.*;
```

ECAAgent myAgent = ECAAgent.initializeECAAgent();

With the API provided by ECAAgent, the user can create class level and instance level primitive events, composite events and define rules on those events. The following table shows the methods used in creation of events and rules that is used for generating events.

Table 7-1. ECA Agent Class API

Method	Return Type	Description
CreatePrimitiveEvent	EventHandle	This method creates a primitive event of instance level.
createCompositeEvent	EventHandle	This method creates a composite event of instance level.
CreateRule	EventHandle	This method creates an instance level rule on the specified object instance
RaiseBeginEvent	static void	This method raises an event at the beginning of a method.

An event is raised through the "raiseBeginEvent" method. A sentinel object raises this event in its methods "start()" and "end()".The syntax of creating primitive events is

```
createPrimitiveEvent (java.lang.String eventName,
                      java.lang.class className,
                      EventModifier type,
                      java.lang.String methodName,
                      java.lang.Object objectName );
```

Where "className" represents the class associated with the event, "objectName" represents the object raising the event. For sentinel S_{id} that has to start at time t_1 and ends at time t_2. The following events are generated.

EventHandle Start_S_{id} = myAgent.createPrimitiveEvent ("Start_S_{id}",

```
                    "webvigil.Sentinel", EventModifier.Begin,
                    "void start()",sentList.getSentObj(S_id)      );
    EventHandle End_S_id = myAgent.createPrimitiveEvent ("End_S_id",
                    "webvigil..Sentinel",EventModifier.END,
                    "void end()",sentList.getSentObj(S_id)     );
```

The syntax for creating temporal events is

```
    createPrimitiveEvent (java.lang.String eventName,
                          java.lang.class className,
                          java.lang.String timeString)
```

where “timeString” represents the time expression.

The temporal events generated for triggering S_{id} at t_1 and t_2 are shown below

```
    EventHandle Time_t1 = myAgent.createPrimitiveEvent ("t1”,
                                 "webvigil.sentinel",
                                 t1);
    EventHandle Time_t2 = myAgent.createPrimitiveEvent ("t2”,
                                 "webvigil.sentinel",
                                 t2);
```

The syntax for creating rules is

```
    createRule (java.lang.Object targetInstance,
                java.lang.String ruleName,
                EventHandle eventHandle,
                Condition condition,
                Action action)
```

Where “condition” and “action” are the methods that have to be executed on the object represented by “targetInstance”, when the event is raised. “eventHandle” is the handle to the event on which the rule is specified.

The rule associated with the start event is shown below.

```
Rule TimeRule_t1  =    myAgent.createRule (new Rules() ,
                       "startRule", Time_t1,
                        “webvigil.Rules.Condition()”
                       “webvigil.Rules.Action()” );
```

Similarly for all the other events, the rules are generated as above. Here when there is more than one sentinel that has to be started or ended at time t_1, a linked list of all these sentinels (“SentinelList”) is maintained in the “Rules” object. “Condition()” is executed when the event “Time_t_1” is raised. In the activation/deactivation of sentinels the paradigm is E-CA rather than E-C-A and hence the condition is always true. The action method retrieves all the sentinels present in its “SentinelList” and raises the events accordingly. When the sentinel S_i is dependent on other sentinels, say Start_S_i = start_S_j *plus* time, a PLUS (composite event) event is generated [16]. The rules associated with all the events such as absolute and relative have the same functionality.

The other events generated by the ECA Rule generation module are the fetch rules that are involved in fetching the pages. As explained in section 4.3 PERIODIC events are used to achieve the required fetch functionality. The following sections illustrate the implementation details for each fetch type.

7.1.1 Fixed Interval Fetch Event Generation

The syntax for generating periodic event is shown below

```
CreateCompositeEvent(EventType eventType,
                     java.lang.String eventName,
                     EventHandle leftEvent,
                     java.lang.String timeString,
```

EventHandle rightEvent)

Where “EventType” is a composite event [16] such as PERIODIC, AND etc.

For a sentinel S_{id} belonging to fetch type “fixed interval” the following are the events generated.

```
EventHandle Fetch_Sid = CreateCompositeEvent (EventType.PERIODIC,
                            “Fetch_Sid”, Start_Sid, time-interval, End_Sid );
```

Here *“time-interval”* is the user defined polling frequency. The rules associated with event “Fetch_S_{id}” fetch the pages. They are more detailed in [8].

7.1.2 <u>On-Change Fetch Event Generation</u>

All the sentinels belonging to this category share the same fetch event. As explained in section 4.3, dummy events are generated to achieve the functionality. These dummy events are primitive events. The fetch rule generated is shown below

```
EventHandle Fetch_Sid = CreateCompositeEvent (EventType.PERIODIC,
                            “Fetch_URLi”, Dummy_Starturl, time-interval,
                             Dummy_Endurl );
```

The pseudo code for action part of “StartRule_S_{id}” and “EndRule_S_{id}” is shown below

```
Action Part of StartRule_Sid
1. Query URLi node for the number of active sentinels on URLi belonging to on-change
   type
2. If the number of active sentinels is zero then raise the event Dummy_Starturl
3. Enable the sentinel Sid

