A Framework for Superimposed Applications : Techniques to Represent, Access, Transform, and Interchange Bi-level Information

Sudarshan Srivivasa Murthy
Portland State University

A FRAMEWORK FOR SUPERIMPOSED APPLICATIONS: TECHNIQUES TO REPRESENT, ACCESS, TRANSFORM, AND INTERCHANGE BI-LEVEL INFORMATION

by

SUDARSHAN SRINIVASA MURTHY

A dissertation submitted in partial fulfillment of the
requirements for the degree of

DOCTOR OF PHILOSOPHY
in
COMPUTER SCIENCE

Portland State University

DISSERTATION APPROVAL

The abstract and dissertation of Sudarshan Srinivasa Murthy for the Doctor of Philosophy in Computer Science were presented March 12, 2009, and accepted by the dissertation committee and the doctoral program.

COMMITTEE APPROVALS:

[redacted]
David Maier, Chair

[redacted]
Cynthia Brown

[redacted]
Lois Delcambre

[redacted]
Alon Halevy

[redacted]
Fei Xie

[redacted]
Kenneth Cruikshank
Representative of the Office of Graduate Studies

DOCTORAL PROGRAM APPROVAL:

[redacted]
Wu-chi Feng, Director
Computer Science Ph.D. Program

ABSTRACT

An abstract of the dissertation of Sudarshan Srinivasa Murthy for the Doctor of Philosophy in Computer Science presented March 12, 2009.

Title: A Framework for Superimposed Applications: Techniques to Represent, Access, Transform, and Interchange Bi-level Information

Superimposed applications (SAs) superimpose (that is, overlay) new information and structures (such as annotations) on parts (such as sub-documents) of existing *base information* (BI). In this setting, SA developers and users work with *bi-level information*, a combination of the superimposed information and the referenced BI parts.

We have designed a framework to assist SAs in the following bi-level-information-management activities: *representation*, *access*, *transformation*, and *interchange*. This framework defines the abstraction *context agent* to activate any BI part and to retrieve information from the context of the part. It includes means to represent and access bi-level information in a conceptual model (the Entity-Relationship model augmented with *relationship patterns*), the relational model, the XML model, and an object model. It defines a mechanism to transform bi-level information to alternative forms using

queries expressed in *existing query languages* and executed by *existing query processors*. It also includes a formal model to improve the expression and execution of certain classes of queries. Finally, the framework employs the notion of *SI dependency graphs* to facilitate interchanging of bi-level information among SA users.

Specifically for the XML model, the framework defines *Sixml*, an XML markup language to represent bi-level information; *Sixml DOM*, an extension of the XML Document Object Model (DOM) to efficiently manipulate Sixml documents at run time; and *Sixml Navigator*, an alternative path navigator that improves both query expression and execution.

Using our framework, an SA can reference heterogeneous BI parts *in situ*, allowing multiple simultaneous organizations of the same BI parts, while preserving their original context. Also, the SA developer may employ the data model and schema that is appropriate for each SA.

We have evaluated each framework component using a method appropriate for the component. For example, we have implemented the context-agent abstraction to reference BI parts of the following types: Microsoft Word, Excel, and PowerPoint; XML, HTML, PDF, audio, and video. We have built six SAs that employ distinct schemas (in different data models). We have implemented the design of Sixml DOM and Sixml Navigator, used them with existing query processors, and experimentally evaluated the implementations' performance.

Acknowledgements

I thank my advisor David Maier for his patient guidance throughout this research. Each time I had an "idea", he tried to find some use for it, or found a way to improve it. (Frankly, I wonder how he tolerated some of my early ideas.) In the process, he gently helped me build and improve my own process to create and validate ideas. Today, I confidently say that the contributions in this research are mine, but I also categorically say that each contribution is better because of Dave.

I thank Lois Delcambre for her support throughout this research. I thank the other members of my committee—Cynthia Brown, Kenneth Cruikshank, Alon Halevy, and Fei Xie—for their valuable advice.

I thank my research collaborators at Virginia Tech and Villanova University for validating key portions of this research. I particularly thank Ed Fox, Kapil Ahuja, Uma Murthy, and Lillian Cassel. I also thank the students at Virginia Tech and at the School of Science and Technology, Beaverton for trying out some applications of this research.

Several colleagues provided constructive feedback along the way. I am especially thankful to Dave Archer, Shawn Bowers, Laura Bright, Rafael Fernández-Moctezuma, Bill Howe, Jin Li, Vassilis Papadimos, Susan Price, Nick Rayner, Len Shapiro, James Terwilliger, Pete Tucker, Kristin Tufte, and Mathew Weaver.

Jo Ann Binkerd, Dana Director, Lorie Gookin, Cindy Pfaltzgraff, Shiva Gudetti, Kathi Lee, Renee Remillard, and Leai Rose, all cheerfully helped me in various administrative matters.

I am grateful to the US National Science Foundation, whose funding partially supported this research.

I am deeply indebted to my parents for inculcating in me the curiosity and commitment needed to engage in any research. Dad, I am sorry I did not complete this research soon enough.

My wife must love me a lot, because nothing else can explain her sacrifice and patience during this research. Thank you, Karin, for holding the fort, for reviewing the drafts, and for always keeping a smile. Also, thanks for gently nudging me to complete this research soon so we can focus on other things.

Table of Contents

List of Tables

List of Figures

1. Introduction

Imagine a researcher co-authoring a paper. In his research for the paper, he finds relevant information in a variety of sources: HTML (Hypertext Markup Language [61]) pages on the World Wide Web (*the web*), PDF (Portable Document Format [6]) documents on the web and on compact discs (CDs), Microsoft (MS) Excel spreadsheets and MS Word [96] documents on the local file system, and so on. He identifies relevant *sub-documents* (that is, portions of documents) and adds annotations containing clarifications, questions, and conclusions in reference to the sub-documents. He frequently reorganizes the information he has collected and the added annotations to reflect his current perspective. He intentionally keeps his information structure loose so he can easily rearrange the content. When he has collected sufficient information, he imports the sub-documents and his annotations into a word-processor document.

As he writes his part of the paper using a word-processor, the researcher may revisit his sources to review information in its original context. For example, he may view a selection in a PDF document using Adobe Acrobat (Acrobat) [8]. Also, as he writes the paper, he may sometimes reorganize its contents, including the imported information, to suit the flow. He may search within an imported annotation, the annotated sub-document, or the surrounding context of the sub-document. He may mix some of the imported information with other information in the paper and transform the mixture to suit his presentation needs. At one or more points in the development of the paper, he sends his version of the paper to his co-authors, possibly along with the background material he has collected.

1.1. The Real-World Objective

Most researchers will be familiar with manual approaches to scenarios similar to the one just described. They may also be familiar with approaches that involve digital documents and annotations. This dissertation is concerned with the digital approaches.

There are two kinds of digital annotations: inline and stand-off. An *inline annotation* is stored within the annotated document, placed near the relevant target sub-document, similarly to an annotation made on a hard copy of the document. An inline annotation makes it easy to relate the annotation to its target, but it assumes the annotator owns the target document. A *stand-off annotation* is stored separately from the target document, using *some* means of relating the annotation with its target. (For example, the annotation may include a page number.) A stand-off annotation makes it harder to relate the annotation to its target, but it does not require the annotator to own the target document because the annotation is *superimposed* on its target.

Stand-off annotations facilitate multiple simultaneous organizations of existing information, without replicating that information. For example, a reader might superimpose a list structure over a set of sub-documents, whereas another reader might superimpose a hierarchy over the same set of sub-documents. Figure 1.1 illustrates such superimposed structures.

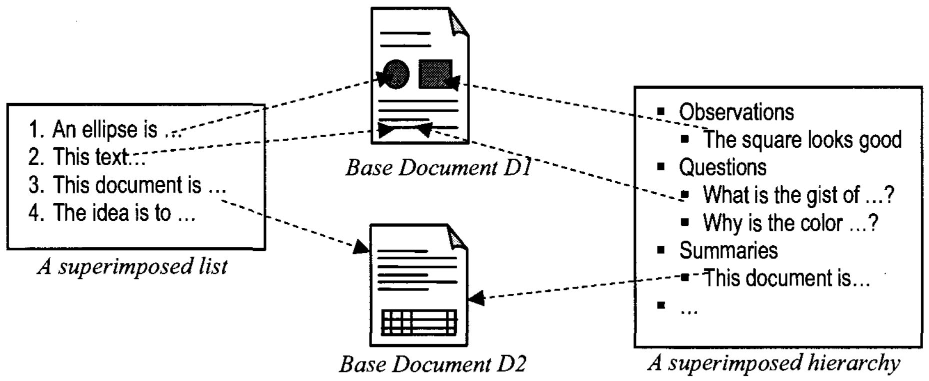


Figure 1.1: Multiple information structures superimposed over existing information. A dashed arrow denotes a reference to an existing document or a sub-document.

This dissertation is concerned with *superimposed information* (SI), which is new information and structures overlaid on *base information* (BI), which is existing documents and sub-documents. For example, a reader's annotation on a text selection in a PDF document is SI. The annotated PDF document itself is BI.

Broadly, our real-world objective is to support the design, development, and deployment of applications that facilitate SI. Specifically, we aim to support the following application capabilities:

1. Select arbitrary portions of BI of many kinds (such as PDF, HTML, and MS Excel) in multiple locations (such as the web and a local disk).
2. Create and maintain SI of different schema in different data models, such as the relational [41] and XML (Extensible Markup Language [43]) models.
3. *Activate* BI (that is, show BI) in its original context by opening the base document in its original application and navigating to the region of interest, as well as bring the context of BI (such as enclosing text) to an application without visually activating BI.

4. Group and link SI and BI, reorganize them as needed, and maintain multiple simultaneous organizations.
5. Combine SI and the referenced BI, and select and transform the combined *bi-level information*.
6. Place references to BI in traditional documents such as MS Word documents and HTML pages.
7. Interchange SI, the references to BI, and the BI itself, with other application users.

1.2. Superimposed Information and Superimposed Applications

In this section, we introduce some terms frequently used in this research, and preview three applications (namely Sidepad, SuperMix, and the HTML+M Editor) to illustrate the range of applications that can be developed using our research.

As mentioned before, superimposed information is data placed over existing base information sources to help organize, access, connect and reuse information elements in those sources [88]. SI references BI *in situ* using an abstraction called a *mark* [32]. Information exists in two layers in this setting: SI in the *superimposed layer*, BI in the *base layer*. Figure 1.2 shows these layers of information and the use of marks as references.

The combination of SI and the referenced BI is called *bi-level information*. For example, a reader's comment superimposed on some text in a PDF document, and the commented text from the base layer, taken together, is bi-level information. Operations such as transformation (illustrated in Section 1.2.1) and interchange (described in Chapter 10) frequently involve bi-level information.

An application used to select and activate BI is a *base application* (BA). For example, Acrobat is a typical base application when interacting with a selection in a PDF document. An application used to create and view SI is a *superimposed application* (SA). An SA is like a traditional application, but with the ability to incorporate marks in SI and the ability to activate and access BI via marks.

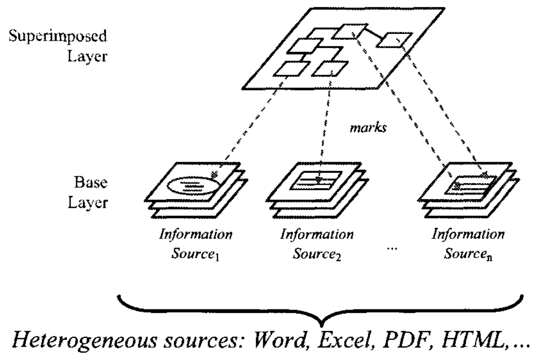
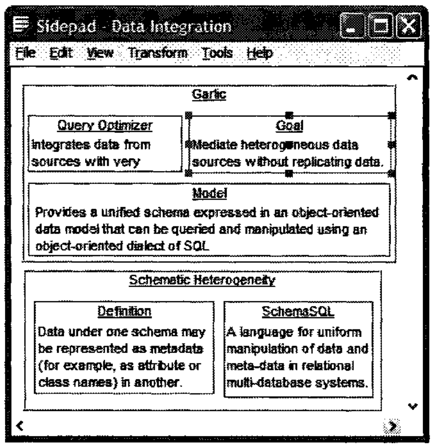


Figure 1.2: Marks referencing base information. This figure is an adaptation of a figure originally presented by Delcambre and others [32], and is shown here with permission [33]

1.2.1. Sidepad

Figure 1.3 shows an SI document (called Data Integration) created using an SA called *Sidepad* [111]. It shows information selections and annotations related to the topic of information integration. The document shown contains five *items*: Query Optimizer, Goal, Model, Definition and SchemaSQL. These items are associated with three distinct base documents of two kinds—PDF and HTML. A Sidepad item has a name, a descriptive text, and a mark (not apparent in the figure). For example, the item labeled

Goal contains a mark into a PDF document. Garlic and Schematic Heterogeneity are *groups*, which are named collections of items and other groups.


Sidepad - Data Integration
File Edit View Transform Tools Help

Garlic

Query Optimizer
Integrates data from sources with very

Goal
Mediate heterogeneous data sources without replicating data.

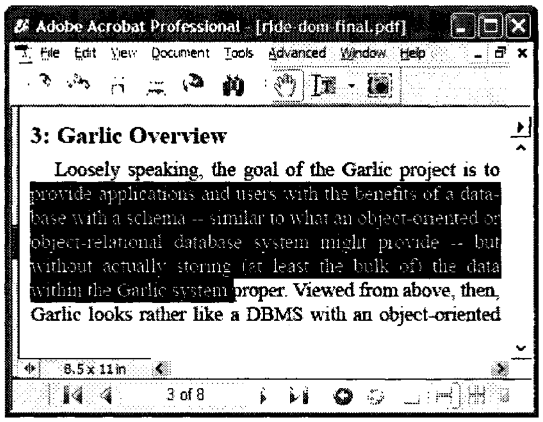
Model
Provides a unified schema expressed in an object-oriented data model that can be queried and manipulated using an object-oriented dialect of SQL

Schematic Heterogeneity

Definition
Data under one schema may be represented as metadata (for example, as attribute or class names) in another.

SchemaSQL
A language for uniform manipulation of data and meta-data in relational multi-database systems.

Figure 1.3: A Sidepad document instance


Adobe Acrobat Professional - [ride-dom-final.pdf]
File Edit View Document Tools Advanced Window Help
8.5 x 11 in
3 of 8

3: Garlic Overview

Loosely speaking, the goal of the Garlic project is to provide applications and users with the benefits of a database with a schema -- similar to what an object-oriented or object-relational database system might provide -- but without actually storing (at least the bulk of) the data within the Garlic system proper. Viewed from above, then, Garlic looks rather like a DBMS with an object-oriented

Figure 1.4: A PDF mark activated

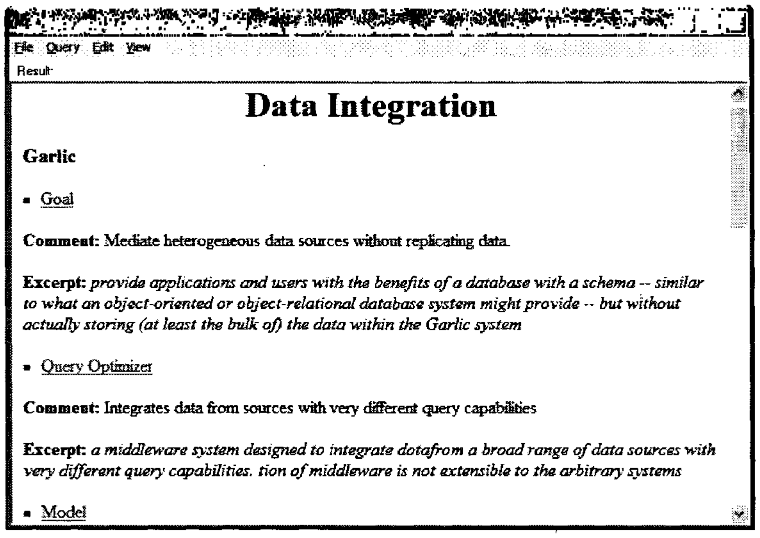
In addition to manipulating items and groups, a user can *activate* a mark (that is, see BI in its original context), and browse *context information* such as *excerpt* (that is, the content of the marked region) and the containing paragraph from within Sidepad. Figure 1.4 shows the result of activating the PDF mark associated with the item Goal. Table 1.1 shows some context information retrieved from this mark.

Table 1.1: Example context information that may be retrieved from a mark. The information shown corresponds to the PDF mark of Figure 1.4

Information Kind	Name	Value
Content	Text excerpt	provide applications and users with … Garlic system
Placement	Page number	3
Presentation	Font name	Times New Roman
Containment	Enclosing paragraph	Loosely speaking, the goal …
Containment	Section heading	3: Garlic Overview

A Sidepad document may be combined with the BI it references, and the combined *bi-level information* may be transformed to an alternative representation such as a draft paper, a table of contents, or a timeline (when temporal data is involved). Figure 1.5 shows the Sidepad document of Figure 1.3 transformed to a draft paper in HTML format. Each bullet indicates an item name. The text labeled Comment, located underneath a bullet, is the item's descriptive text; the text labeled Excerpt is the text excerpt retrieved from the mark associated with the item. The URL (Uniform Resource Locator [14]) attached to the bulleted text (and denoted by an underline) may be used to activate the mark associated with the item, just as one would from within Sidepad.

Sidepad uses the application programming interface (API) of *System S* (an implementation of the run-time services we have defined) to create marks, activate marks, retrieve context information from marks, and to transform documents. Sidepad implements the abstractions *item* and *group* itself, and it provides the necessary user interface (UI) to manipulate items and groups.



Data Integration

Garlic

- Goal

Comment: Mediate heterogeneous data sources without replicating data.