Action Part of EndRule_Sid
1. Query URLi node for the number of active sentinels on URLi belonging to on-change
   type
2. If the number of active sentinels is one then raise the event Dummy_Endurl
3. Disable the sentinel Sid
```

7.2 Implementation of Change Detection Graph

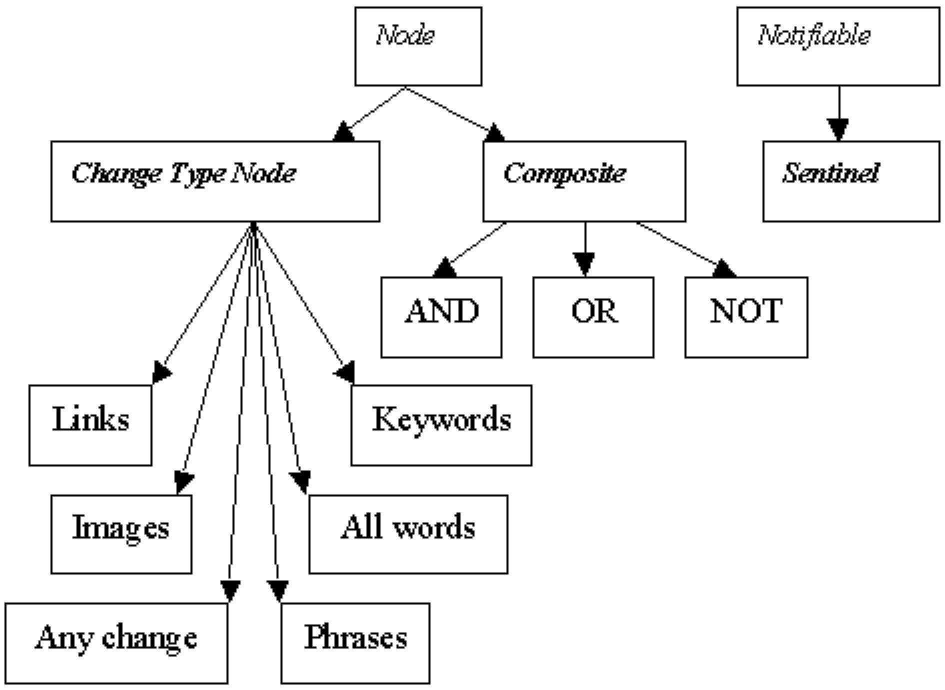


Figure 7.1. CDG Class Hierarchy

Figure 7.1 depicts the hierarchy among the key classes used in the implementation. Primitive changes are the different changes monitored on the pages (links, images, any-change, all-words, phrases and keywords). The composite changes constitute NOT, OR and AND. In, Figure 7.1 each box represents a class. The classes *Change Type Node* and *Composite* are abstract classes, the classes Node and Notifiable are interface classes and all the other classes are normal classes. The Node interface has an insertSentinel method that is implemented by all the event classes. The insertSentinel method implements the addition of sentinels to the grouping structure. The classes AND, OR, and NOT are normal classes whose instances represent the composite event nodes in the change detection graph. Each URL representing the page monitored belongs to the class *URLNode*. A list of all the URL's and their corresponding references are maintained in hash table termed as *URL Node List*. The following table describes the methods of the class URLNode. The URLNode class has a linked list associated with it. This linked list contains the references to all the ChangeTypeNode objects.

Table 7-2: Member Functions of URLNode Class

Method	Description
InsertChangeType	Inserts the change type nodes into the subscriber list when they are created. If properly inserted returns true.
GetNumberOfSentinels(*fetch-type*)	Returns the number of active sentinels monitoring this page. This method is called in the action part of sentinels belonging to on-change.
PropogateVersion(*page-type*)	Invoked when a version corresponding to the URL is fetched. This version is propagated to all its subscribers.

Change Parameters

The events (primitive and composite) are associated with a ChangeList that is passed to the sentinels. The change list constitutes the change detected, the versions on which the change is detected, the type of the page (HTML/XML) and the time at which the change is detected. The change is a vector containing three lists; i). insert list containing objects (links, images, words, phrases) that are inserted. ii). Delete list containing the objects that are deleted and iii) move list containing objects that are moved. A primitive event is associated with a single ChangeList whereas a composite event is associated with multiple sets of ChangeList, that is, the collection of the ChangeList of all the constituent events. This collection is called ListOfChangeLists.

Table 7-3: Description of ChangeList and ListOfChangeLists Classes

Class Name	Description
ChangeList	Created at the ChangeTypeNode's to store the change parameters and are propagated to the sentinels.
ListOfChangeLists	Linked List containing the ChangeList's.

The following table shows some of the methods in the sentinel class. It contains the reference to the producer (ChangeTypeNode or Composite) and also a subscriber reference to which it acts as a representative (Composite). It also contains a flag indicating its state (enabled/disabled).

Table 7-4: Member Functions of Sentinel Class

Method	Description
Enable()	Changes its state to enabled and informs its producer. It also updates the knowledge base. This method is invoked in the start rule associated with this sentinel.
Disable()	Changes its state to disabled and informs its producer. It also updates the knowledge base. Return type is void. This is invoked in the end rule associated with this sentinel.
Notify(ChangeList change, Boolean flag)	This method is invoked when ChangeTypeNode computes change. The flag is true if there is a change and false if there is no change. If the subscriber list is empty and the change is true the notification module is notified about the change. If the change is false the notification module is not informed. If there is a subscriber (Composite) it is notified of the computed change with the flag.

The following table shows some of the methods of ChangeTypeNode. It contains the following references; i). Change detection algorithms (HTML/XML), ii). Grouping structure, iii). Subscriber list containing sentinels, iv). Producer (URLNode), and v). Version Controller.

Table 7-5: Member Functions of Change Type Node Class

Method	Description
InsertSentinel(Sentinel)	Called when a new sentinel is inserted. The grouping structure takes care of the insertion of the sentinel in the proper group.
Propagate()	Invoked for propagating changes to the sentinels when a change is computed. If there is a change then a flag is associated with
DetectChange(Version)	Invoked by URLNode to propagate a version. The version manager is contacted for the corresponding previous version based on the group and based on the version type (HTML/XML) the corresponding change detection algorithms are invoked. The corresponding sentinels are informed about the change and a flag indicating whether the change is detected.

7.2.1 Composite Change Detection

This section discusses the implementation of change detection mechanism at the composite node. The NOT node contains the reference of the sentinel subscribed to it and also the reference to the proxy sentinel created to represent this event at its child node. The following table shows the methods

Table 7-6: Member Functions of NOT Class

Method	Description
Propagate (ListOfChangeLists list, Boolean flag)	The proxy sentinel created at the child node invokes this method. The flag is negated and the subscriber (Sentinel) is notified.
InsertSentinel (Sentinel sentinel, Node node)	Adds the sentinel to its subscriber list. A proxy sentinel is created and the InsertSentinel method of the node is called.

The following table shows the important methods in OR class. The OR node contains the references of both the children i.e., the references to the sentinels that represent it. It also contains the reference of the sentinel, which is subscribed to this event (OR).

Table 7-7: Member Functions of OR Class

Method	Description
Propagate(ListOfChangeLists list, Boolean flag)	The proxy sentinel created at the child node invokes this method. The list contains the changes computed and the flag denotes whether there is a change detected or not. If the flag is true, only then the sentinel subscribed to it is notified.
InsertSentinel(Sentinel sentinel, Node leftNode, Node rightNode)	Adds the sentinel to its subscriber list. Two proxy sentinels are created and the InsertSentinel method of the leftNode and rightNode is called.

The AND node contains two additional data structures with respect to the OR node. As the change should be detected on two versions of the same page, the change detection should wait until the changes are obtained from both the children. These changes are stored in a change table represented by the class ChangeTable. The AND node contains two change tables for each of its children. A change table consists of a set of entries. Each entry in the change table denotes an event occurrence. An event entry consists of a ListOfChangeLists and *changeflag* indicating whether a change is detected or not at the lower level. The steps for detecting AND events are shown below.

```
1. If the ListOfChangeLists and changeflag is propagated from the left event
2.      If the left table is not empty
3.           Remove all the entries in the leftchange table, thus maintaining the latest change event (on the same versions of the page).
4.           For every entry in the right table, logical AND is computed on the changeflag received from the left event and ListOfChangeLists are merged together and the merged ListOfChangeLists and computed changeflag is propagated to the sentinel subscribed to it.
5.      Else
6.           Add the ListOfChangeLists and changeflag to the table
7. If the ListOfChangeLists and changeflag is propagated from the right event
8.      If the right table is not empty
9.           Remove all the entries in the rightchange table, thus maintaining the latest change event.
10.          For every entry in the left table, logical AND is computed on the changeflag received from the right event and ListOfChangeLists are merged together and the merged ListOfChangeLists and computed changeflag is propagated to the sentinel subscribed to it.
11.     Else
12.          Add the ListOfChangeLists and changeflag to the table
```

The following table shows the methods.

Table 7-8: Member Functions of AND Class

Method	Description
Propagate(ListOfChangeLists list, Boolean flag)	The proxy sentinel created at the child node invokes this method. The event is detected as explained above and the subscriber (Sentinel) is notified with the merged lists and the flag indicating the detection of event.
InsertSentinel(Sentinel sentinel, Node leftNode, Node rightNode)	Adds the sentinel to its subscriber list. Two proxy sentinels are created and the InsertSentinel method of the rightNode and leftNode are called.

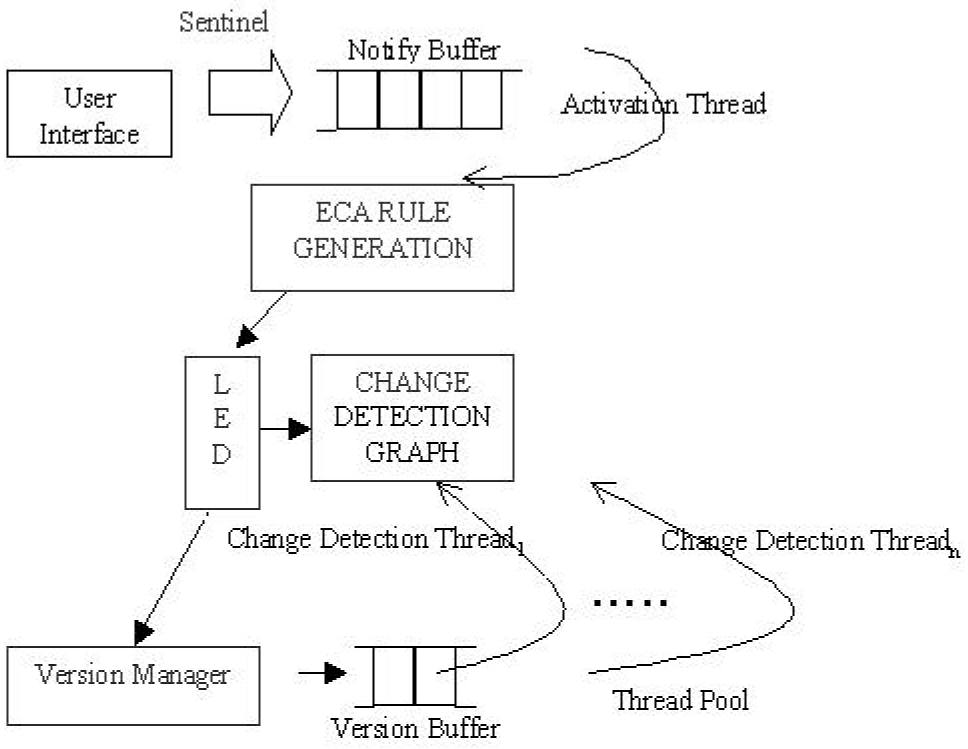


Figure 7.2: Complete System

7.3 Multithreading Issues

As shown in the Figure 7.2 when the user registers a sentinel with the system, the activation thread creates the required ECA rules and the corresponding nodes in the CDG. Once the fetch rule associated with the sentinel fires, the version of the page is fetched and the version manager is informed. The version manager puts the version in the version buffer. The thread waiting on the version buffer removes the version and propagates it to the CDG for change detection. This means that if there are versions fetched and waiting in the buffer to be serviced they will be handled serially by the CDG. This wait will be significant when there are several fetch rules fetching the pages. In order to make the CDG scalable it should be able to detect changes to multiple pages concurrently. Hence the first design goal is to have a multitasking CDG. Multitasking can be achieved by multithreading. In case of multithreading each version will be serviced in a separate thread. When changes are detected concurrently, two or more threads may be accessing the same data structures at a given time. To prevent race conditions, appropriate synchronization mechanisms must be provided for the protection of data structures. However, locking of data structures must not be so coarse-grained that it will effectively serialize their access. Hence synchronization mechanisms must be carefully chosen to a fine granularity of locking and to maximize concurrency.

7.3.1 Multithreading the Change Detection

In order to make the CDG multithreaded a new thread can be spawned for each version of page inserted into the version buffer. Versions are fetched at a faster rate as it involves only fetching the page, whereas change detection involves invoking the different change detection algorithms. Spawning a new thread for each version, results in creation of more number of threads that in turn reduce the system performance due to context switches. This can be avoided by maintaining a thread pool as shown in Figure 7.2.

Hence the parallelism is restricted to the size of the thread pool. The size of the thread pool is configurable and can be set to any value. This value can be decided through experiments.

7.3.2 Synchronization Issues

The System is made up of several data structures that will be shared and hence may be concurrently accessed by threads. Following is the list of shared data structures:

1. Notify Buffer: This buffer is handled by the UI (User interface) to insert the new registrations; the activation thread waiting on the buffer processes these requests.
2. Version Buffer: Queue containing the versions of pages. The fetch rules associated with each sentinel insert the versions of pages fetched and the change detection threads (Thread pool) retrieve each version for change detection.