Excerpt: *provide applications and users with the benefits of a database with a schema -- similar to what an object-oriented or object-relational database system might provide -- but without actually storing (at least the bulk of) the data within the Garlic system*

- Query Optimizer

Comment: Integrates data from sources with very different query capabilities

Excerpt: *a middleware system designed to integrate dotafrom a broad range of data sources with very different query capabilities. tion of middleware is not extensible to the arbitrary systems*

- Model

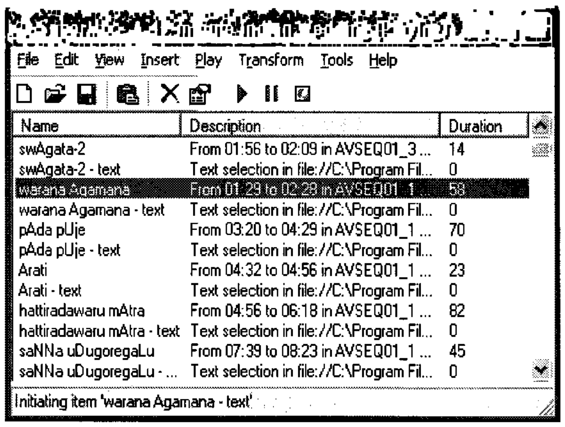
Figure 1.5: A Sidepad document and selected base information transformed to HTML

1.2.2. SuperMix

SuperMix is an SA to compose and play multimedia presentations. A SuperMix *composition* is a sequence of *cohorts*, where a cohort is a set of *members*. A member has a name, a description, and is optionally associated with a mark. Depending on the associated base type, a member can also have *duration*. For example, a video clip has duration, whereas an HTML selection does not. Duration information is obtained from the context of the associated mark, if that information exists for that mark. "Playing" a member shows the corresponding marked content (for example, a video clip) in a specific area inside a "player" window.

Figure 1.6 shows a SuperMix composition of video clips (and their text descriptions) from an Indian wedding. Each row shows a member and each cohort has two members: a video and a text. The column Name contains phrases in the South-Indian language *Kannada*, written in Roman script using a transliteration scheme [79]. The highlighted member is associated with a mark to a video clip of duration 58 seconds. The next member is associated with a text selection in an HTML file. The value 0 in the second column (in the figure) indicates that duration does not apply to this member.

A cohort's members are presented simultaneously, whereas cohorts themselves are presented sequentially. A cohort's presentation is complete when its first member is "completely" presented, or when all its members are completely presented. (We omit the details, but this behavior is configurable.) All cohorts in the composition shown in Figure 1.6 have exactly two members: a video clip and a text description. In this case, a cohort's presentation is deemed complete when its first member (the video) is completely presented. Thus, playing this composition plays a series of video clips, and presents a text annotation with each clip.



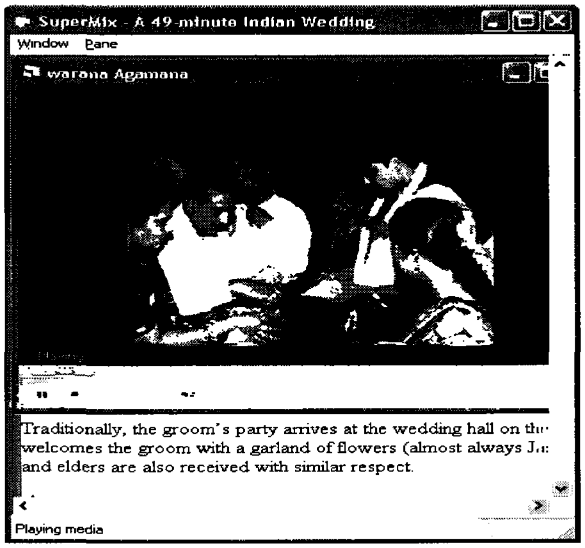
Name	Description	Duration
swAgata-2	From 01:56 to 02:09 in AVSEQ01_3 ...	14
swAgata-2 - text	Text selection in file://C:\Program Fil...	0
warana Agamana	From 01:29 to 02:28 in AVSEQ01_1 ...	58
warana Agamana - text	Text selection in file://C:\Program Fil...	0
pAda pUje	From 03:20 to 04:29 in AVSEQ01_1 ...	70
pAda pUje - text	Text selection in file://C:\Program Fil...	0
Arati	From 04:32 to 04:56 in AVSEQ01_1 ...	23
Arati - text	Text selection in file://C:\Program Fil...	0
hattiradawaru mAtra	From 04:56 to 06:18 in AVSEQ01_1 ...	82
hattiradawaru mAtra - text	Text selection in file://C:\Program Fil...	0
saNNa uDugoregaLu	From 07:39 to 08:23 in AVSEQ01_1 ...	45
saNNa uDugoregaLu - ...	Text selection in file://C:\Program Fil...	0

Initiating item 'warana Agamana - text'

Figure 1.6: A SuperMix composition

When a cohort is presented, each of its members is presented in a separate *pane*. Figure 1.7 shows the cohort corresponding to the member highlighted in Figure 1.6 being played. A video-clip is playing in the top pane of this figure; a description of this clip is displayed in a pane below the clip.

To help appreciate the utility of SuperMix, we provide some details of the composition in Figure 1.6. The composition provides a 49-minute overview of an Indian wedding that took place over a 30-hour period. The video recording of the wedding is about 163 minutes long and is available on three CDs. The notes on the various rituals in a wedding are in a single HTML document (that we created). Here are some statistics related to the composition: 88 cohorts, 176 members, 172 marks, four base documents (three video files and one HTML document), and two base applications (MS Windows Media Player [103] and MS Internet Explorer [95]).



Traditionally, the groom's party arrives at the wedding hall on th
welcomes the groom with a garland of flowers (almost always J
and elders are also received with similar respect.

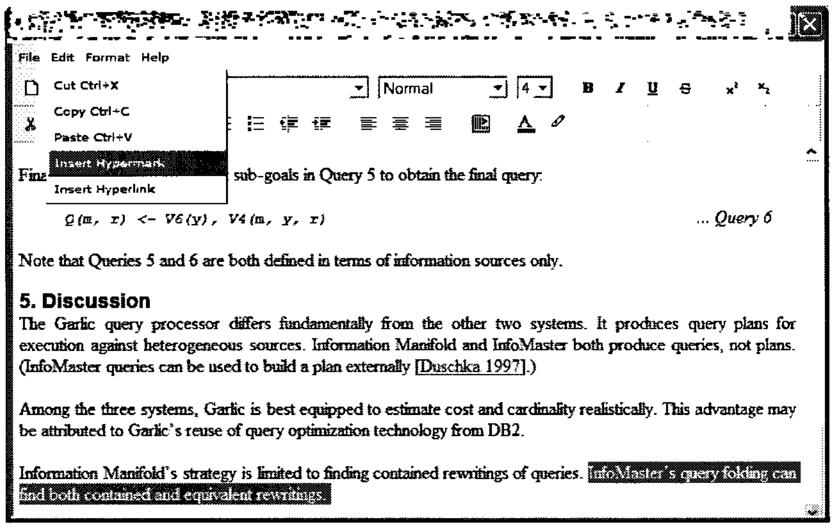
Figure 1.7: A SuperMix cohort playing. The video in the top pane corresponds to the highlighted member in Figure 1.6. The text in the bottom pane corresponds to the member immediately after the highlighted member. The video is courtesy of Gopalakrishna, and is reproduced here with permission [54]

The composition in Figure 1.6 does not present video clips in the exact chronological order of their recording, instead it groups them by rituals, without contradicting the tradition (of an Indian wedding) much. That is, it creates an *alternative organization* of base information. For example, the highlighted member in Figure 1.6 refers to a clip from the *first* CD. (The corresponding ritual took place at the beginning of the wedding.) The video member just before the highlighted member (named swAgata-2) references a clip from the *third* CD (the corresponding ritual took place after the wedding ceremony), but that member plays *before* the highlighted member.

SuperMix uses the API exposed by System S to create and activate marks, retrieve context information from marks, and transform compositions. SuperMix implements the abstraction *composition*, and provides the UI to create and play compositions.

1.2.3. The HTML+M Editor

The *HTML+M Editor* is a word-processor-style application that allows traditional hyperlinks and *hypermarks* (which are marks represented as traditional hyperlinks) in a document. The SI is saved in HTML format and may be viewed in any HTML browser. The HTML+M Editor is based on the “HTML Editor” sample application available in the MS Developer Network Library archive [108].



Fina sub-goals in Query 5 to obtain the final query:

```
Q(m, r) <- V6(y), V4(m, y, r)                    ... Query 6
```

Note that Queries 5 and 6 are both defined in terms of information sources only.

5. Discussion

The Garlic query processor differs fundamentally from the other two systems. It produces query plans for execution against heterogeneous sources. Information Manifold and InfoMaster both produce queries, not plans. (InfoMaster queries can be used to build a plan externally [Duschka 1997].)

Among the three systems, Garlic is best equipped to estimate cost and cardinality realistically. This advantage may be attributed to Garlic’s reuse of query optimization technology from DB2.

Information Manifold’s strategy is limited to finding contained rewritings of queries. InfoMaster’s query folding can find both contained and equivalent rewritings.

Figure 1.8: An HTML+M document being edited

Figure 1.8 shows a survey paper being edited in the HTML+M Editor. It shows the user associating a hypermark with a selection. The user has already associated a hypermark with another region of the survey paper—the last two words in the first paragraph of Section 5 (the citation “Duschka 1997”)—as indicated by an underline. When this document is viewed in an HTML browser, clicking on a region that contains a hypermark activates the BI selection that the hypermark represents.

The HTML+M Editor uses the API of System S to insert hypermarks into documents. The same API is also used when a hypermark is activated inside an HTML browser. The HTML+M Editor implements the word-processing features.

1.3. The Research Objective

Our research objective was to develop a comprehensive and generic *framework* that supports the design, development, and deployment of *any* SA that supports any subset of the application capabilities listed in Section 1.1. In this section, we briefly describe the rationale for this research objective, and summarize the considerations and features of the framework developed. We begin with the rationale for the research objective.

Different SAs are likely to be developed for different goals, just as different traditional applications have been developed for different goals. For example, one may use MS Word to write papers, but use MS PowerPoint [96] to prepare presentations. These applications have different information models, support different use cases, and employ different UIs. Similarly, the SAs Sidepad, SuperMix, and the HTML+M Editor have different SI models, support different use cases, and employ different UIs.

Regardless of the differences in the SI and UI models they employ, all SAs afford some common capabilities to their users: associate SI with marks; activate marks; retrieve context information from marks; and others. We believe that reusable run-time services can be developed to support these and the other application capabilities listed in Section 1.1. Such services alleviate the need for an SA developer to implement the common capabilities. Instead, the developer can focus on SA-specific features.

As with the application capabilities, some of the development activities will also be similar for many SAs. Reusable solutions can support such activities as well. Table 1.2 shows some common SA-development activities. Activities with dark shading indicate significant scope for reuse. Lightly shaded activities have some scope for reuse. Activities not shaded have little or no scope for reuse (because they tend to be SA-specific).

Our framework for SAs is a result of recognizing the aforementioned commonalities. Using our framework, developers can support marks to new kinds of base information, using any appropriate addressing scheme. Also within the framework, an SA can reference any supported BI type, regardless of the location of the BI and references to it by other SAs. The framework, the SAs, and the base applications can all evolve independently with minimal adverse impact on each other.

Table 1.2: Common SA design and development activities

Phase	Activity	Remarks
Design	Design SI	Design conceptual, logical, and physical schemas. For example, an Entity-Relationship schema [25], relational schema, and a physical relational database respectively.
	Design UI	For example, the Sidepad UI (in Figure 1.3).
Implementation	Implement UI	In addition to the UI for the SA, implement viewers such as those needed to display context information.
	Implement SI-layer operations	For example, create and group items in Sidepad; persist Sidepad documents.
	Implement mark creation, activation, and context retrieval	For example, mark a region in a PDF document, activate it (as in Figure 1.4), and retrieve its text excerpt.
Deployment	Configure	Decide location and number of run-time components.
	Deploy	Install and run components at selected locations.

Our framework includes methodologies to represent bi-level information in different data models; a set of run-time services to represent, access, transform, and interchange bi-level information; and a set of guidelines [112] to deploy the run-time services and the SAs that use the services. For brevity, this dissertation omits the description of the deployment guidelines.

Our framework supports SI over both documents and sub-documents, but, in this dissertation, we mainly discuss SI over sub-documents because operations on sub-documents present some unique challenges. We call out an operation on documents if it is substantially different from an equivalent sub-document operation.

We made the following considerations in developing our framework:

- We need to work with arbitrary SI schemas because we do not know the exact information needs of the SAs that might use our framework.
- We need to work with distributed, heterogeneous BI *in situ*. Thus, in general, we cannot modify or move BI, and, in some cases, we might even be prevented from copying it. We cannot preprocess BI (for example, load BI into a database) because, in general, the collection of information SAs reference is not known in advance.
- We need to support the use of base applications with varying capabilities. For example, we need to work with base applications (such as MS Excel) that support information structuring, as well as with applications (such as a text editor) with little or no such capability.

- We need to support arbitrary (but reasonable) deployment configurations for SAs. For example, we need to support the deployment of an SA and our framework on desktops. We also need to support an SA that uses our framework in a client-server setting.

1.4. Related Work

Applications and technologies that support some subset of the seven capabilities listed in Section 1.1 exist, but none supports the complete set of capabilities.

Acrobat [8] and MS Word [96] support inline annotations. They do not support multiple, simultaneous organizations of annotations, and they fix the annotation structure. For example, MS Word can display annotations (called *comments*) only as lists, and an annotation contains a user name, annotation date, and an annotation text. In both applications, annotations are shared by sharing the annotated documents.

Some hypertext systems allow creation of stand-off annotations, and help maintain multiple organizations of the same information. However, they tend to constrain the types of source, granularity of information, the location of information consulted, or the presentation model. For example, NoteCards [56] requires information consulted to be in a specific format, stored in a proprietary database, and allows references only to documents (called *cards*), not to sub-documents. It also fixes the presentation model for hypertext networks (that is, it fixes the user interface). IRIS [55] supports references to documents and sub-documents located anywhere in the file system, but requires specially constructed base applications. It allows creation of multiple hypertext networks called *webs*, but a user can work with only one web at a time. It also fixes the UI for a web. The Dexter Hypertext Reference Model [57] allows references to documents (called *components*) and sub-documents (called *anchors*) of any type. It stores descriptions of components, anchors, and links in a *storage layer*. A *run-time layer*, which is not part of Dexter, displays a hypertext network. None of these systems (NoteCards, IRIS, and Dexter) can retrieve the excerpt of a base selection for use out of context.

Systems such as OLE 2 [18] and OpenDoc [132] facilitate the creation of *compound documents* that can contain references to information in other documents. They allow annotations over documents and sub-documents, but they provide the user little control over the kind of information that can be retrieved from a referenced document. Annotations are shared by sharing a compound document, but participating users must follow a convention for the location of referenced documents. For example, they must store the referenced documents in the same folder as the compound document.

Modern HTML browsers can navigate to practically any kind of information using *handlers* (which are pieces of executable software), but they limit the kinds of data that can be incorporated within a document (in comparison to compound-document systems). Natively, browsers support references only to sub-documents the author has marked (using appropriate markup tags). That is, a user reading a document cannot create references to arbitrary portions of a document without modifying the document.

Of the systems mentioned in this section, compound document systems provide the best support for multiple, simultaneous organizations of annotations. Compound document systems, IRIS, and Dexter facilitate development of multiple applications to create and maintain stand-off annotations. However, none of these systems have the ability to retrieve information such as the "paragraph that contains the referenced sub-document" from the context of a sub-document. Also, none of these systems readily supports querying a mixture of annotations and the annotated sub-documents.

1.5. Organization

This dissertation is composed of 11 chapters, including this chapter. Chapter 2 provides a summary of this research, including an overview of the contributions, and an introduction to the various components of our framework.

Chapters 3 through 10 describe the contributions of this research. Chapter 3 describes *SPARCE* (the Superimposed Pluggable Architecture for Contexts and Excerpts) [110], and shows its use to create marks, activate marks, and access context information. It explains how SPARCE may be extended (for example, to add support for new types of base information) without affecting existing components. An evaluation of an implementation of SPARCE is included.

Chapter 4 defines a methodology to conceptually model bi-level information in the Entity-Relationship (ER) model [25], and shows how a resulting conceptual schema enables queries over bi-level information [113]. The methodology includes procedures to translate a conceptual bi-level-information schema to a logical schema in either the relational or the XML model. The methodology is evaluated by using it to prepare both conceptual and logical schemas for three SAs. Chapter 4 also introduces *Sixml* (pronounced `'siks-m&l`) [120], a means of representing SI as XML: A *Sixml document* is an XML document that represents bi-level information using our methodology.

Chapters 5 through 9 describe how bi-level information may be selectively transformed using queries in existing query languages. Chapter 5 explores the key issues in representing and querying bi-level information in the XML model, and outlines our goals and strategies to transform bi-level information.

Chapter 6 describes a means of efficiently retrieving context information from a large number of marks using a *bulk accessor* [121], and illustrates the use of the bulk accessor in an existing relational query processor. It also presents the results of an experimental evaluation of the bulk accessor.

Chapter 7 describes *Sixml DOM* [120], an object model to manipulate a Sixml document and the bi-level information derived from the Sixml document. It also describes implementation strategies and summarizes the results of experiments with different implementations.

Chapter 8 formally presents a means of *cloaking* (that is, hiding) data to improve the performance of certain classes of queries over bi-level information.

Chapter 9 builds on the developments in Chapters 6 through 8. It shows how a *bi-level navigator* (called the *Sixml Navigator*) [120] can be used in existing query processors to evaluate queries over XML bi-level information. An experimental evaluation of the bi-level navigator is also presented.

Chapter 10 describes a means of interchanging bi-level information among SA users. It introduces the notion of *SI-dependency graphs*, and shows how one of these graphs can be used to package bi-level information for interchange.

Chapter 11 summarizes this dissertation and presents concluding remarks. It also outlines two future applications for this research.

2. Research Summary

This chapter describes the major contributions of this research; introduces the components of our framework to assist in the design, development, and deployment of superimposed applications (SAs); gives an overview of the evaluation of the framework; and compares the features of a reference implementation of the framework with related systems.

As mentioned in Section 1.3, we support SI over both documents and sub-documents, but, for simplicity, we focus the discussion on SI over sub-documents. We call out a situation involving entire base documents if it is substantially different from a similar situation involving sub-documents.

2.1. Contributions

The major contributions of this research are:

1. The concept of *context information* for sub-documents, documents, and applications (collectively called *base parts*) that reside in the base layer.

2. The concept of *bi-level information*, which is a combination of superimposed information (SI), the base-part references, and the context information obtained from the referenced base parts.

3. Techniques to represent, access, transform, and interchange bi-level information, and an evaluation of these techniques. The techniques include:

    3.1. A system of representing context information as *hierarchical property sets*.

3.2. A methodology to define *conceptual schemas* over bi-level information, and procedures to translate conceptual schemas to logical schemas in the relational [41] and the XML [43] models.

3.3. The abstraction *context agent* with an associated application-programming interface (API) to support retrieval of context information from arbitrary base parts.

3.4. A generic means of accessing bi-level information in the relational and XML models.

3.5. An architectural component called a *bulk accessor* to efficiently retrieve context information from a large number of base parts.

3.6. The notion of *bi-level queries* to transform bi-level information using queries expressed in existing query languages.

3.7. The notion of an XML *bi-level navigator* and its use in existing XML query processors to execute bi-level queries without modifying the processors or the query languages.

3.8. A means of selectively *cloaking* (that is, hiding) parts of data to improve the expression and execution of certain classes of queries, and the application of cloaking to querying bi-level information. For example, it is possible to hide context information so that queries operating only on SI are more easily expressed and are more efficiently executed.

3.9. The notion of *SI-dependency graphs* to denote SI and the information on which SI depends, and the use of an SI dependency graph to package bi-level information for interchange among SA users.

4. The design of *Superimposed Application Shareable Services* (SASS, a set of run-time services to realize the techniques listed in Contribution 3), including architectural desiderata, an architectural reference model, and a reference implementation.

5. A set of *deployment guidelines* for SAs and the components of SASS.

In the rest of this section, we review each of the major research contributions. Section 2.3 gives an overview of our approach to evaluate the contributions.

2.1.1. Context Information and Bi-level Information

Our first contribution is the ability to *uniformly* reference *base parts* (that is, sub-documents, documents, and applications in the base layer) of *arbitrary types*; and the concept of *context information* (or just *context* for short) for such parts. *Context information* is the set of information that can be obtained from a base part [110]. An *excerpt* (that is, the content of a base part) is one kind of context information. "Containing paragraph" and "font name" are other kinds of context information.

The second contribution is the concept of *bi-level information* [113], which is SI combined with the context information for the referenced base parts. This integrated access to bi-level information allows SA developers and users to produce useful artifacts and to provide useful services. For example, a user might transform a Sidepad document to an HTML document [61] (such as that shown in Figure 1.5) containing descriptions of

Sidepad items and the text excerpts of the referenced sub-documents. Access to bi-level information also enables analytical tasks such as finding the base documents on which an SI document depends. Such analysis is necessary to interchange an SI document (among SA users).

2.1.2. Representing, Accessing, Transforming, and Interchanging Bi-level Information

Our third contribution is a collection of techniques to support the following common activities in relation to bi-level information: representation, access, transformation, and interchange. These techniques support a developer in both designing and developing an SA. They also assist SA users at run time.

This section reviews our support for the aforementioned activities, and the different data models we especially consider in this dissertation for these activities. Table 2.1 lists the activities and the data models.

Table 2.1: Activities on bi-level information and the data models emphasized. A number in parentheses denotes the chapter where a combination of activity and data model is considered.

Activity	Data models for SI	Data models for Context Information
Conceptual representation	ER (4)	ER (4)
Logical representation	XML (4, 7); Relational (4); Other (3)	XML (4); Property set (3)
Access	XML (7); Relational (6); Other (3)	XML (7); Relational (6); Property set (3)
Transformation	XML (9); Relational (4)	XML (9); Relational (4)
Interchange	Any (10)	Not applicable

2.1.2.1. Representing Bi-level Information

Representing bi-level information involves representation of SI, the *marks* (which are references to base sub-documents) the SI employs, and the context information retrieved from the marks. (For simplicity, we limit this discussion to sub-document references, but we also support references to base documents and applications.) We allow an SA developer to represent SI in any data model (such as the relational or XML models) with a schema appropriate to the SA. For example, each of the three SAs presented in Section 1.2 uses a distinct data model to represent its SI.

We define three representations for a mark: encoded string, XML fragment, and Uniform Resource Identifier (URI) [15]. An SA may store a mark (in any of the three forms) entirely within its SI, or store only a unique identifier (ID) string we assign the mark. A mark is assigned an ID when it is stored in a *mark repository*, a collection of marks managed by a service we define. These choices allow marks to be represented in data models that support string values, XML fragments, and URIs. Because XML fragments and URIs can be represented as strings, those representations can be used in any data model that supports strings; and almost every modern data model supports strings. Chapter 3 describes mark representations and mark repositories.

When representing bi-level information, we do not represent a complete base document if only its sub-document is referenced. For example, if a Sidepad document references a PDF [6] sub-document, we do not model the containing PDF document. Instead, we model only the context information (including the excerpt) retrieved from the referenced sub-document.

We model context information for each mark as a *hierarchical property set* in which each kind of information element retrieved forms a part of the hierarchy, and each information element retrieved has a name and a value. (See Table 1.1 for an example.) A hierarchical property set provides a uniform representation for context information regardless of the base-document type, and forms the basis for representation of marks for other activities. Chapter 3 describes the representation of context information.

To support designing SI, we provide a means of modeling SI, the marks, and context information, in both conceptual and logical data models [113]. For conceptual modeling, we extend the Entity-Relationship (ER) model [25] with the notion of *relationship patterns* (that is, recurring relationships) [114] to associate any number of marks with SI entities, relationships, and attributes (of both entities and relationships). We also define procedures to translate conceptual ER schemas to logical schemas in the relational and XML models. Chapter 4 describes this contribution.

We have also developed methodologies to represent marks and context information in the relational and XML models, independent of the conceptual modeling solution. We have paid special attention to the association of marks with different XML constructs (such as elements, attributes, and text content). To this end, we have defined *Sixml* (SI represented as XML, pronounced `'siks-m&l`) [118, 120], a means of expressing marks using only the constructs available in XML Schema [170]. We also define a procedure to serialize (that is, write) Sixml data using only the syntax to serialize traditional XML documents [43]. Chapter 7 describes Sixml and its serialization.

2.1.2.2. Accessing Bi-level Information

Access to bi-level information requires access to SI, the marks, and the context information retrieved from the marks. In general, the SA developer is responsible for providing access to SI because we do not know the organization of SI, *a priori*. The SA developer is also responsible for access to marks that are embedded in SI, but we define an API to store and access marks in mark repositories we manage. We also define an API to activate marks regardless of where they are stored.

We define an abstraction called *context agent* to extract context information from base parts. A developer can implement the API we define for this abstraction to retrieve context information for any base type.

The mark-management and context-management APIs are implemented in *SPARCE*, the Superimposed Architecture for Contexts and Excerpts [110], our middleware for mark and context management. Chapter 3 describes SPARCE and illustrates its use to access context information from base parts of a variety of types including PDF, HTML, and Microsoft (MS) Word [96]. (SPARCE relates to the Contributions 3.1 and 3.3, and is a part of SASS called out in Contribution 4.)

Some access patterns (such as those involved in transforming bi-level information) of SAs might retrieve context information from a large number of marks. We define a component called the *bulk accessor* [121] to support such access patterns. The SA developer can configure the bulk accessor to exploit data characteristics such as the number of marks and base sources, and the sequence of mark access. Chapter 6 describes the bulk accessor, illustrates its integration into an existing relational query processor, and presents the results of an experimental evaluation.

In Section 2.1.2.1, we introduced our methodologies to represent bi-level information in the relational and XML models. In many cases, bi-level information in the relational model can be easily manipulated using existing mechanisms (such as user-defined functions [147]), but the same is not true for the XML model. To make manipulation of Sixml data (which is SI represented as XML) easier, we define *Sixml DOM* [120], an extension of the XML Document Object Model (DOM) [34]. Sixml DOM makes marks first-class objects in DOM; accommodates both marks embedded in SI and references to marks in repositories via IDs; and retrieves context information *on the fly* (using the bulk accessor). Chapter 7 describes Sixml DOM, reviews alternative implementations, and presents the results of an experimental evaluation.

2.1.2.3. Transforming Bi-level Information

An SA developer can retrieve marks and context information for the marks using our APIs, *explicitly* combine the retrieved information with SI, and transform the combined bi-level information to new forms (such as an HTML table of contents). Carrying out these tasks using *imperative* programming languages requires much development effort, and it can create dependence on specific programming platforms. (Chapter 5 illustrates these issues.)

As an alternative, we define a means of *implicitly* preparing bi-level information and *declaratively* transforming it. We accomplish these tasks by representing context information for the referenced base parts in the same data model as SI, or by representing both context information and SI in another data model (such as the relational and XML models). To prepare bi-level information, an SA supplies only the SI and the associated marks to a *transformation service* we define. The service expands its input to include the context information appropriate to the transformation, and executes the requested transformation. The SA or the SA user describes the transformation to be executed using a *bi-level query* in an existing query language such as SQL [92] or XPath [166].

We do not require that a transformation service support all data models. We allow several transformation services, each service possibly supporting a specific data model. We do not fix a query language to express transformations, but naturally expect that a language appropriate for the data model of the SI is employed. For example, SQL might be the query language if SI is in the relational model, but the language might be XPath or XSLT [177] if SI is in the XML model. We also do not fix a strategy (such as the order of retrieving information parts or the order of evaluating query parts) to execute bi-level queries, because the right strategy depends on factors such as the data model, the representation scheme, and the query language.

We demonstrate the ability to execute bi-level queries in the XML and relational models. In the relational model, we represent bi-level information using the representation scheme we alluded to in Section 2.1.2.1, express bi-level queries using standard SQL, and execute the queries using existing query processors. Chapters 4 and 6 provide the details.

In the XML model, we define a *bi-level navigator* [120] to expose bi-level information in the XPath data model so that bi-level information can be queried using *existing* XML query languages and query processors. The bi-level navigator accepts a Sixml document as input and uses Sixml DOM to prepare bi-level information. Chapter 9 describes this navigator and illustrates its use in existing XPath and XSLT processors. The chapter also presents the results of an experimental evaluation using two representation schemes for Sixml documents. Chapter 5 introduces the alternative representation schemes.

Finally, certain classes of bi-level queries (for example, queries that examine and return only SI) can be harder to express and they might execute poorly in a bi-level information setting. To improve this situation, we define a means for selectively *cloaking* (that is, hiding) parts of data to a query processor. Chapter 8 formally describes cloaking and shows its application to bi-level query processing. Chapter 9 describes how the bi-level navigator implements cloaking, and presents experimental results that illustrate the benefits of cloaking.

2.1.2.4. Interchanging Bi-level Information

To interchange bi-level information, we model SI, the associated marks, and the referenced base documents as an *SI-dependency graph* (which is a directed acyclic graph), and use the graph to *package* bi-level information for interchange among SA users. We also define a process to unpack a received package and allow the receiving user to freely choose the location of SI and base documents extracted from the package. Chapter 10 describes SI-dependency graphs and their use in interchanging SI.

2.1.3. Superimposed Application Shareable Services

Our fourth contribution is a design for *Superimposed Application Shareable Services* (SASS, pronounced `'sas`), which is a set of reusable *runtime services* (that is, services available at SA execution time) to access, transform, and interchange bi-level information.

We have designed SASS with the following architectural qualities in mind. (Bass and others [13] provide an overview of qualities of software architectures.)

- Functionality: The implementation must provide runtime services that are helpful in implementing the seven application capabilities listed in Section 1.1.
- Reusability: Many SAs must be able to use the same SASS implementation. More than one SA instance must be able to run simultaneously on the same computer, and each instance must be able to interact with multiple base documents.
- Modifiability: It must be possible to independently improve SASS and the SAs, with minimal adverse impact on each other.
- Extensibility: It must be possible to support new base types and context elements without affecting existing SAs and context agents.
- Package flexibility: It must be possible to change the location of the components of SASS to meet application and user needs. For example, we must be able to dep-

loy the components of SASS on the same machine as the SA, or on a different machine. (This quality is related to deployment of SASS. See Section 2.1.4.)

- Testability: The SASS implementation must aid verification and validation of itself, and of the SAs that use it.
- Usability: The SASS implementation must use familiar metaphors, and follow relevant development and UI conventions. It must also aid usability of SAs developed using it.

We have also defined a *reference model* (that is, a conceptual layout of the components) for implementations of SASS, and used the reference model to implement a prototype SASS called *System S*. Section 2.2 includes an overview of SASS and the reference model.

2.1.4. Deployment Guidelines

The fifth contribution is a set of guidelines to deploy SAs and the components of SASS [112]. For brevity, we do not present the guidelines in this dissertation, but summarize here the motivation to develop the guidelines. We also provide an outline of the guidelines.

Component-based systems (such as SASS) allow new components to be plugged in easily, and allow existing components to be easily replaced. They also offer flexibility of deployment of the components involved. With proper interface design and abstraction, components (both data and executable) can be either centrally deployed or distributed, without affecting the services provided. This flexibility is important because placing a component at the right location can improve performance, especially for frequently used services.

In this vein, the overall performance of an SA and SASS can be improved by matching the location of executable and data components to the needs of SA users and BI providers. However, deployment configurations of SAs can vary widely. For example, one user might install the Sidepad application on a desktop computer and consult information available mostly on his local file system. Another user might use Sidepad to consult information primarily on the web. In the former case, Sidepad and SASS might be deployed on the same computer. In the latter, Sidepad and some parts of SASS might be installed on the user's computer, and other parts might run on a remote server. In contrast to these two cases, it is also possible to build a web-based SA that interacts with a SASS installation on a remote web server.

Motivated by these observations, we have developed guidelines for *five* deployment alternatives, where each alternative varies the location of SA and of the components of SASS. The guidelines define some performance metrics, which we use to explore the trade-offs in each alternative. They also discuss potential barriers for performance, and posit some means to improve performance.

2.2. Framework Overview

Figure 2.1 provides an overview of our framework to support the design, development, and deployment of SAs. The top section of the figure shows design-time support to model SI. The bottom section shows support for deployment. The shaded boxes in the middle section represent SASS.

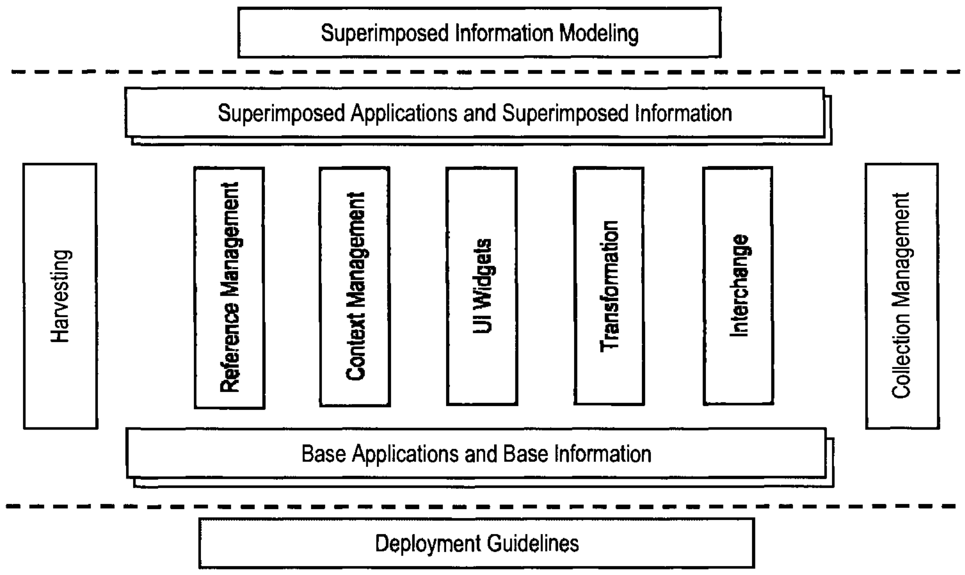

Superimposed Information Modeling
Superimposed Applications and Superimposed Information
Harvesting
Reference Management
Context Management
UI Widgets
Transformation
Interchange
Collection Management
Base Applications and Base Information
Deployment Guidelines

Figure 2.1: A framework to support design, development, and deployment of SAs

The service Reference Management in Figure 2.1 supports creation and retrieval of marks. (This service's name captures our framework's ability to support references to base sub-documents, documents, and applications.) Context Management supports activation of marks and retrieval of context information from marks. UI Widgets provides UI tools (such as a viewer to browse context information) that multiple SAs may share. Transformation and Interchange support transformation and interchange of bi-level information, respectively.

The boxes labeled Harvesting and Collection Management in Figure 2.1 are not part of SASS, instead they build on SASS. *Harvesting* refers to the programmatic generation of marks (as opposed to manual marking). For example, a script might mark the citations in a research paper. The needs and means of harvesting vary among applications and tasks, but harvesters can build on the Reference Management service.