3. Change Detection Graph (CDG): Graph of URL nodes, change type nodes, and operator nodes. Accessed by change detection threads for propagation of versions and also by the rules in LED when enabling/disabling a sentinel. When a sentinel is enabled/disabled, the sentinel information is propagated to the corresponding nodes (URL node, change type node) in the graph.
4. URL Node List: List of URL nodes. The LED (Local Event Detector) and the change detection threads share this list. The rules associated with a sentinel's start/ end time access this list to insert/delete the URL node corresponding to the sentinel. The change detection threads access this list to retrieve the URL node corresponding to the version of page.

Race Conditions.

When the result of two or more threads performing an operation depends on unpredictable timing factors, there is race condition. There are a number of situations where race conditions can occur during change detection. The use of buffers (for decoupling the computations) and asynchronous actions (such as delete, disable etc.) can occur at any time the changes are being detected. Before we describe the proposed solution, we give a few examples of scenarios where race conditions can occur.

1. Thread A is in the process of deleting a URL node at position 7 from the *URL Node list*. Thread B is traversing the *URL Node list* to get the URL node at position 13 to which it wants to propagate the version. Thread B could be looking at node 7 when the list manipulation is occurring. Thread B will decide that node 7 is not the desired node and moves to the next position in the list. However, since thread A has disconnected this node from the list the next position could be NULL. The result of what thread B reads will hence depend on the timing factor and has been compromised by the race condition. Hence the access of the *URL node list* and several such shared data structures must be guarded for mutual exclusion.

2. Consider only one sentinel registered (links changes on page p). Thread A will be accessing the CDG while propagating the version of the page from the URL node (representing p) to the change type node (links). Even before the propagation, when sentinel reaches end time, thread B (end rule) will delete the links node since there are no more sentinels interested on links change leading to a race condition. Mutual exclusion can be attained using synchronization mechanisms or locks. In addition, locks are also used for controlling the sequence of execution of threads.

3. The fetch rule is inserting versions of the same page at a faster rate. Thread A from the thread pool starts and is processing $version_1$. As there is no control over which thread is preempted and which thread is executed by the operating system, another thread B (belonging to the thread pool) can start processing $version_2$ and detects change before $version_1$ is processed. This results in incorrect change detection.

In order to control this execution, locks are used. There are several types of locks [23] (semaphores) and the right choice must be made.

7.3.2.1 Types of Locks

Mutex lock is a synchronization primitive that allows multiple threads to synchronize access to shared data by providing mutual exclusion. The mutex lock has only 2 states: locked and unlocked. Once a thread has acquired the mutex lock on a data structure other threads attempting to lock the structure will be blocked until it is unlocked. Since mutex allows only one thread to access any data at a given time, it is the most restrictive type of access control. For example, when a mutex is used to synchronize access to a list, the mutex will control the entire list. While the list is being accessed by one thread it is unavailable to all other threads. If most accesses are reads and writes of the existing nodes as opposed to insertions and removes, then a more efficient approach will be to allow items in the list to be individually locked.

Read-write lock is another synchronization primitive that was designed specifically for situations where shared data is read often by multiple threads/ tasks and rarely written. A read-write lock is similar to a mutex lock except that it allows multiple threads to concurrently acquire the read lock whereas only one writer at a time may acquire a write lock. In the current scenario the *Insert* or *delete* operation on a list will require acquiring the read-write lock in the *writelock* mode, while the seek (search) of a node will require acquiring the lock in the *readlock* mode. By using the read-write locks we can have parallel search operations on the URL node list. The only drawback of using read-write locks is that locking operations take more time than the locking operations on mutexes. Hence locking strategy must be chosen carefully. Read-write locks are justified for the URL node list where inserts to the list happen not that often, only when new pages are requested for monitoring; thereafter all other operations are search operations on the graph to find a particular node. *Readlock* mode can be used to allow threads to search the list in parallel.