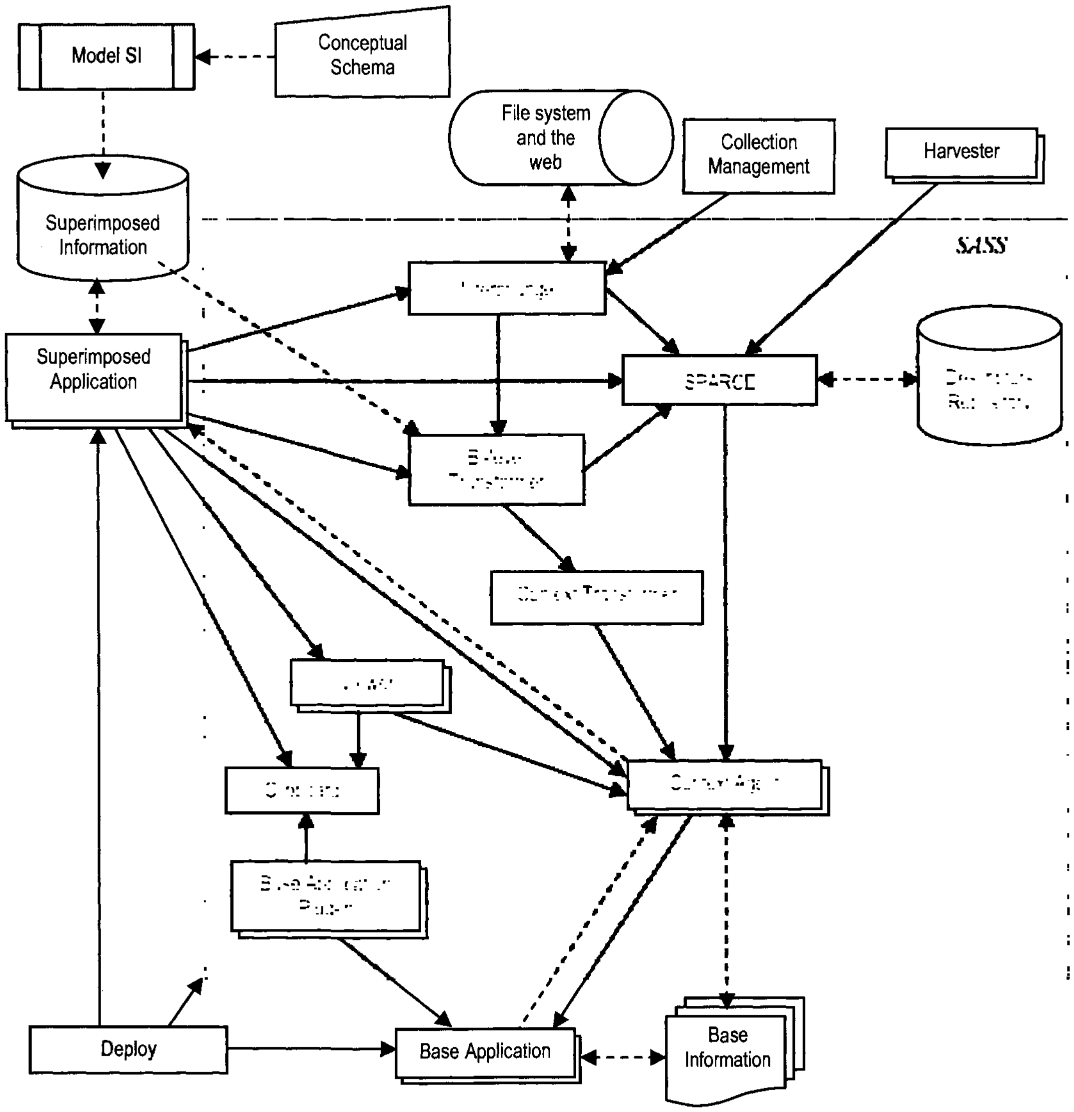


Figure 2.2: A reference model for the framework to support design, development, and deployment of SAs. Solid arrows show control dependency, dashed arrows show data flow

Collection management refers to the management of a set of SI documents along with the marks they reference, and possibly the base documents to which the marks correspond. For example, one might manage a collection of Sidepad documents and the referenced base documents in a digital library. Collection management can reuse parts of the Interchange service.

Figure 2.2 shows a reference model for our framework. The solid arrows in the figure denote control (or code) dependencies. The dashed arrows indicate data flow. The box labeled Model SI indicates our methodologies to conceptually (and logically) model bi-level information. The box labeled Deploy refers to post-implementation activities related to deploying SASS and the SAs.

The area shaded dark represents SASS. The boxes Bi-level Transformer and Context Transformer together allow SA developers and SA users to manipulate bi-level information. The box Interchange provides a means to interchange bi-level information. The box Viewer represents UI widgets. The other boxes together indicate mark-management and context-management services.

2.3. Evaluation Overview

In this section, we provide an overview of the evaluation method for the different components of our framework.

We have evaluated our methodologies to conceptually and logically represent bi-level information by using the methodologies in three SAs: Sidepad, the Superimposed System Information Browser (SSIB) [113], and the Superimposed Scholarly Review System (SISRS) [109]. Section 4.9 presents the evaluation details.

We have experimentally evaluated the performance of the bulk accessor, Sixml DOM, and the bi-level navigator using datasets containing between a few thousand marks and over 100,000 marks. Chapters 6, 7, and 9 describe the experiments.

We have evaluated the design of SASS and validated its architectural qualities by creating a reference implementation called *System S* using a combination of the .NET [129] and ActiveX technologies [93] for the MS Windows [104] platform. We have used the extensibility mechanism in System S to support referencing base parts of the following types: MS Word, MS Excel, MS PowerPoint, PDF, XML, HTML, and a variety of audio and video formats. Chapter 3 provides the details.

To evaluate the utility of SASS, we have developed *five* SAs (Sidepad, SuperMix, HTML+M Editor, SSIB, and SISRS) using System S, and developed multiple queries over bi-level information created in these applications. We have also built a utility called *Mash-o-matic* [115] to generate a class of applications called *mash-ups*, and to generate data for mash-ups.

The following is a list of applications developed by others using our framework.

- The Superimposed Multimedia Presentation Editor and Player (SIMPEL) [123], an SA to organize multimedia content on a timeline and play the content in a synchronized manner.
- IHMC CmapTools [63], a commercial application to develop concept maps, augmented to incorporate marks [124].

- The Superimposed TRansactor for Integrating Data into Entities (STRIDE) [10], an SA designed to capture human attention when integrating data for specific tasks.
- The Guava Context Agent [153], to mark into UI controls (such as text fields and list fields) in a class of applications developed using the .NET Framework [129].

2.4. Topics Excluded

Several aspects of SA development and SI management exist that are not covered in this research, or are covered in only a limited way.

In general, we do not handle updates to base sources with existing marks, nor do we handle base sources that move. However, our framework does not preclude interactions with such sources. Chapter 3 addresses this topic.

When transforming bi-level information, we do not exploit the data-management capabilities that a base application (such as a database management system) might have. Exploiting certain base-application capabilities can help execute some transformations more efficiently, and developers might be able to express the transformations more easily (or elegantly) using those capabilities.

We do not define a specific runtime service to store SI. Delcambre and others [32] have defined a generic SI storage service called SLIMStore. We do not consider storage of BI, because we consult base information *in situ*.

We support referencing of sub-documents in a *cross-platform* manner (that is, across different operating platforms), but we do not consider cross-platform support for *all* the runtime services we define. However, we believe the design of SASS is portable to most modern operating platforms and is amenable to implementation in most modern programming languages. For example, our research partners at Villanova University (under the supervision of Professor Lillian Cassel [60]) have ported parts of the System S implementation to Java [71].

2.5. A Comparison of Related Systems

Table 2.2 shows a comparison of System S with some of the systems mentioned in Section 1.4 with respect to the runtime services our framework defines. The first two rows of Table 2.2 correspond to the service Reference Management in Figure 2.1. The third and fourth rows correspond to the service Context Management. We do not compare the systems with respect to the service UI Widgets. The terms shown in italics in the table are defined in the literature of the respective systems.

None of the related systems assists in modeling information as our framework does. Also, the literature for these systems does not address deployment issues.

This comparison shows that our research framework provides a comprehensive set of design and development tools to SA developers, and that it enables the developers to provide a rich set of services to users of their applications.

2.6. Summary

Broadly, this research examines the issues in realizing and leveraging bi-level information. It examines SAs and bi-level information from a software-engineering perspective as well as an information-engineering perspective. It defines a framework to design, develop, and deploy SAs; and presents techniques to represent, access, transform, and interchange bi-level information.

This chapter has provided a summary of the contributions, components, and evaluation of this research. Chapter 3 begins the detailed description of the research with information about representing and accessing marks and context information.

Table 2.2: A comparison of System S and some related systems. The comparison is with respect to the runtime services shown in Figure 2.1

	MS Word 2002 [96]	**MS PowerPoint 2002 [96]**	**Acrobat 7.0 [8]**	**IRIS [55]**	**Dexter [57]**	**OLE 2 Compound Documents [18]**	**System S**
Mark creation mechanism	Select a *character span* and create a *bookmark*	A *link* to a slide is automatically created when a slide is referenced	Place cursor on a page and create a *bookmark*	Create an *anchor* and perform the *Start Link* operation	Define an *anchor* within a *component*	Create a *link* in a *server application*	Create a mark in a base application
Means to incorporate marks	Insert a bookmark anywhere in the *same* document	Insert a *hyperlink* to a slide in the current or another presentation	Insert a bookmark anywhere in the *same* document	Create an anchor and perform the *Complete Link* operation	Define a *link* with anchors as *endpoints*	Place a link in a *compound document*	Associate a mark with an SI element such as a Sidepad item
Mark activation	Document *must* already be open to activate a bookmark	Activate a hyperlink	Document must already be open to activate a bookmark	Activate a link; native support to see sub-document in context	Activate a link's endpoint; *each run-time layer* is required to show sub-document in context	Activate a link; native support to see sub-document in context	Activate a mark; native support to see sub-document in context
Context retrieval	Not supported within the application, but API available	Not supported within the application, but API available	Not supported within the application, but API available	Not supported	Not supported	Retrieve *only content*, but in several formats	Retrieve content and many other kinds of context information
Transform bi-level information	View unfiltered sequence of *only* comments and revisions	View unfiltered sequence of *only* comments and revisions	View list of *only* comments; filter, sort, and group operations afforded	Not supported	Not supported	No native support; a *controller application* may support it using retrieved contents	Natively supported; may use a declarative query language
Interchange SI	Share entire document	Limited sharing using the *Pack and Go* feature [4]	Share entire document	Use a hypertext interchange format [141]; manually share referenced documents, and use a convention for file location	Use the *Dexter Interchange Format*; manually share referenced documents	Share compound document; manually share referenced documents and use a convention for file location	Natively supported; referenced documents may be freely relocated

3. Representing and Accessing Base References and Contexts

Chapter 2 introduced the notion of *Superimposed Application Shareable Services* (SASS) and reviewed its role in our framework for superimposed applications (SAs). The Superimposed Pluggable Architecture for Contexts and Excerpts (*SPARCE*) [110] is the part of SASS that supports creation of references to base sub-documents, documents, and applications (collectively called *base parts*), activation of base parts, and retrieval of context information from base parts. It is designed to satisfy the architectural requirements listed in Section 2.1.3.

This chapter describes SPARCE, provides a summary of its evaluation, and reviews related work.

3.1. Introduction

This section introduces some terms, provides an overview of SPARCE, and reviews a process of creating references to base parts.

SPARCE implements the *mark* abstraction to reference a base sub-document; the abstraction *document* to reference a base-layer entity such as a document or a database in which marks may be created; and the abstraction *application* to reference a base program used to view and access marks and documents.

In this dissertation, for simplicity (and for historic reasons), we use the term *mark* (a reference) to also mean a base sub-document (a referent). Likewise, we use the terms *document* and *application* generally to mean a referent. We disambiguate the use of these terms when the meaning is not clear from the context.

The information necessary to reference a base part is called a *descriptor*. A *mark descriptor* includes information such as the location of a sub-document within a base document. A *document descriptor* includes information such as the path to the disk file containing a base document. An *application descriptor* contains information such as the name and version of a base application.

The abstraction *context* denotes information concerning a base part. Presentation information such as font name, containment information such as enclosing paragraph, and placement information such as page number are examples of context information retrieved from a mark. File path and file size are examples of context information retrieved from a base document. Application name and publisher name are examples of context information retrieved from a base application.

Excerpt is the content (such as text and image) retrieved from a mark or a document. An application does not have an excerpt. An excerpt is a part of a base part's context.

Figure 3.1 shows a reference model for SPARCE. The module Reference Management handles operations such as creation of base-part references. Context Management is responsible for *activating* a base part (that is, showing the base part in its original context) and for retrieving context information from the base part. The Clipboard facilitates inter-process communication. The Descriptor Repository provides storage for base-part descriptors.

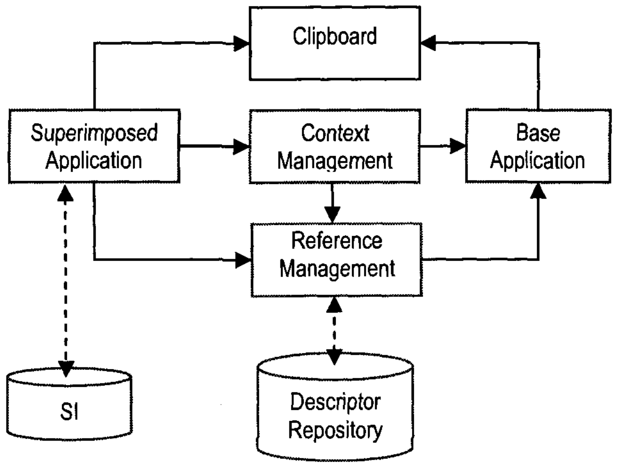


Figure 3.1: The SPARCE reference model

We now briefly describe the process of creating marks. Marks may be created interactively or programmatically. Figure 3.2 shows a user of an SA creating a mark interactively. In this case, the user first selects a sub-document within a base application—for example, a text selection in a Microsoft (MS) Word [96] document—and copies the selection to the clipboard that the operating system (OS) provides. This operation copies a mark descriptor to the clipboard. The user then "pastes" the clipboard contents into an SI document, in an SA. In response, the SA retrieves the mark descriptor from the clipboard, and associates the retrieved descriptor with an SI element (that the user chooses). For example, in the Sidepad application introduced in Section 1.2.1, the user may associate a mark descriptor with an item.

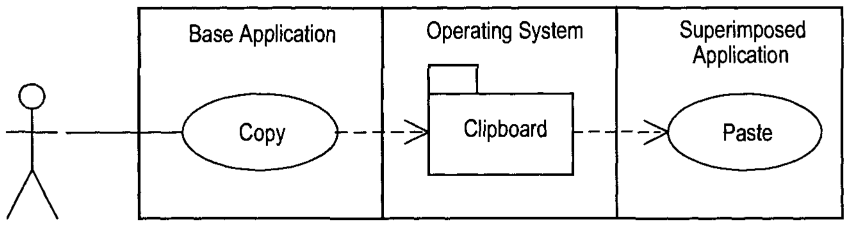
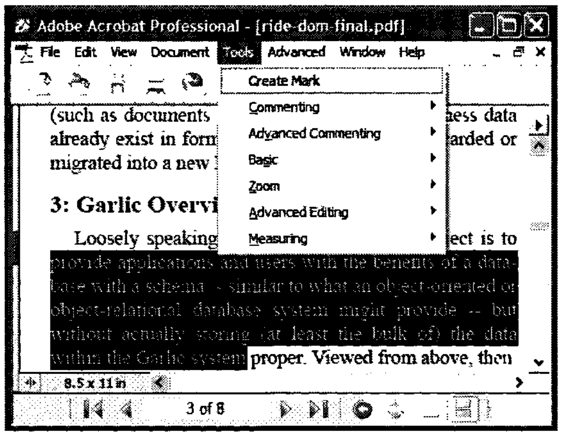


Figure 3.2: Interactively creating marks

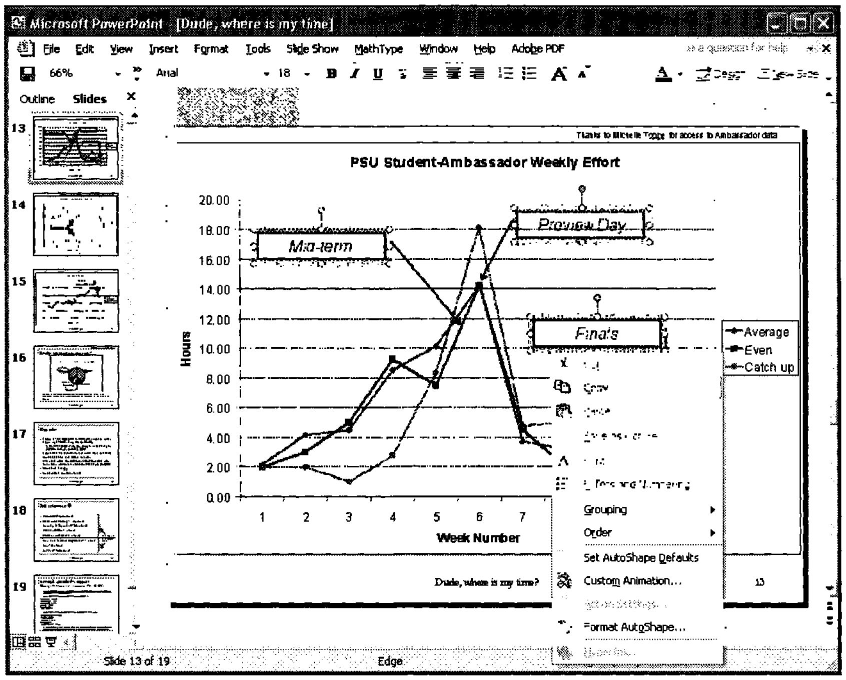

Adobe Acrobat Professional - [ride-dom-final.pdf]
File Edit View Document Tools Advanced Window Help
Create Mark
Commenting
Advanced Commenting
Basic
Zoom
Advanced Editing
Measuring
8.5 x 11 in
3 of 8

(such as documents ess data
already exist in form arded or
migrated into a new

3: Garlic Overvi

Loosely speaking ect is to
provide applications and users with the benefits of a database with a schema – similar to what an object-oriented or object-relational database system might provide -- but without actually storing (at least the bulk of) the data within the Garlic system proper. Viewed from above, then

(a)



(b)

Figure 3.3: Examples of initiating mark creation interactively. (a) Using a new tool 'Create Mark' inserted into Acrobat; (b) Using the native copy operation in MS PowerPoint

There are several ways to implement the "Copy" operation in a base application. For example, some base applications (such as Adobe Acrobat [8]) allow their user interface (UI) to be extended. In this case, a special mark-creation tool can be inserted into the application. The user invokes this special tool to create a mark descriptor. Some base applications (such as MS PowerPoint [96]) provide a hook into their native copy operation. When the user copies information to the clipboard in these applications, the hook can be used simultaneously to copy a mark descriptor to the clipboard.

Figure 3.3 illustrates these two example means of initiating mark creation. In the first case, a mark to a text selection is being created using a special tool named 'Create Mark' inserted into Acrobat. In the second case, a mark is being created to three text boxes in a slide using the clipboard-copy operation available natively in MS PowerPoint.

3.2. Representing and Accessing Base References

In this section, we describe two representations for a base part's descriptor (delimited string and XML fragment), the notion of a descriptor repository, and a means of representing a base-part reference as a Uniform Resource Identifier (URI) [15]. We also introduce a run-time object representation for base-part references.

3.2.1. Descriptors as Delimited Strings

A descriptor represented as a delimited string is a sequence of sub-strings separated by the "tab" character (the Unicode character `\u0009` [157]). The first sub-string identifies the kind of base part described. The second sub-string identifies a software wrapper called a *context agent* used to interact with the base part described. The first two sub-strings of a descriptor are required, but context-agent developers are free to decide the other sub-strings. At run time, SPARCE interprets only the first two sub-strings, and passes the entire descriptor to the appropriate context agent. Section 3.3 describes context agents.

The following is an example mark descriptor from our SPARCE implementation (described in Section 3.6.1). The symbol → represents the tab character. (The spaces around a tab character are included only for clarity.) The second sub-string in this example shows the name of an ActiveX class [93]. The third sub-string denotes that the sub-document referenced is a text selection in a PDF document. The fourth sub-string indicates that the referenced sub-document ranges over the Words 395-439 on Page 2. The last sub-string shows the date and time at which the descriptor was created. The ellipsis denotes sub-strings omitted for brevity.

```
Mark → AcrobatAgents.PDFAgent → AcrobatPDFTextMark → 2|395|439 →...→ 2004-05-28 14:03:02
```

A descriptor is commonly copied to the clipboard as a delimited string when a mark is created interactively (as described in Section 3.1).

3.2.2. Descriptors as XML Fragments

A descriptor may also be represented as an XML element. The name of the element (Mark, Document, App) is derived from the kind of the base part described. Figure 3.4 shows the XML representation for three descriptors in our SPARCE implementation. In each descriptor, the *optional* attribute ID of the top-level element denotes the globally-unique identifier (GUID) [18] assigned to the descriptor. (The figure shows simplified values instead of true GUIDs to improve readability.) The text content of the *mandatory* sub-element Agent identifies the context agent used to interpret the descriptor. Other than the sub-element Agent, context-agent developers are free to choose the inner structure of a descriptor.

The representation of descriptors in Figure 3.4 is *normalized* [12] because the mark descriptor references a document descriptor (using the element DocumentID), and a document descriptor references an application descriptor (using the element AppID). This representation reduces redundancy when more than one mark is created in the same document, or when marks are created in more than one document that requires the same base application.

A descriptor may directly contain another descriptor (in an *un-normalized* fashion), instead of referencing the other descriptor by its ID. For example, a mark descriptor may contain the element Document directly instead of the element DocumentID.

```
<Mark ID="M4">
 <Agent>AcrobatAgents.PDFAgent</Agent>
 <Class>AcrobatPDFTextMark</Class>
 <Address>2|395|439</Address>
 <Description>
  Page 3 in f.pdf (Acrobat PDF)
 </Description>
 <CachedText>provide applications and ...</CachedText>
 <Who>smurthy</Who>
 <Where>C3</Where>
 <When>2004-05-28 14:03:02</When>
 <DocumentID>D6</DocumentID>
</Mark>
```

(a)

```
<Document ID="D6">
 <Agent>AcrobatAgents.PDFAgent</Agent>
 <Location>E:\Base\f.pdf</Location>
 <AppID>A8</AppID>
</Document>
```

(b)

```
<App ID="A8">
 <Agent>AcrobatAgents.PDFAgent</Agent>
 <Name>Adobe Acrobat 5.0</Name>
</App>
```

(c)

Figure 3.4: Base-part descriptors represented as normalized XML fragments. (a) A mark descriptor; (b) A document descriptor; (c) An application descriptor

An SA may optionally store (some or all of) the descriptors it employs in a *descriptor repository*, which is a persistent collection of descriptors. For a descriptor stored in a repository, the SA includes only the descriptor's GUID in its SI, instead of including the descriptor directly. Figure 3.5 shows an XML representation of a part of the

Sidepad document in Figure 1.3. The first Sidepad item shown, denoted by the first instance of element Item, embeds a complete mark descriptor. The second item references a mark descriptor stored in a descriptor repository. (Chapter 7 describes how an SA associates a repository with SI.)

```
<SidepadDoc title="Data Integration">
 <Item name="Goal">
  Mediate heterogeneous data sources without replicating data
  <!--Embed a mark descriptor directly in SI. ID is optional in this case. -->
  <Mark ID="M4">
   <Agent>AcrobatAgents.PDFAgent</Agent>
   <Class>AcrobatPDFTextMark</Class>
   <Address>2|395|439</Address>
   <Description>Page 3 in f.pdf (Acrobat PDF)</Description>
   <CachedText>provide applications and ...</CachedText>
   <Who>smurthy</Who>
   <Where>C3</Where>
   <When>2004-05-28 14:03:02</When>
   <Document>...</Document>
  </Mark>
 </Item>
 <Item name="Model">
  Provides a unified schema expressed in...
  <!--Reference a mark descriptor stored in a repository. ID is mandatory in this case. -->
  <Mark ID="M12"/>
 </Item>
</SidepadDoc>
```

Figure 3.5: Example use of mark descriptors in SI represented as XML

SPARCE manages descriptor repositories, and assigns a GUID to each descriptor in a repository. We do not fix a representation scheme or data model for descriptors in a repository, but provide ways to represent descriptors in any schema in the relational and XML models. (The XML fragments in Figure 3.4 use the scheme that our prototype SPARCE implementation employs.) Chapters 4 and 6 describe the use of descriptors in the relational model. Chapters 4 and 7 describe in detail the use of descriptors in the XML model.

Referencing a base part using a GUID creates a dependency between an SA and a descriptor repository. This dependency does not exist if SI includes descriptors directly, but directly including descriptors does not eliminate the dependency of SI on base documents and applications. Chapter 10 describes a means to manage these dependencies (when interchanging bi-level information).

3.2.3. Referencing Base Parts using URIs

We now describe a means (that we have defined) to represent a base-part reference as a URI. (A URI names a resource, such as a document or a printer, independent of the resource's location.)

```
URI              =  scheme ":" hier-part [ "?" query ] [ "#" fragment ]
hier-part        =  <<as defined in RFC 3896>>
fragment         =  <<as defined in RFC 3896>>
scheme           =  "sparce"
query            =  bp_reference ["?" action]
bp_reference     =  bp_descriptor / bp_id
bp_descriptor    =  "descriptor=" descriptor ["?" enc_name_value]
descriptor       =  <<a serialized descriptor, possibly encoded>>
enc_name_value   =  "encoding=" enc_type
enc_type         =  "none" / "base64"
bp_id            =  "markid=" markID / "documentid=" documentID / "appid=" appID
markID           =  <<SPARCE-assigned mark ID>>
documentID       =  <<SPARCE-assigned document ID>>
appID            =  <<SPARCE-assigned application ID>>
action           =  "action=" verb
verb             =  "activate" / "showContext" / "getContext"
```

Figure 3.6: A context-free grammar to construct URIs in the `sparce` scheme

In our approach, a base-part reference is constructed as a URI in a scheme called `sparce`. A URI in this scheme is chiefly for use in traditional documents, such as web pages, word processor documents, and spreadsheets, so that a user can add some SA capability to an existing application (such as a word processor or a web browser) without changing the application. Thus, the user is able to exploit the information model and functionality of existing applications even though the applications are not built expressly as SAs. This capability comes from the user incorporating `sparce` URIs in the information created in an existing application, and from the application or the OS invoking the registered "handler" software when a `sparce` URI is activated.

Figure 3.6 shows a context-free grammar to construct URIs in the `sparce` scheme. This grammar is to be interpreted in accordance with the general syntax for URIs specified in the Internet Engineering Task Force's Request for Comments 3896 (RFC 3896) [15]. In this grammar, brackets denote optional tokens, the slash symbol (`/`) denotes an alternative, and double angle brackets contain informal descriptions. Strings shown in double quotes must be used literally, without the quote marks. Spaces outside quotation marks are used only to improve readability. Such spaces must be ignored when constructing a URI.

The non-terminal symbols `URI`, `hier-part`, `fragment`, `scheme`, and `query` used in this grammar are originally defined in RFC 3896. We retain the RFC 3896 rules for the symbols `URI`, `hier-part`, and `fragment`, but redefine the rules for the symbols `scheme` and `query`. Specifically, we restrict the value of `scheme` to the string `"sparce"`. We also restrict the value of `query` such that it can only identify a base part and associate an action to be performed on the base part. Our rules for these two symbols generate strings that are valid according to RFC 3896.

The symbol `query` allows a base part to be referenced directly using a descriptor or using a descriptor's ID. When a descriptor is used directly, it may be encoded using the Base 64 encoding scheme [77] (which, among other things, encodes the descriptor to a string that is safe for transmission in a variety of environments).

A `sparce` URI may optionally indicate one of the following actions to be performed on the referenced base part: *activate* (the default action), *show context*, and *get context.*

The following URIs are constructed using the grammar in Figure 3.6. The first URI directs the user's computer (denoted by the server `localhost`) to activate the mark M4. The second URI retrieves the context information for document D6 from the server `sidewalk.cs.pdx.edu`. The third URI asks the local computer to activate application A8. The last URI asks the local computer to activate the mark whose descriptor is embedded in the URI. (The descriptor in the last URI example is from Section 3.2.1.)

```
sparce:localhost?markid=M4

sparce://sidewalk.cs.pdx.edu?documentid=D6?action=getContext

sparce:?appid=A8?action=activate

sparce:?descriptor=Mark→AcrobatAgents.PDFAgent→…→2004-05-28 14:03:02
```

3.2.4. An Object Model for Base References

We also define an object model to work with base parts at run time. Figure 3.7 shows this model as a static class diagram drawn using the Unified Modeling Language (UML) syntax [159]. The superimposed application and base applications (that is, the packages shaded gray) are not part of SPARCE, but are shown for completeness.

SPARCE does not define the classes shown with filled lines (for example, MS Word Agent), but it instantiates them at run time to interact with the base layer.

The abstract class Context-aware Object represents a reference to a base part one might "see in context" and for which context information can be obtained. Marks, documents, and applications are context-aware objects. A context-aware object is created from a descriptor, and has a GUID. The GUID is the same as that of the source descriptor, if the descriptor has a GUID. If the source descriptor does not have a GUID, a new GUID is assigned to the resulting context-aware object. The same GUID is also assigned to the source descriptor.

An SA can work with the class Context-aware Object to interact with a base part regardless of its kind. It can cast a context-aware object to a mark, document, or an application (as appropriate) to work with aspects specific to the kind of the base part.

3.2.5. Storing and Accessing Base References

The abstract class Descriptor Repository in Figure 3.7 defines the API to create, store, and retrieve base-part descriptors. The method GetCAO creates a run-time object representation of a base-part descriptor. It creates an instance of the class Mark, Document, or Application based on the descriptor, and casts the object created as an instance of the class Context-aware Object. An SA uses this method to work with a base part whose descriptor is stored directly in SI.

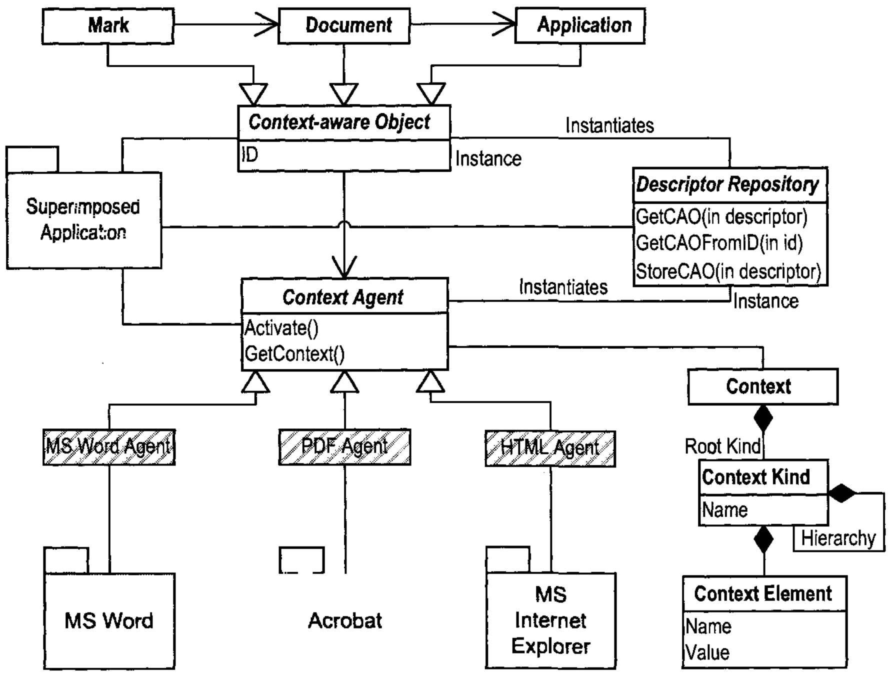


Figure 3.7: The SPARCE object model

The method GetCAOFromID returns an instance of Context-aware Object for the descriptor whose GUID is supplied, after retrieving the descriptor from the repository. An SA uses this method to work with a base part for which only the descriptor ID is stored with SI (instead of the complete descriptor being stored with SI).

The method StoreCAO stores a descriptor in a descriptor repository and returns the GUID assigned to the descriptor. Two versions of this method exist: one accepts a descriptor; another accepts an instance of Context-aware Object (possibly created using the method GetCAO).

Any number of descriptor repositories (that is, instances of implementations of the abstract class Descriptor Repository) may exist. An SA might even use multiple descriptor repositories simultaneously. Chapter 7 describes in detail the use of descriptor repositories in the XML model.

The API described in this section does not include methods to retrieve descriptors. We consider that aspect as a part of representing, accessing, and transforming bi-level information in specific data models. Table 2.1 lists the data models we have considered.

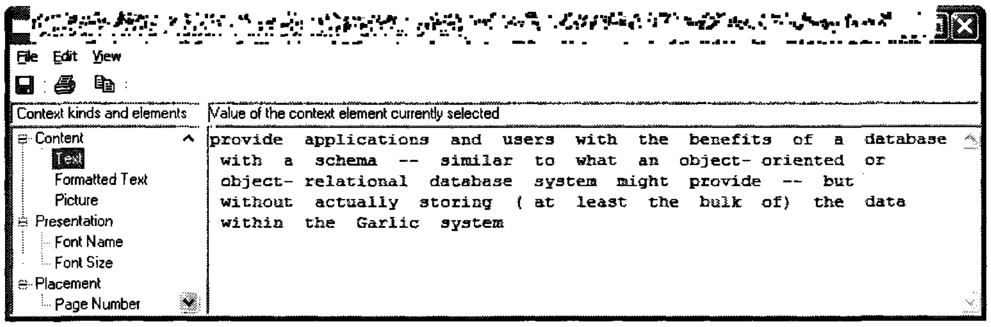
An SA may freely alter the descriptors it stores with SI, but we do not allow an SA to directly update a descriptor stored in a repository. To ensure repository consistency, we allow only components of SASS to modify a descriptor in a repository. For example, the service to interchange bi-level information alters a base document's descriptor if the base document is relocated. Chapter 10 describes the interchange service.

3.3. Representing and Accessing Context Information

This section describes how context information for a base part is represented and retrieved. It also introduces the abstraction *context agent* and shows how it is used to activate a base part and retrieve context information.

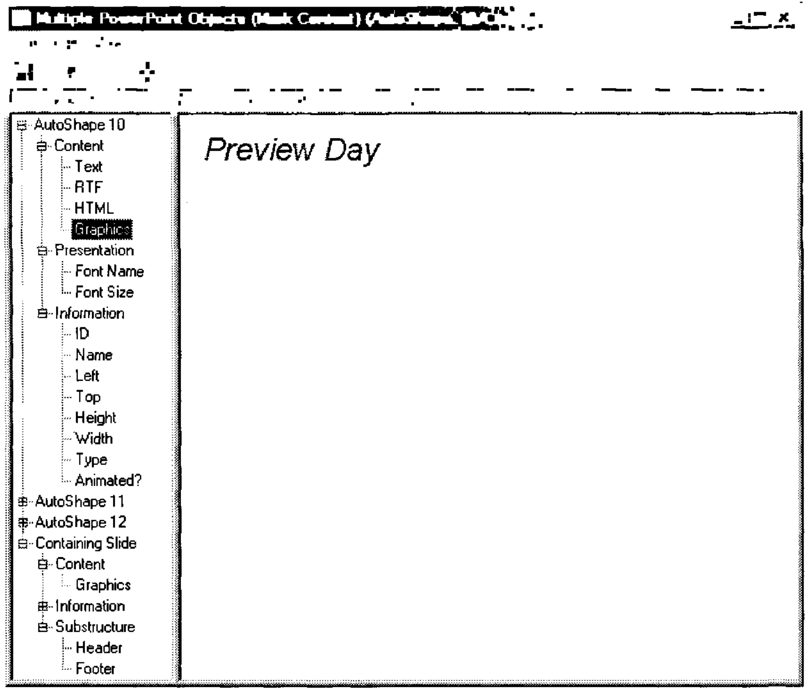
3.3.1. Representing Context Information

The context information for a referenced base part is a *hierarchical property set* (that is, a set of name-value pairs organized hierarchically). In this scheme, context elements are organized into *context kinds*. For example, information such as font name and font size are of the kind "presentation", whereas information such as line number and page number are of the kind "placement". A context kind may have sub-kinds. Pieces of information at the leaf level of a context hierarchy are called *context* *elements*. (This organization of context information is analogous to a hierarchical structure of directories and files: A context kind is similar to a directory; a context element is similar to a file.)



```
provide applications and users with the benefits of a database
with a schema -- similar to what an object- oriented or
object- relational database system might provide -- but
without actually storing ( at least the bulk of) the data
within the Garlic system
```

(a)



Preview Day

(b)

Figure 3.8: Context information from marks displayed in the Context Browser. Figure 3.3 shows the corresponding marked regions. (a) Context information for a PDF text selection; (b) Context information for a selection of multiple objects in an MS PowerPoint presentation

Figure 3.8 shows the context information retrieved from the two marked regions shown in Figure 3.3. The context information for each mark is shown in a *Context Browser*, a utility we have implemented (using the access mechanism described in Section 3.3.2). Figure 3.8(a) shows the browser displaying the partial context information for the mark to the PDF text selection of Figure 3.3(a). The tree in the left pane displays the context hierarchy. The right pane displays the value of the context element currently selected in the context hierarchy. In this case, the browser is showing the value of the text excerpt (which is a string) retrieved from the mark.