Semaphore is a synchronization primitive that has a value associated with it, which is the number of shared resources regulated by the semaphore. Whenever a thread acquires a semaphore, the semaphore count is decreased by 1. Whenever a thread releases a semaphore, its count is increased by 1. Any thread wanting to acquire the semaphore must wait till its count is greater than 0. Traditionally, semaphore operations have been known as P and V operations. P operation is equivalent to acquiring the semaphore. V operation is the same as releasing the semaphore. Semaphores are used primarily when there is more than one shared resource that needs to be regulated.

For synchronization of data structures in the system, mutex locks or semaphores can be used when the operations involved are primarily inserts and deletes that require exclusive access. For data structures such as the *Url node list*, where a majority of the operations are search operations on the list and updates on individual nodes, read-write locks can be used for locking the list and semaphore or mutex locks can be used for locking individual nodes. Details of the locking algorithm are explained in the next section. Table 5-1 shows the choice of locks made for locking the various data structures.

7.3.2.2 Locking of Change Detection Graph

In a multithreaded system, several threads of execution share the Change Detection Graph (CDG), and access to the graph has to be synchronized. Using a mutex lock for the CDG locking will give only two states (locked and unlocked) of access for the entire graph so that only one thread can be accessing it at any time. To allow finer granularity, more than one thread should be able to access independent nodes of the graph concurrently, as long as they are not updating the same nodes. Since in the graph the processing at each lower level node depends on the information at the higher-level nodes, locking each node will not solve the synchronization issue. Hence a lock has to be obtained on the leaf node (URL node). Once the leaf node is locked, other threads cannot access higher-level nodes connected through the leaf node. Access to all higher-level

Table 7-9: Data Structures and Synchronization

DATA STRUCTURES	LOCK USED with RATIONALE
Notify Buffer, Version Buffer	Mutex locks are used since operations used are primarily inserts and deletes. These operations need an exclusive lock mode that is provided by mutex locks**.** Using mutex locks is preferred to read-write locks because an operation on read-write lock has a high overhead.
URL Node List	Read Write locks are used for locking the list, as operations on the list are primarily search of the list to find an individual node. Shared mode (read lock) can be used while scanning the list to allow parallel scans and exclusive mode (write lock) is needed when nodes are inserted or deleted from the list.
Change Detection Graph (CDG)	Read-write lock for locking URL node list. Write lock provides exclusive access to graph while inserting or deleting a node. When accessing list in shared (read) mode, lock hash table is used for managing access to individual nodes. Lock hash table minimizes number of semaphores needed to lock nodes of the CDG. Thread suspend and continue calls are used to prevent more than one thread from accessing any node at a time. Lock hash table minimizes overhead of managing several locks.

nodes in the graph has to be started at the leaf node. One way to achieve a finer granularity would be to have a read-write lock on the change detection graph and a semaphore lock on each URL node of the tree. However, when a large number of pages are monitored, the number of URL nodes in the graph will grow. Allocating and maintaining locks for each and every URL node of the graph is cumbersome and will require too many locks. A better option is to maintain a hash table of the URL nodes of the graph that are currently being accessed. Each URL node of the graph will hash to a bucket of the hash table. The bucket will maintain a list whose elements represent the Ids of URL nodes of the CDG currently being accessed. Thread IDs of threads waiting for a particular node will also be saved in a queue for each element in the list. In order to traverse the list of node IDs the bucket needs to be locked. This means that the maximum number on semaphore locks required for synchronizing access to the CDG is equal to the number of buckets. In this way number of locks to be maintained is minimized and at the same time a fine granularity of locking is achieved for locking the CDG.