Figure 3.8(b) shows the context browser displaying a part of the context information from the mark created in Figure 3.3(b) to three text boxes in an MS PowerPoint slide. The top-level entries named AutoShape 10, AuthoShape 11, and AutoShape 12 in the context hierarchy represent the three text boxes. (PowerPoint assigns these names to the text boxes). The entries under AutoShape 10 show the context hierarchy for the text box with the content 'Preview Day'. The last top-level entry named Containing Slide shows the context hierarchy for the slide that contains the three marked text boxes. The right pane in the context browser is currently showing an image of AutoShape 10.

Representing context information as a hierarchical property set enables developers to support a context hierarchy that is specific to a base type (that is, type of BI) as illustrated in Figure 3.8. The representation also lets a developer customize the hierarchy for each mark. For example, the context for an MS Word mark to text situated inside a table can include a ‘column heading’ context element, but the context for a mark to text outside any table can exclude that context element.

We define both object and XML representations for context information. In Figure 3.7, the classes Context, Context Kind, and Context Element define the object model. Chapters 7 and 9 discuss the XML model.

3.3.2. Accessing Context Information

SPARCE uses an abstraction called a *context agent* (which is a mediator [162]) to activate a context-aware object and to retrieve context information for it. In Figure 3.7, the class Context Agent models a context agent. Several context-agent implementations (that is, specializations of the class Context Agent) may exist. Figure 3.7 shows three such implementations: MS Word Agent, PDF Agent, and HTML Agent, each supporting a distinct base type with the help of an appropriate base application. SPARCE pairs a context-aware object with a context-agent implementation based on the information contained in the descriptor for the context-aware object. For example, the element Agent in Figure 3.4(a) contains the name of the ActiveX class that implements a context agent for PDF marks.

SPARCE passes the complete descriptor of a context-aware object to the associated context agent. The agent interprets the descriptor, and performs the operations an SA requests.

An SA uses the context agent abstraction to operate on a context-aware object. Using this abstraction instead of using specific implementations enables an SA to work with any supported base type and context-aware object. Also, it allows new context-agent implementations to be added and existing context agents to be modified, with minimal advese impact on the SAs.

An SA uses the method GetContext to retrieve context information. In response, a context-agent implementation returns an instance of the class Context (containing the context information). The SA uses the retrieved context information as suits it. For example, by default, Sidepad populates the descriptive text of an item from the text excerpt of the mark associated with the item. SuperMix synchronizes a composition using the duration information obtained from the context of a mark to an audio or video clip. (See Section 1.2.)

3.3.3. Activating Base Parts

Activation is the process of showing a referenced base part in its original context. The result of activating a base part varies across references, but in general, activating a base application launches the application; activating a base document activates a base application and then opens the document; and activating a mark activates the appropriate base document and then "highlights" the sub-document which the mark references.

SPARCE supports two styles of activation: traditional style and arena style. The *traditional style* conceptually mimics the manual process a user follows to activate a base part. In this activation style, a base application decides the characteristics of the window where the base part is displayed. For example, Figure 1.4 shows the result of a traditional activation. In this case, the base application, Adobe Acrobat, determines the location of the window on the screen, and possibly the window dimensions. If two marks for the same application are activated, the exact location and dimensions of the two windows might not be predetermined: The two marks may be activated in the same window or in different windows. If the marks are activated in the same window, the mark activated later might replace the result of the earlier mark activation. If the marks are activated in different windows, the windows might overlap.

The traditional style of activation suffices for SAs such as Sidepad, but other SAs such as SuperMix need better control over activation. We provide *arena style* activation to support such applications. An *arena* is a UI window that an SA may split into smaller regions called *panes*. The SA may then direct the result of activating a base part to a particular pane. For example, Figure 1.7 shows two panes activated by SuperMix: a video mark in the top pane, and an HTML mark in the bottom pane. When the current video clip is completely played, SuperMix plays the next clip in the top pane, and shows the text for the new video clip in the bottom pane. The location and the dimensions of each pane are unaltered between activations.

An SA uses the same set of context-agent implementations for either style of activation, except that in the arena style, it provides each context agent a handle to the pane that should contain the result of activating the base part.

Every context-agent implementation supports the traditional style of activation and optionally supports the arena style. An SA can determine at run-time if a context-agent implementation supports the arena style of activation (by querying the interfaces that the context-agent implementation supports).

3.4. Supporting New Context Elements and Base Types

Supporting new context elements or changing the context elements supported by a context agent only requires changing the definition of the hierarchical property set in the relevant context-agent implementation. An SA may ignore new context elements if it is not capable of handling the new elements, or if it does not require them. After changing the context-agent implementation, the SA needs to be recompiled (but not rewritten) if the context-agent implementation and the SA are linked statically; the SA does not need to be recompiled if the linking is dynamic.

Support for a new base type can be added by following these five steps:

1. Study the base type to understand support for marking. This study should include understanding the addressing schemes possible for the base layer. Choose the addressing schemes to support.
2. Design the structure and content of descriptors. Figure 3.4 gives an example.
3. Determine the context elements and the context hierarchy (or hierarchies) to support.
4. Study the base application to understand the means to interactively create marks (that is, to copy descriptors to the clipboard). This step is related to providing a UI element within the base application as illustrated in Figure 3.3. Choose and implement the interactive mark-creation means.

5. Implement a context agent. This step is related to activating marks and retrieving context information for marks with the help of the base application.

Again, supporting a new base type may require SAs to be recompiled if the SAs are to be statically linked to the context agent that supports the new base type.

Multiple context-agent implementations may exist for the same base type, and these implementations may employ distinct (possibly incompatible) descriptors. However, this possibility does not pose any problem, because a context-aware object created from a descriptor is processed only by the context-agent implementation indicated in that descriptor.

3.5. Mark Robustness

We now briefly discuss issues related to mark *robustness*, that is, the ability of a mark (which is a sub-document reference) to remain valid when some aspect of its base document changes. We limit this discussion to robustness of sub-document references for simplicity, and because sub-document references present some unique challenges.

3.5.1. Mark Invalidation

A mark may be subject to three kinds of invalidation: context invalidation, address invalidation, and intent invalidation.

3.5.1.1. Context Invalidation

Context invalidation occurs when the context information for a mark changes in any manner after the mark is created. For example, the font name of a marked region in a PDF document, or the content of the text surrounding the region, might change.

Content invalidation is a special case of context invalidation. It occurs when the content of a marked sub-document changes after mark creation.

Context invalidation can affect an SA or a context-agent implementation that caches context information to improve performance or to support *disconnected operations* (that is, support operations on parts of base information even when some base documents are inaccessible). Context invalidation can also affect a context-agent implementation that uses a context-based addressing scheme. For example, an implementation might cache the text excerpt (retrieved using the access mechanism we define) at mark-creation time and use the excerpt as the sub-document address.

3.5.1.2. Address Invalidation

Address invalidation occurs when a mark cannot be activated even though its base application can be activated. There are several reasons for address invalidation. For example, context invalidation may cause address invalidation if the sub-document addressing scheme is based on context information (such as text excerpt or section heading). A mark's address may also become invalid when the marked region is "deleted" or if the region containing the marked region is deleted. For example, assume that the addressing scheme for marks into text selections in PDF documents uses a page number, and the starting and ending indexes of the words in the selection. Then, the address of a mark to the last few words in a PDF document becomes invalid if those words are deleted. In the same addressing scheme, the address of a mark to any selection in the last page of a PDF document becomes invalid if the last page is deleted.

Similarly, if a record in a relational database is addressed using values of key attributes, the address can become invalid if the record is deleted.

3.5.1.3. Intent Invalidation

Intent invalidation occurs when a change to a base document results in a mark that activates successfully, but the mark no longer references the sub-document the mark creator originally intended. There are several reasons for intent invalidation. For example, inserting new data into a document can shift a marked sub-document causing the mark to reference the newly inserted data.

Context invalidation can cause intent invalidation if the sub-document addressing scheme is based on context information. For example, if "slide number" is used to mark into a presentation, reordering the slides invalidates user intent.

Understanding user intent is one of the harder parts of mark management. For example, when the user marks the first paragraph in a document, it can be hard to understand if the user means to mark into the particular text of the paragraph, or if he intends to mark into whatever is the first paragraph.

Resolving a mark whose intent is invalidated might depend on the capabilities of the base application. For example, the application would need to support addressing schemes that capture the user's intention accurately (or, at least, sufficiently). Having the user direct mark resolution, or confirm the result of a resolution, is one way to handle intent invalidation. Capturing sub-document address using multiple addressing schemes at mark-creation time (the "belt and suspenders" approach) is a way to reduce the frequency of (undetected) intent invalidation.

3.5.2. The Role of Addressing Schemes

The sub-document addressing scheme that a mark uses largely determines the robustness of a mark. Several addressing schemes may be possible for a given base type, and each scheme might provide robustness under different conditions. For example, when addressing a section in an MS Word document, one can use the starting and ending indexes of the characters in the section, use the section heading, or use the text content of the section as the address. A character index remains valid as long as the document has a sufficient number of characters, but it might not retain user intent. The scheme using section heading works as long as the heading is unaltered. Finally, text content works as the address as long as the text is unique and it appears somewhere in the document.

SPARCE does not prescribe or proscribe specific sub-document addressing schemes. (It does not even interpret sub-document addresses.) Context-agent implementations are free to choose one or more addressing schemes based on factors such as the goal of addressing, the structure (or lack thereof) of base documents, and the capabilities of base applications.

3.5.3. Improving Mark Robustness

To improve the robustness of a mark, we make the following recommendations to context-agent implementers:

- Avoid addressing schemes based solely on content or context.
- Exploit read-only base sources where available.
- Capture some context information at mark-creation time, and use the captured context to validate and redirect a mark, if necessary.
- Use multiple addressing schemes. Ensure that each scheme provides robustness under different conditions.
- Where available, incorporate *immutable identifiers* from the base layer in sub-document addresses. (For example, MS PowerPoint assigns a unique and immutable ID to each slide.) Immutable identifiers assure that the same base object is accessed always, as long as the object is not deleted. (The use of immutable identifiers might not prevent context invalidation.)
- When resolving a sub-document address, locate the closest sub-document, or locate the containing sub-document, instead of just failing if an address is invalidated. For example, when activating a PDF mark, activate the containing page, if the marked words on that page are deleted.

3.6. Evaluation

We have evaluated the representation and access mechanisms discussed in this chapter by implementing SPARCE as middleware, and by building context agents and SAs that use the SPARCE implementation. In this section, we provide an overview of the implementation, and discuss how it performs with respect to the architectural qualities listed in Section 2.1.3. We also review the key design decisions the evaluation validates and briefly discuss some of the design alternatives.

3.6.1. Implementation

3.6.1.1. SPARCE

We have implemented the architectural components of SPARCE shown in Figure 3.7 for the MS Windows platform using primarily the ActiveX technology [93]. This implementation supports both the traditional and arena style of activation (described in Section 3.3.3). The implementation also includes a "handler" to interpret base-part references represented as URIs in the `sparce` scheme that are constructed using the grammar shown in Figure 3.6. (The application, most likely the OS, invokes the handler when the user activates a `sparce` URI. The handler parses the URI and uses the SPARCE API to complete the requested operation on the referenced base part.)

Our research partners at Villanova University, under the supervision of Professor Lillian Cassel [60], have ported parts of our SPARCE implementation to Java [71].

3.6.1.2. Context Agents

We have implemented context agents for the following base types: MS Word, MS Excel [96], MS PowerPoint, PDF, HTML, XML, and several audio and video formats. Table 3.1 provides an overview of these implementations and the sub-document addressing scheme each implementation employs. Our colleague James Terwilliger has also implemented a context agent for marks into form fields in applications that conform to the Guava framework [153, 154].

We briefly review some of these context-agent implementations and extensions made to base applications (to create mark descriptors).

MS Office marks: We have developed a single *add-in* (that is, software code added in) [97] for MS Office applications to copy mark descriptors to the clipboard. This add-in hooks into the native copy operation of MS Office applications (including MS Word, Excel, and PowerPoint). When the user copies a selection to the clipboard (as shown in Figure 3.3(b)), the add-in also copies a mark descriptor corresponding to the selection to the clipboard.

Though we use a single add-in to copy mark descriptors from different MS Office applications, we have implemented distinct context agents for each MS Office application. We made this choice because the sub-document addressing scheme and the process of interaction varies widely among those applications. For example, the sub-document address for an MS Word text selection contains just the indexes of the first and last characters in the selection. (MS Word presents the main text of a document as a sequence of characters.) The context for an MS Word mark can be large, but the context hierarchy tends to be fairly simple: text excerpt, containing paragraph, containing section, and so on.

In contrast, the sub-document address in the case of MS PowerPoint can be quite complex because marks may be created into a variety of information types from different views. For example, the user may select a complete slide or a range of slides in the outline view or in the slide sorter. He can select one shape or multiple shapes in a slide. When multiple slides or shapes are selected, the selected objects might not be contiguous.

Table 3.1: Overview of context agents implemented for use with SPARCE

Base types	Base application	Sub-document addressing scheme
Text selection in an MS Word document	MS Word	Indexes of the first and last character of the text selected
Range of cells in a spreadsheet	MS Excel	Sheet name, row names and column names of the first and last cell of the range selected
Text selection, shape,set of shapes, slide, set of slides	MS PowerPoint	View type, slide identifier, shape identifier, indexes of the first and last character of the text selected
Text selection in a PDF document	Adobe Acrobat	Page number, indexes of the first and last words of the text selected
Text and image selection in an HTML page	MS Internet Explorer [95]	Path to the containing element in DOM tree, text of selection (for text only)
One or more nodes in an XML document	MS XML 4.0 [107]	XPath [166] and XPointer [167] expressions
Audio span, video span in WAV, MP3, MPEG, and other formats	MS Windows Media Player [103]	Time offsets for the beginning and end of the span
Form fields such as textboxes and lists	Guava [154]	Application name, form name, field name

The context hierarchy of an MS PowerPoint mark can be much more complex than that of an MS Word mark, largely due to the inherent nested organization. For example, a text selection inside an MS PowerPoint text field has the usual context information such as plain text excerpt, HTML excerpt, and font information. Its container, the text field, adds information such as name, shape, ID, size, and location. The containing slide adds information such as ID, number, header, and footer. Figure 3.8(b) illustrates some of these context elements.

Activating an MS PowerPoint mark requires special care. For example, a mark created in the editing mode might be activated when the base presentation is being shown (that is, when the presentation is running). In this case, the mark should be activated without exiting the show mode (because marks can be employed to transition among slides in different presentations).

HTML marks: We have developed a custom tool in VBScript [160] to extend MS Internet Explorer [95] to enable mark creation. The HTML context agent uses the HTML Document Object Model (DOM) [35] (which represents an HTML document as a tree) to manipulate the base document. DOM provides a browser-independent means of handling HTML marks, but some of its limitations also pose interesting challenges. For example, DOM does not provide a direct means to obtain the path to a node in a tree, or to obtain the position of a node among its siblings. Thus, given a user-selected node, the script to create a mark needs to walk up the tree to the root node to find the path to the selected node. The script must also visit the preceding siblings of the selected node, and of each ancestor node along the path, to compute the position of the node. These operations can be time consuming, especially because scripts are interpreted at run-time (not pre-compiled).

Audio and video marks: We have extended the MS Windows Media Player [103] to facilitate creation of marks into a variety of audio and video formats. To mark an audio or video span, the user separately denotes the start point and end point of the span, and then copies the span to the clipboard using a special tool added to the player. The special tool incorporates the end points of the span into the mark descriptor.

Guava marks: Our colleague James Terwilliger has implemented a context agent for marks into form fields in applications that conform to the Guava framework [153, 154]. (The Guava framework enables the use of an application's user interface as a query interface to the database that stores the application data.) To enable these marks, Terwilliger has defined a class of "markable" form fields using the .NET Framework [129]. When running a .NET application, a user can select any form field of this class, copy a mark descriptor to the clipboard, and employ the mark in any SA. Terwilliger has also implemented a context agent to activate a Guava mark and to retrieve context information for it. (Activating a Guava mark involves launching an appropriate .NET application, activating a sequence of forms in the application, and highlighting the marked field.)

3.6.1.3. Superimposed Applications

We have built six SAs using SPARCE (and other components of System S): the three SAs described in Section 1.2 (Sidepad, SuperMix, and the HTML+M Editor); an SA called the Superimposed System Information Browser (SSIB, described in Section 4.2) that allows a computer system administrator to browse information such as event logs and OS updates; an SA called the Superimposed Scholarly Review System (SISRS, described in Section 4.9.2) that facilitates superimposition of review comments; and a general-purpose browser and editor for SI represented as XML (described in Section 7.6.2). We have also modified a previously existing application called the *Schematics Browser* [17] to use SPARCE.