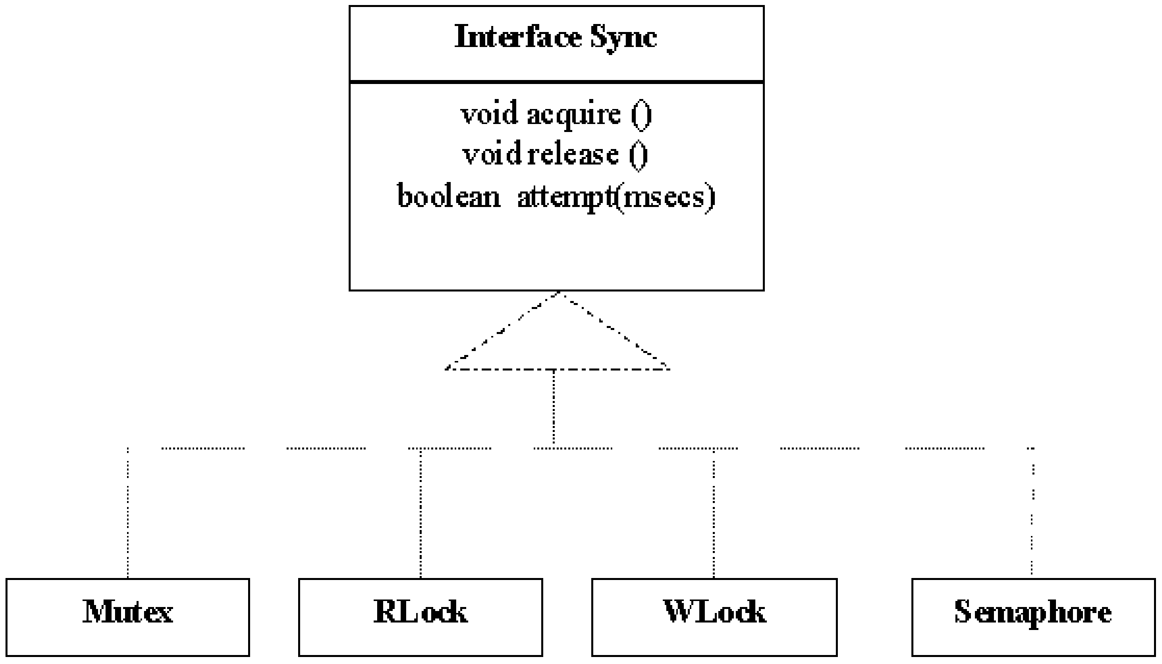


Figure 7.3. Class diagram of lock package in WebVigiL

7.3.3 Implementation of Locks [23]

The locks used in accessing different data structures have been explained in chapter 5. This section explains the implementation of these locks. All locks belong to the *webvigil.locks* package. The locks in this package provide three different types of synchronization protocols. They are:

1. Sync: acquire/release protocols
2. Channel: put/take protocols
3. Executor: executing Runnable tasks

WebVigiL uses only one protocol, the sync protocol. All the locks, Mutex, ReadWrite and Semaphore locks in this protocol implement Sync interface. This interface provides three methods, *acquire(), release()* and *attempt(),* which the locks override. The first method *acquire()* is used when a lock is needed to be acquired. It is essential when a thread needs to enter a synchronized block. The thread that enters the critical section or synchronized block (in JAVA jargon) needs to release the lock to let other threads waiting to enter the critical section. The *release()* method is used for this purpose. The other method *attempt()* is used to acquire a lock within a specified amount of time. Read/Write locks come with the facility to control the number of readers and writers. It also provides mechanisms to assign priorities to readers and writers. WebVigiL does not need locks with such priority. Read Write locks in WebVigiL are used only for issuing read locks and write locks. It should be noted that the locks in this package are non-reentrant meaning the thread that owns a lock has to wait for that lock till it releases. The relationship between different locks in this package is shown in Figure 7.3.

7.3.3.1 Lock Hash Table

The classes defined for the hash table are *HTLock , HBucket* and *HLink*. HT*Lock* is the lock hash table class. There is a single instance of this class in the system. *HB*ucket is the class for each bucket of the hash table. Each bucket is guarded by a semaphore *bucket_sema*, and each bucket contains a chain *of HLink.* The *HLink* contains *obj_id, thr_id* , *next* and *nextp*. *Obj_id* is the unique ID (address of the node is used for hashing) of the URL node being accessed by a thread whose thread id is *thr_id*. *next* is a pointer to the next *HLink* in the bucket chain. *nextp* is a pointer to an *HLink* which contains the *thr_id* of a thread that is suspended and waiting to access the same URL node. Figure 7.4 gives the data structure of the lock hash table.

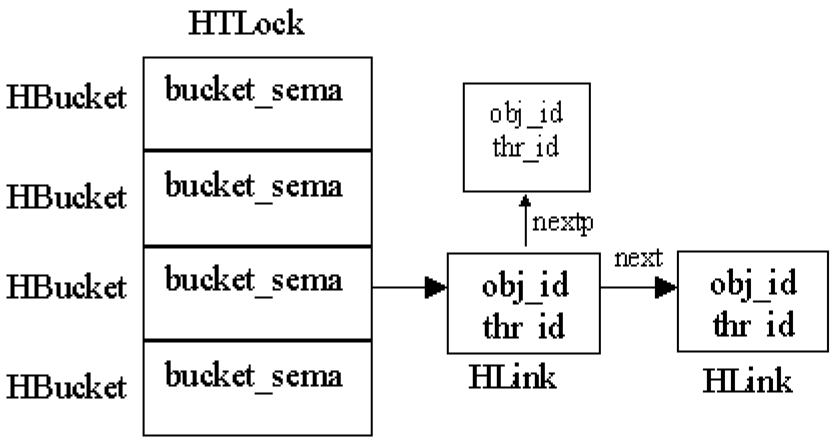


Figure 7.4.Lock Hash Table Data Structure

When a version of the page is fetched, it is propagated to the corresponding URL node. For traversal, read-write locks are used to give shared access to the URL node list. Lock hash table is used to access individual nodes when the version of page is propagated. Each node of the CDG must have a unique object ID for hashing purposes. Since the address of the node is unique, it is used as the object ID. The sequence of operations needed for locking is as follows: First the node object is hashed to find its bucket in the hash table. A semaphore lock (*bucket_sema)* is then acquired on the bucket so that no two threads may be accessing its chain of *HLink* at the same time. The bucket chain is then searched for the object ID of the URL node. If the object ID is not found, it means that no other thread is accessing this node. Then an *HLink* containing that node's object ID and thread ID are added to the bucket's *HLink* chain, the bucket semaphore is released, and the current thread is granted access to the URL node. The thread can now detect changes that are monitored on that page. On the other hand, if the object ID is found in the chain, it means that another thread is operating on the node at the same time. The current thread's thread ID is added to the list of waiting threads for that URL node. *bucket_sema* is released once the object ID is located, so that other threads can traverse the *HLink* chain for accessing nodes of the CDG. The current thread is suspended and will be continued only when the desired URL node becomes available to it. After a thread finishes accessing the URL node, it removes its thread ID from the *HLink* chain. The next thread in the queue of suspended threads is released by a "*thread_resume*" and it can now access that URL node. Figure 7.5 gives the locking algorithm.

```
LOCKING URL NODE

bucket_id = hashing(object_id)
P(bucket_sema(bucket_id))
Search hbucket for object_id
Found object_id:
   -insert into vertical chain(queue of
    waiting threads)
   -V(bucket_sema(bucket_id))
   -suspend(current_thread_id)
Else
   -insert into bucket chain
   -V(bucket_sema(bucket_id))
```

```
RELEASING URL NODE

bucket_id = hashing(object_id)
P(bucket_sema(bucket_id))
Search hbucket chain for object_id
If nextp != null
   -V(bucket_sema(bucket_id))
   -resume the next thread in
    waiting thread queue
Else
   -delete hlink from chain
   -V(bucket_sema(bucket_id))
```

Figure 7.5. Locking Algorithm for URL nodes

CHAPTER 8

CONCLUSIONS AND FUTURE WORK

8.1 Conclusion

WebVigiL is envisioned as a complete system that allows monitoring and notification of changes to structured documents in a distributed environment. WebVigiL is a system currently under development at ITLAB at The University of Texas at Arlington for providing an alternative paradigm for monitoring changes to the web (or any structured document). The contribution of this thesis towards the system is in the following areas:

- Design and Development of ECA Rule Generation
- Construction and Maintaining Change Detection Graph
- Storage and Retrieval of pages

In the ECA Rule Generation module, ECA Rules to support enabling and disabling of sentinels based on the start and end events and fetch rules for fetching pages for on-change and fixed-interval option has been designed and implemented. Construction and maintaining of change detection graph to support primitive events (changes to links, images, keywords, phrases, any-change and all-words) and composite events (NOT, AND and OR) has been designed and implemented. The grouping structure for efficient change detection based on the fetch type and compare-options has been developed. The multithreading and synchronization techniques have been designed and tested for correctness. The storage structure for storing the versions of the pages being fetched is developed and implemented. All these modules have been integrated into the WebVigiL system.

8.2 Future work

Currently sentinels belonging to the category of fixed-interval fetch type have individual fetch rules. Grouping of these sentinels together can be investigated. Pages having frames are not handled. A page with multiple frames has a base page, in which the reference to the pages in frames is given. Hence the base page is a set of references to various other pages. Change detection to these pages can be achieved by having a composite change on all the referenced pages and the base page. This composite change detection across multiple pages can be incorporated to handle pages with multiple frames.

In the current implementation the number of change detection threads is set to ten. This thread pool size can be determined based on testing and analyzing the time taken for change detection by each thread. Persistence and recovery issues with respect to the system have to be dealt.

APPENDIX A

HASH FUNCTIONS

<u>Function 1</u>