Our research collaborators have developed an SA called the Superimposed Multimedia Presentation Editor and Player (*SIMPEL*) [123] using SPARCE. SIMPEL is an SA to organize multimedia content on a timeline and play the content in a synchronized manner. It is developed using the .NET Framework. The same collaborators have also augmented a commercial tool called CmapTools [63] to use marks in a concept map [124]. They have introduced a new resource type called "mark" in CmapTools and allow a mark descriptor to be attached to each mark resource.

Our colleague David Archer has developed an SA called the Superimposed TRansactor for Integrating Data into Entities (STRIDE) [10] using SPARCE. STRIDE is designed to capture human judgment when integrating data for specific tasks.

3.6.1.4. Clipboard and UI Widgets

We have defined the *Clipboard* abstraction, and implemented it for the MS Windows platforms: The MS Windows implementation is an ActiveX wrapper to the MS Windows clipboard API. It includes functions that make it easy to copy mark descriptors to, and retrieve descriptors from, the clipboard. (The clipboard implementation may be quite different on other platforms. For example, one might implement clipboard operations from the ground up on platforms that do not natively provide a clipboard.)

The clipboard implementation can keep track of multiple mark descriptors copied, and allows SA developers and users to retrieve any of the copied descriptors. This feature allows the SA user to create several marks in the base layer, possibly in different base documents, before employing one or more marks in an SA.

We have implemented two UI widgets for the benefit of both context-agent implementers and SA developers: a set of tabbed "property pages" to display properties (such as ID and base address) of a context-aware object; and a *Context Browser* to let a user browse context information retrieved from any context-aware object. (Figure 3.8 shows two uses of the Context Browser.)

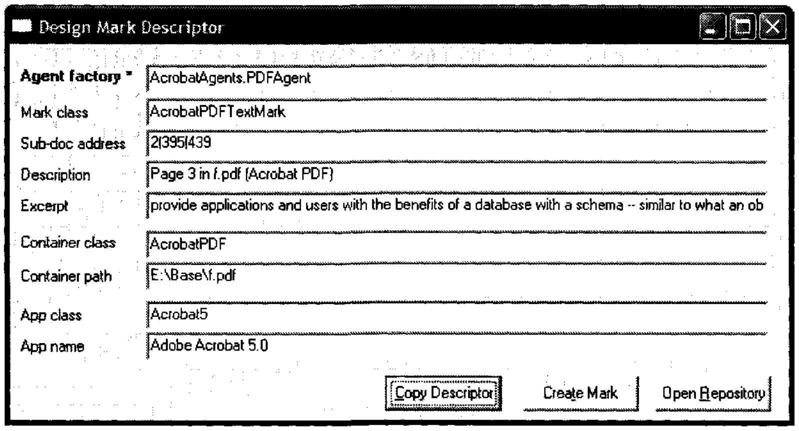
3.6.1.5. Development and Testing Aids

We have implemented the following development and testing aids for context-agent implementers and SA developers:

- A utility to construct a mark descriptor and copy it to the clipboard without extending a base application. This utility is useful in the initial stages of adding support for a new base type. Figure 3.9 shows the use of this utility to construct the mark descriptor shown in Figure 3.4. The field Agent factory contains the name of the context-agent class. The other fields are self explanatory.
- A "Do Nothing" context-agent class to test a mark descriptor without implementing a context agent for the descriptor. An instance of this class accepts any descriptor, but does not interpret it. Also, it returns an empty property set as the context for any mark.
- Logging and exception-reporting components to trace the execution path of SPARCE, the context agents, and the SAs.

A context-agent implementer may use any SA as a testing aid because an SA can work with any context-agent implementation. Likewise, an SA developer may use any base type and any context-agent implementation to test the SA's ability to incorporate and activate marks.

Context-agent implementers and SA developers may use the Context Browser to test retrieval of context information for any mark.


Design Mark Descriptor

Agent factory *	AcrobatAgents.PDFAgent
Mark class	AcrobatPDFTextMark
Sub-doc address	2|395|439
Description	Page 3 in f.pdf (Acrobat PDF)
Excerpt	provide applications and users with the benefits of a database with a schema -- similar to what an ob
Container class	AcrobatPDF
Container path	E:\Base\f.pdf
App class	Acrobat5
App name	Adobe Acrobat 5.0

Figure 3.9: Utility to construct and test a mark descriptor

3.6.2. Architectural Qualities

In this section, we summarize our experience with implementing and maintaining SPARCE, the context agents, and SAs to show that the system possesses the desired architectural qualities for SASS (listed in Section 2.1.3). The descriptions of the architectural qualities are reproduced here (in italics).

3.6.2.1. Functionality

The implementation must provide runtime services that are helpful in implementing the seven application capabilities listed in Section 1.1.

SPARCE, the context agents, the Clipboard, and the UI widgets collectively implement the runtime services Reference Management, Context Management, and UI Widgets in Figure 2.1. Together, they also support Capabilities 1 through 4 listed in Section 1.1. These capabilities relate to creation and activation of marks, and to creation and organization of SI.

The URI representation of a mark descriptor in the `sparce` scheme (described in Section 3.2.3) and the corresponding handler implementation together support Capability 6, making it possible to employ marks in traditional documents such as word processor documents and spreadsheets.

Capabilities 5 and 7 (transforming and interchanging bi-level information, respectively) are supported by other parts of System S with the help of SPARCE. Chapters 9 and 10 discuss support for these capabilities.

3.6.2.2. Reusability

Many SAs must be able to use the same SASS implementation. More than one SA instance must be able to run simultaneously on the same computer, and each instance must be able to interact with multiple base documents.

All the SAs implemented use the same SPARCE implementation. We have not made any special changes in the SPARCE implementation, or in the context agents, for any of these SAs. We have run (several) instances of different SAs simultaneously on the same computer and have verified that each SA instance is able to use marks in multiple base documents.

The handler software for the `sparce` URI scheme uses the same context-agent implementations the SAs use. We have used URIs in the `sparce` scheme in documents created by third-party applications such as MS Word, Adobe Acrobat, and HTML editors. We have used these applications simultaneously with the SAs.

We have also verified that different SAs, and traditional applications that employ `sparce` URIs, can reuse mark descriptors stored in the same descriptor repository.

3.6.2.3. Modifiability

It must be possible to independently improve SASS and the SAs, with minimal adverse impact on each other.

Over the course of implementation (between March 2003 and January 2007), we have updated SPARCE, the context-agent implementations, and the SAs several times. For example, the source files related to SPARCE have been checked into our version control database 352 times. (A check-in operation requires at least one change in a source file.) Between August 2005 and September 2006, there have been 13 releases of the complete implementation.

Execution tests have shown that throughout these changes and releases, modifying one part of the system (for example, SPARCE) has not adversely affected other parts (for example, the context-agent implementations and the SAs). Also, upon modification of a part's source code, we have recompiled the source code for only that part. That is, we have been able to evolve SPARCE, the context agents, and the SAs independently.

The ability to modify different parts of the system without adversely affecting other parts is largely due to the separation of concerns afforded by our design (via abstractions such as context agent and context-aware object), and due to the dynamic loading, linking, and instantiation [93] of context-agent classes.

3.6.2.4. Extensibility

It must be possible to support new base types and context elements without affecting existing SAs and context agents.

We have used the steps outlined in Section 3.4 to develop all the context agents listed in Table 3.1, without any adverse impact on SPARCE and the SAs. We have also verified that changing the definition of context a context agent supports does not affect SPARCE and the SAs.

An SA that depends on a specific context element might be affected if a context agent no longer supports that element, but attempts to seek non-existent context information does not cause an exception in SPARCE and the implemented context agents. In this case, the SA might need to be altered to remove its dependence on the missing context element. Similarly, an SA might need to be altered if it is to take advantage of a newly added context element. Applications such as the Context Browser are unaffected by changes to the definition of context because they do not depend on specific context kinds or elements.

The ability to associate each mark with a (different) context-agent class, and the use of the abstractions context agent and context-aware object, dynamic linking and instantiation of context agents make it possible to extend support for new base types. The use of the abstractions context, context kind, and context element makes it possible to extend context definition.

3.6.2.5. Package Flexibility

It must be possible to change the location of the components of SASS to meet application and user needs. For example, we must be able to deploy the components of SASS on the same machine as the SA, or on a different machine.

The different components that make up SPARCE are packaged as ActiveX servers (seven servers in all). Due to the design of SPARCE, and some facilities in the ActiveX technology, any of these servers may be packaged either as an in-process server or as an out-of-process server. An *in-process server* runs in the address space of the client application that uses the server (and hence on the same computer as the client). An *out-of-process server* runs in its own address space. It may run on the same computer as the client or on a different computer.

With an in-process server, each client application gets its own instance of the server, whereas several clients may share the same instance of an out-of-process server. Consequently, different client applications might be able to share certain resources (such as a connection to a database) when using an out-of-process server.

We have verified that the server packaging does not affect the functionality of SPARCE and the SAs, except for some expected changes in performance [112]. For example, the execution speed tends to be better when a server is loaded in-process.

However, an out-of-process server provides resilience to both the server and the client because when one of the processes (server or client) aborts, the other process can continue to run.

We have also verified that descriptor repositories may be located on local or remote file systems (with respect to the location of SPARCE, the context agents, and the SAs). For example, we have deployed the SAs and context agents on one computer, SPARCE on another computer, and a descriptor repository on a third computer.

3.6.2.6. Testability

The SASS implementation must aid verification and validation of itself, and of the SAs that use it.

We have used the UI widgets (mentioned in Section 3.6.1.4) and our development and testing aids (listed in Section 3.6.1.5) to verify SPARCE, the context agents, and our SAs. For example, we have used the Context Browser extensively to verify the context information that a context agent returns for a mark. Also, we have frequently used the Sidepad SA to test new context-agent implementations.

We have used our logging facility to validate SPARCE, the context agents, and our SAs. Using this facility, we are able to trace the execution path of each part of the system and ensure that each part is indeed functioning as it should. However, validation of execution paths is not sufficient. Context-agent implementers and SA developers need to use appropriate techniques to validate that their implementations meet the needs of their users.

3.6.2.7. Usability

The SASS implementation must use familiar metaphors, and follow relevant development and UI conventions. It must also aid usability of SAs developed using it.

The interactive mark-creation process requires users to perform only the familiar and natural "Copy" and "Paste" (as described in Section 3.1), and the mark-creation process is similar across base types. Also, a user may use the same descriptor in any number of SA instances. The user may also copy several descriptors, from different base documents, to the clipboard before using them in any SA.

Our experience (and that of our collaborators) shows that it is quite easy to develop context agents and SAs with SPARCE. The following list illustrates the ease of use.

- A context-agent needs to implement only four functions: `Activate`, `GetContext`, and `GetElementValue`, plus a constructor.
- Copying a mark descriptor to the clipboard after a user has selected a sub-document region can often be accomplished in one line of code. For example, the following line of MS Visual Basic [101] code suffices to copy a descriptor string to the clipboard:

  ```
  SPARCEClipboard.Copy(descriptor)
  ```
- Retrieving a mark descriptor from the clipboard and creating a mark is usually accomplished in one line of code. For example, the following line of MS Visual Basic code creates a mark using the descriptor most recently placed in the clipboard. The identifier `repository` denotes an instance of a descriptor repository.

```
repository.GetCAO(SPARCEClipboard.RecentDescriptor)
```

- James Terwilliger needed only 8 hours [155] to implement the Guava context agent [153] (following the steps outlined in Section 3.4).
- Our collaborators spent 120 hours developing the SA SIMPEL [123], of which they spent only about two hours on tasks related to integration with SPARCE.

3.6.3. Design Decisions

In this section, we summarize the key design decisions our evaluation has validated. We also briefly discuss some of the design alternatives considered.

3.6.3.1. Flexible Representation and Storage of Base Descriptors

As described in Section 3.2, a base-part descriptor may be represented as a delimited string, XML fragment, and as a URI. (Chapter 4 discusses the representation of descriptors in the relational model.) These choices allow base parts to be employed in a variety of applications. Further, a descriptor needs to include only the name of a context-agent class. The rest of the descriptor structure is unconstrained. This flexibility allows a developer to structure descriptors according to his needs.

The alternative of fixing a data model and structure for descriptors would simplify the system, but it would also limit the number of applications that benefit from our framework.

We allow each base-part descriptor to specify the context-agent class used to interact with the referenced part. Thus, each base part can potentially have its own context-agent implementation. The alternative of using a single context agent for each base type (or base document) prevents the use of domain-specific context agents. For example, when working with patent information in PDF format, one might use a context-agent implementation that returns domain-specific context information such as "dependent claims", but use a different PDF agent implementation in other applications.

As described in Section 3.2.2, an SA may include base-part descriptors directly in SI, or it may store descriptors in a repository that SPARCE manages. This choice allows an SA developer to store descriptors in a location and manner that is most appropriate for the SA, yet be able to perform all operations on the referenced base parts. For example, an SA might deposit its SI and descriptors in a digital library managed by a third party [11, 112].

The alternative of requiring an SA to manage storage of descriptors itself likely increases SA-development effort and hinders sharing of SI among SAs and among SA users. Alternatively, requiring an SA to always store descriptors in a SPARCE-managed repository might (seriously) constrain SA development and deployment. For example, when using a SPARCE-managed repository, an SA developer must use the SPARCE API to manipulate descriptors, and he might need to transform the descriptors from SPARCE's data model to the SA's data model.

3.6.3.2. Use of High-level Abstractions

In our design, context-agent implementations are unaware of the existence of SAs. In turn, SAs are unaware of the existence of specific context-agent implementations, because SAs activate base parts and access context using only the classes Context Agent, Context, Context Kind, and Context Element.

This isolation between context-agent implementations and SAs makes it possible for context agents and SAs to evolve independently, without affecting each other. Figure 3.10 shows an SA's view of SPARCE. This figure is obtained from Figure 3.7 by removing from that figure the classes that an SA does not directly use. We have added a link between an SA and the class Context to denote that an SA may consume context information.

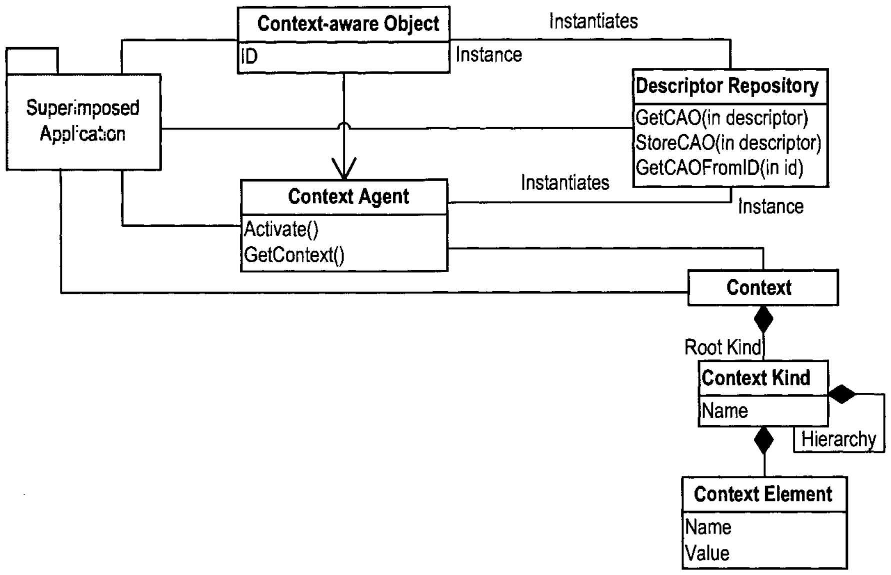


Figure 3.10: A superimposed application's view of SPARCE

The use of the abstractions Context Kind and Context Element allows the run-time representation of any context information, but in some programming languages, a naïve implementation can result in loss of compile-time type guarantees. For example, a naïve MS Visual Basic 6.0 [101] implementation would represent both a text excerpt and a page number as the same type (probably a string), but a Java implementation can distinguish the types of these two context elements (as String and Integer, respectively). (A Visual Basic 6.0 implementation can define wrapper classes such as "String" and "Integer" to aid compile-time typing.)

Another design choice we made is related to the use of the class Context Agent to access base parts, instead of extending the class Mark for each type of mark to be supported. We illustrate our choice and an alternative using an example.

Consider the task of supporting references to MS Word marks, MS Word documents, and the MS Word application. In our approach, a *single* context-agent class (called MS Word Agent in our implementation) can accomplish this task because much of the code to work with MS Word marks, documents, and the application is the same. An instance of the class Mark, Document, or Application is passed to an instance of this context-agent class to activate and access the appropriate base part.

In our approach, it is possible to reuse the same context-agent instance to access multiple base parts by reinitializing the context-agent instance with a different context-aware object. For example, the class MS Word Agent can be first initialized to access an MS Word mark (or document), and then reinitialized to access another MS Word mark (or document). As Chapter 6 illustrates, this ability can reduce execution time and save memory when retrieving context information for a large number of context-aware objects.

An alternative approach is to extend the classes Mark, Document, and Application (using inheritance) to implement the classes Word Mark, Word Document, and Word Application, respectively. This approach results in three classes, each with similar code (or four classes, with the fourth class privately implementing common code).

Attempting to reuse context-aware objects (as is possible in our approach) in the alternative approach can adversely affect SAs. For example, assume two SAs use the method GetCAO to retrieve the *same* mark, say an instance of the class Word Mark, from a descriptor repository. In this case, the mark the first SA holds would be invalidated if the second SA reuses the Word Mark instance to load another mark. (A shared context-agent instance can be similarly invalidated in our approach. The situation is remedied using a new instance of the context agent, but the alternative approach would need two objects— Mark and WordMark—to remedy the situation.)