```
hashValue = x;
for( i=0; i<n; i++ )
        hashValue = 131*hashValue + key[i];
hashValue = i % tableSize;
```

where *n* is the length of the key (string), *key[i]* is the *i*th character of the key, tableSize represents the length of the hash table (number of buckets), *x* is initialized to a random number preferably a prime number.

<u>Function 2</u>

```
hashValue = x;
for (i=n-1; i>=0; i--)
        hashValue = ((hashValue<<5)^(hashValue>>27))^key[i];
hashValue = hashValue % tableSize;
```

here *x* equals 0.6180339887. This number is called magic number (sqrt(5)-1/2) [22]. The other variables are same as in Function 1.

<u>Function 3 (Java Hash)</u>

```
hashValue = 0;
for( i=0; i<n; i++ )
        hashValue = hashValue + key[i]*31^(n-1);
hashValue = hashValue % tableSize;
```

where *key[i]* is the *i*th character of the key, *n* is the length of the key. The other variables are same as in Function 1.

REFERENCES

[1] 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.

[2] Chen, Y.-F. and E. Koutsofios. *WebCiao: A Website Visualization and Tracking System*. in *WebNet97*. 1997.

[3] Mind-it, *http://www.netmind.com/.*

[4] Lu, B., S.C. Hui, and Y. Zhang. *Personalized Information Monitoring over the Web*. in *First International Conference on Information Technology and Applications (ICITA)*. 2002. Australia.

[5] Liu, L., C. Pu, and W. Tang. *WebCQ: Detecting and Delivering Information Changes on the Web*. in *Proceedings of International Conference on Information and Knowledge Management (CIKM)*. 2000. Washington D.C: ACM Press.

[6] Xyleme, *http://www.xyleme.com/.*

[7] Jacob, J., *WebVigiL: Sentinel specification and user-intent based change detection for Extensible Markup Language (XML)*. 2003, The University of Texas at Arlington.

[8] Chakravarthy, S. and D. Mishra, *Snoop: An Expressive Event Specification Language for Active Databases*. Data and Knowledge Engineering, 1994. **14**(10): p. 1--26.

[9] Pandrangi, N., *WebVigiL: Adaptive fetching and user-profile based change detection of HTML pages*. 2003, The University of Texas at Arlington.

[10] 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.

[11] Tanpisut, W., *Design and Implementation of Event based subscription/notification paradigm for distributed environments.* 2001, The University of Texas at Arlington.

[12] Anwar, E., L. Maugis, and S. Chakravarthy, *A New Perspective on Rule Support for Object-Oriented Databases*, in *1993 ACM SIGMOD Conf. on Management of Data.* 1993: Washington D.C. p. 99-108.

[13] Chakravarthy, S., et al., *Design of Sentinel: An Object-Oriented DBMS with Event-Based Rules.* Information and Software Technology, 1994. **36**(9): p. 559--568.

[14] Mishra, D., *SNOOP: An Event Specification Language for Active Databases*, in *MS Thesis*. 1991, Database Systems R&D Center CIS Department University of Florida, E470-CSE, Gainesville, FL 32611.

[15] Stonebraker, M. and G. Kemnitz, *The Postgres Next-Generation Database Management System.* Communications of the ACM, 1991. **34**(10): p. 78--92.

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

[17] Dasari, R., *Events And Rules For JAVA: Design And Implemenation Of A Seamless Approach*, in *Database Systems R&D Center, CIS Department*. 1999, University of Florida: Gainesville.

[18] Pandrangi, N., et al. *WebVigiL: User Profile-Based Change Detection for HTML/XML Documents.* in *Twentieth British National Conference on Databases*. 2003. Coventry, UK.

[19] Gatziu, S. and K.R. Dittrich, *SAMOS: an Active, Object-Oriented Database System.* in IEEE Quarterly Bulletin on Data Engineering, 1992. **15**(1-4): p. 23--26.

[20] Gehani, N. and H.V. Jagadish, *Active Database Facilities in Ode*. IEEE Bulletin of the Technical Committee on Data Engineering, 1992. **15**(1-4).

[21] 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.

[22] Knuth, D.E., *The Art of Computer Programming*. 3 ed. Vol. 3. 1998: Addison-Wesley.

[23] Lea, D., *Concurrent Programming in Java. Second Editio, 2000.*

BIOGRAPHICAL INFORMATION

Anoop Sanka was born on June 20, 1978 in Rajahmundry, India. He received his Bachelor of Technology degree in Computer Science and Engineering from National Institute of Engineering, Mysore, India in September 1999. In the Fall of 2000, he started his graduate studies in Computer Science and Engineering at The University of Texas, Arlington. He received his Master of Science in Computer Science and Engineering from The University of Texas at Arlington, in December 2003. His research interests include active databases and Web technologies.