3.6.3.3. Representing and Accessing Context as Hierarchical Property Sets

Our evaluation shows that hierarchical property sets aptly handle the wide variability in context information among base types (as illustrated in Figure 3.8), and among marks of the same base type. A hierarchical property set provides a uniform representation for context information and it simplifies the API to access context information. For example, the object model shown in Figure 3.7 uses only the classes Context, Context Kind, and Context Element to model context information for any mark. With these three classes, an SA is able to programmatically access context information for marks of any base type.

A simple alternative is to use non-hierarchical property sets, but that representation makes it hard for the developer to organize (and for the user to comprehend) context information. For example, without hierarchies, it would be quite challenging to organize the context information shown in Figure 3.8(b).

Another alternative is to define a separate schema for the context information applicable to each base type, and define an API that is specific to each base type. For example, define a schema specific to context information for MS Word marks, and define a corresponding API. Similarly, define a schema and API specific to MS PowerPoint. (This is the approach MS Office applications take.)

In this alternative, an SA can detect new or missing context elements at compile time when a context agent revises the definition of context, but it requires that the SA and the context API implementation be linked statically, making it harder to independently evolve SAs and context agents. Also, this approach widens the API to access context information. For example, the API to access the context information for the MS Word *Range* object (which represents a selection in an MS Word document) includes over 30 members [105]. Each Range object exposes these members, even when a member is not applicable to a particular object. (The value of a member that does not apply is typically NULL or empty).

In contrast, our context access API defines only 8 methods, and is able to provide context information for any base type. Also, the context of a mark contains only those elements that apply to that mark.

3.6.3.4. Use of the Clipboard

Our design employs the clipboard to interactively create marks (in addition to providing a means to programmatically create marks without using the clipboard), which has two key advantages. First, it improves usability of the system because a user performs only familiar and natural clipboard operations and he typically performs only two operations ("Copy" and "Paste") to create and consume a mark. Second, using a clipboard de-couples base applications from SAs, and allows each class of application to evolve without affecting the other. (A related benefit is that, in some operating environments, supporting "copy and paste" makes it easy to support the "drag and drop" means to create and consume marks.)

An alternative to using the clipboard is to consume a mark as soon as it is created, but doing so would require an SA to be running at mark-creation time. Also, if multiple SAs are running when a mark is created, it is hard to (automatically) choose the SA in which the mark is consumed. That is, several usability issues would exist.

3.6.4. Evaluation Summary

Our evaluation validates our representation for base-part descriptors and our middleware architecture to activate the referenced base parts and to retrieve context information from the parts. The evaluation also validates our design decisions, and shows that our choices indeed satisfy the architectural desiderata we set up at the beginning of this research.

Table 3.2 summarizes the key design decisions and the architectural qualities to which each decision contributes.

Table 3.2: Key design decisions and the architectural qualities to which each decision contributes

Design decision	Functionality	Reusability	Modifiability	Extensibility	Package flexibility	Testability	Usability
Flexible descriptor representation	✓	✓		✓			✓
Flexible descriptor storage	✓	✓					✓
Context-agent class per descriptor	✓			✓		✓	✓
High-level abstractions	✓	✓	✓	✓	✓	✓	✓
Dynamic instantiation of context agents		✓	✓	✓	✓		
Choice in server packaging and deployment	✓	✓	✓	✓	✓	✓	
Context as hierarchical property set	✓	✓	✓	✓			✓
Use of clipboard for interactive marking	✓	✓	✓	✓	✓		✓

3.7. Related Work

Chapters 1 and 2 list some systems that provide features comparable to those of SPARCE. In this section, we provide more details about some of those systems and describe a few systems not mentioned in the earlier chapters. The systems we describe give an insight into alternative approaches to solving the problems SPARCE solves.

Before we describe the alternative approaches and systems, we briefly mention the predecessors of SPARCE.

3.7.1. Predecessors of SPARCE

The mark abstraction was first defined in a middleware architecture called SLIM and has been used to build an SA called SLIMPad [32]. SLIM supported marks over multiple base types, but a mark could only be activated. Context information could not be retrieved for a mark.

Prior to SLIM, Delcambre and others built a system called CARTE to provide navigation over a set of HTML pages using superimposed structured maps [31] based on topic maps [158]. (Section 4.10 reviews structured maps and topic maps.) CARTE did not use (or have) the mark abstraction. Instead, it referenced a base selection using a URI. That is, a reference to a sub-document was possible only if the base document exposed the sub-document's address as a URI. For example, CARTE could reference a span in an HTML document if the document defined a bookmark over the span. This requirement limited the number of base types that could be referenced. CARTE stored SI and the URIs in a relational database.

3.7.2. Early Visions

We first give an overview of some pioneering visions that have contributed to SPARCE and related systems.

3.7.2.1. Memex

In 1945, Vannevar Bush [22] envisioned a device called *Memex* to store and consult information efficiently. In his vision, Memex is a desk with translucent projection screens, a keyboard, a microfilm-based storage, and control levers for navigation. Its contents (books, pictures, periodicals, and so on) are stored as photographic images on microfilm. A user can attach a mnemonic code to an information selection (for example, to the title page of a book) and use the code later to navigate directly to that element. Because there are several screens, the user can browse a selection in one screen while a different selection is projected on another screen. The user can attach annotations to material being browsed.

Two or more information selections may be tied together using a common mnemonic code to form a *trail.* A selection may be used in any number of trails. Because trails persist, they can be recalled at any time. They can also be reproduced and passed to other users.

Some of Memex's features relevant to SPARCE are: annotation, linking, sharing, and indexing. SPARCE, along with context agents and SAs, facilitates annotation, and linking. SPARCE does not directly support sharing, but it helps other parts of SASS share bi-level information (as described in Chapter 10). SPARCE does not index contents of descriptor repositories. An SA may index its SI.

Memex assumes ownership of all referenced information; SPARCE does not.

3.7.2.2. Evolutionary List File

In the 1960s, Nelson [126] proposed a file structure called the Evolutionary List File (*ELF*) for use in his software system for personal filing and manuscript assembly.

ELF stores three kinds of elements: entries, lists, and links. An *entry* is the basic unit of information, and it can be text, a picture, or a definition of an operation. A *list* is an ordered set of entries; an entry may be placed in any number of lists. Lists are used to create categories, trails, and other structures. A *link* connects two entries in different lists; an entry in a list is linked to at most one entry in another list. Links are bi-directional.

An entry may be annotated. An annotation is also stored as an entry, with a link to its target entry. ELF supports multiple simultaneous organizations of the same information by allowing an entry to be used in more than one list simultaneously. However, an entry placed in more than one list is replicated and the replicas are kept consistent.

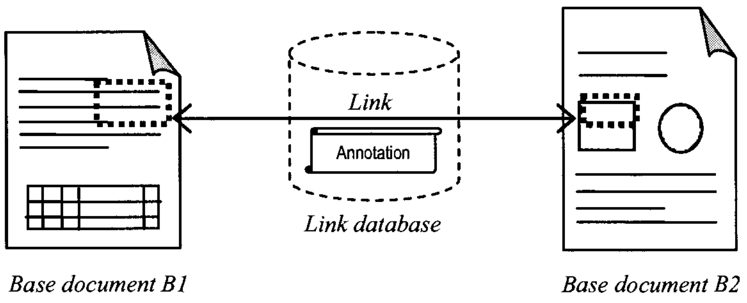
Nelson proposes versioning of entries and lists. SPARCE does not maintain versions of descriptors or base information, but an SA may maintain versions of its SI.

3.7.3. Hypermedia Systems

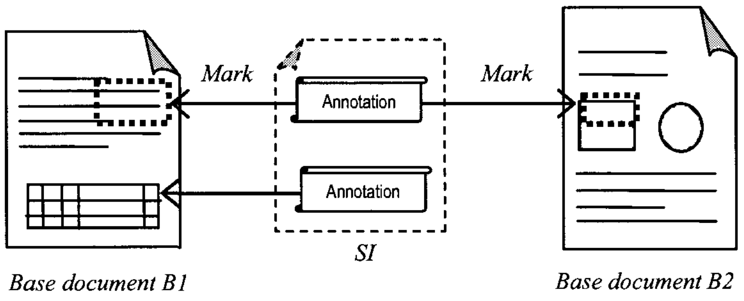
Nelson [126] first used the term *hypertext* to mean information containing text and graphics in such complex ways that it is hard to present the overall information in linear media (such as paper). The term *hypermedia* was used in the 1980s to include multimedia data such as video [26]. In this section, we first compare SPARCE with hypermedia systems in general, and then compare SPARCE with specific hypermedia systems.

In general, hypermedia systems facilitate linking of two or more documents or sub-documents. A link signifies a relationship among the linked entities and is chiefly used to facilitate navigation from one linked entity to another.

Some systems allow annotations to be attached to links. Figure 3.11(a) shows a typical hypermedia link with an attached annotation. In this approach, no new "document" needs to be created to represent a hypermedia network, because it suffices to store only the link definitions and the annotations. Some hypermedia systems also require that the linked documents, or the descriptions of the linked documents, be stored in a specific database. For example, Dexter [57] requires document descriptions to be stored in its database.



(a)



(b)

Figure 3.11: A comparison of hypertext links and marks. (a) A hypermedia link between selections in two base documents. An annotation is attached to the link, and links and annotations are stored in a link database; (b) An SI document with marks into two base documents. Annotations are maintained as SI

The most widely used hypermedia system, the World Wide Web (or, "the web"), uses a slightly different approach than what we have described thus far. Specifically, links on the web are uni-directional, and are *embedded* in the document that originates a link. This approach is in contrast to Nelson's position [127] that a hypertext link should be bi-directional and be stored separate from the linked documents.

In the SPARCE approach, a mark describes one endpoint—a base selection—of a *potential* link. An actual link is created when a mark is associated with an SI element (such as a Sidepad item), and the link always points to the base selection. Two base selections may be indirectly linked by associating marks to the base selections with the same SI element. The first annotation in Figure 3.11(b) shows such a link. However, this link does not facilitate navigation from one base selection to another.

SPARCE offers flexibility about where mark descriptors are stored. A descriptor may be stored in a descriptor repository, similar to a hypermedia system storing a link specification in a link database. Alternatively, an SA may choose to store a mark descriptor along with SI.

Most hypertext systems support only the activation operation on links; they do not support retrieval of context information (such as text excerpts). In contrast, SPARCE provides a means to represent and retrieve rich context information for the referenced base selections. The ability to retrieve context information allows an SA user to examine base selections without activating the (complete) containing document. As described in Chapter 5, it also enables declarative querying of the combined SI and context information.

In the rest of this sub-section, we describe two hypermedia systems, IRIS and Dexter, and compare them to SPARCE.

3.7.3.1. IRIS Hypermedia Services

IRIS Hypermedia Services [55] is a set of services over cooperative applications designed originally as a part of the Intermedia hypertext system [181]. IRIS includes five Intermedia applications—a text editor, a graphics editor, an image viewer, a 3D object viewer, and a timeline editor—specially designed to facilitate creation of hypermedia networks. These applications are called *source applications*, and a document created in one of these applications is called a *source document*. New source applications may be developed using Intermedia's framework.

Each source application contains a UI element to create a link between *anchors* (that is, selected regions) in source documents. The linked anchors may be in the same document or in different documents (of the same type or different types). When a source document is opened, the source application visually indicates anchors that participate in links. Users may select any anchor and follow a link to see another anchor in context. The source document containing the other anchor is opened automatically, if it is not already open. The link creation process has four steps: Create an anchor, start link, create another anchor, and complete link. Links are binary and bidirectional. The link structure does *not* accommodate annotations.

IRIS is designed to create links between sub-documents (via anchors). An entire document may be linked only by creating an anchor that covers the entire document. In contrast, with SPARCE, an SI element may reference any context-aware object, which may be a mark, a document, or an application.

An IRIS anchor is described using two pairs of integers: One pair describes the position of the anchor's beginning within a document; another describes the anchor's extent. (The domain of these integers varies by source application. For example, the integers denote character positions for a text editor, but they indicate screen coordinates in case of a graphics editor.) This anchor structure suffices for the addressing schemes the five source applications included in IRIS use, but it may be insufficient (or inconvenient) for other applications. For example, it can be challenging to describe a span in an HTML document using this structure. In comparison, SPARCE does not fix a structure for descriptors, thereby allowing each context-agent implementer to choose the best structure for the addressing schemes he supports.

IRIS stores link specifications and anchor descriptions in a relational database. This database partitions links and anchors into *webs*. A user may create any number of webs, but can work with only one web at a time. Also, a web must be open in order to create or follow links. (An IRIS web can be viewed as an instance of a particular SI model.) IRIS includes a browser to view webs.

IRIS requires that source applications store the source documents they create in a special file-system folder. All changes to a source document pass through IRIS so that the link and anchor descriptions are kept consistent. This approach can be quite expensive because anchor locations can change frequently in an interactive editing process (for example, when editing a word-processor document).

SPARCE allows an SA to decide where it stores mark descriptors, and it does not require base documents to be stored in specific locations. It does not attempt to keep mark descriptors consistent with base documents, but descriptors may be changed when necessary to reflect changes to the base layer.

3.7.3.2. Dexter

Dexter [57] is a hypertext reference model resulting from a series of discussions among designers of hypertext systems such as NoteCards [56] and Intermedia [181]. The goal behind the model is to define common abstractions that make it easy to build and compare hypertext systems. The Dexter model is specified formally using the Z notation [146].

A Dexter hypertext network is composed of *components*, which are basic units of storage. A component may be one of three types: *atomic* (that is, primitive), *composite* (which is a directed acyclic composition of other components), and *link* (which is a relationship between components). A component is associated with a *presentation specification* that guides the component's display. Also, each component has a unique identifier (UID) that distinguishes it across space and time.

An *anchor* specifies a part of a component's contents. It is represented using an *id-value* pair. The id is a natural number unique within the anchor's component. The value of an anchor is an address of an item contained within its component. The exact format and content of an address can vary among content types.

A *link* relates components. An endpoint of a link is specified using a component specification and an anchor id. An endpoint may also include a presentation specification so that the component can be displayed in a manner appropriate to the relationship intended. A link may have two or more endpoints and its directionality is configurable.

Halasz and Schwartz [57] discuss specifying a sub-document as an endpoint (using a component and the id of an anchor within that component), but they do not discuss specifying a document (that is, just a component) as an endpoint. We assume that a document could be made a link's endpoint by omitting anchor id, or by using a special anchor id.

A component is stored as one unit together with its *attributes* (which are name-value pairs), presentation specification, a list of anchors, and the actual content of the component. Storing the list of anchors as a part of the component specification means the component is altered whenever a new anchor is created, or whenever an existing anchor is changed or deleted.

A hypertext system in the Dexter model is comprised of three layers: a storage layer, a within-component layer and a runtime layer. The *storage layer* manages a database of components. The *within-component layer* interprets contents of components for purposes such as *anchoring* (that is, addressing information inside a component). The *runtime layer* displays components according to presentation specifications.

The bulk of the Dexter model focuses on the storage layer. This layer defines an *accessor* function that maps a UID to a unique component. It also defines a *resolver* function to map a component specification to zero or more UIDs of components. (A *component specification* is a filter over the set of stored components.)

To preserve consistency of links and anchors, Dexter requires all changes to a component to pass through both the within-component layer and the storage layer. This approach to data consistency is similar to that of IRIS.

As described by Halasz and Schwartz, a Dexter hypertext system requires all components to be stored in its database. However, Hardman and others [58] state that a component descriptor in the database may point to an external source such as a disk file or a web page that supplies the actual component content. We believe that Dexter cannot guarantee consistency of anchors into such a component because changes to the contents are not guaranteed to pass through the storage layer.

The Dexter run-time layer is analogous to an SA, but it has more responsibilities than an SA: It is responsible for presenting both hypertext networks and the components in a network. By comparison, an SA is primarily responsible for presenting SI, and base applications typically display base selections. However, an SA can itself display context information it retrieves from base parts (via context agents).

3.7.4. Web-based Annotation Systems

A *web-based annotation system* is a system that uses the web infrastructure to facilitate annotations of resources on the web. In the late 1990s and early 2000s, several such annotation systems existed (for example, CritLink [182]), but only one of them, Annotea [78], is *reliably* available for use today.

Many of the web-based annotation systems allow annotations only over HTML pages. With these systems, in general, a user creates a new annotation by first selecting a region of an HTML page and then selecting a UI element that has been injected into the page, or has been added into the user's web browser. When the user loads an HTML page, the base page is modified to indicate annotations, or the annotations are displayed in a separate area of the page or in a separate window.

A variety of approaches have been used to build web-based annotation systems. The most popular approaches are: using a proxy web server, using custom extensions to existing web browsers, and using a custom web browser. A system using a proxy web server (for example, CritLink) requires users to access the web page to be annotated via a specific web server that acts as a proxy for the web server that holds the annotated page. The proxy server serves up the page to be annotated with appropriate modifications to view, create, and modify annotations. User annotations are stored on the proxy server.

The proxy web server approach has the advantage that users need only a web browser, and no other special software. These systems can be developed and maintained more easily than those involving custom web browsers (or browser extensions), but their capabilities are limited due to the use of (and dependence on) HTML.

Some web-based annotation systems such as Third Voice (now defunct [84]) extend existing web browsers using custom plug-ins. This approach requires users to install custom plug-ins, but it allows the annotation system to provide a richer UI and better annotation capabilities. It also makes it possible to store annotations on a user's local file system, or on a remote server.

Another web-based annotation system, Annotea [78], allows attaching annotations to documents addressable using a URI. It uses the Resource Description Framework (RDF) [140] to define annotation schemas, uses the XPointer framework [169] to reference annotated regions of documents, and employs HTTP [45] to transport data. A user may choose to store annotations locally or on remote annotation servers.

Annotea is not designed for use with a specific web browser, but an implementation is integrated into the Amaya web-page editor [9]. Annotea can be implemented for use with other browsers, or it may be implemented as a stand-alone application.

Annotea annotations are limited to XML documents due to the XPointer framework. Annotea's data model and annotation schemas are also fixed. SPARCE does not have these limitations. Like SPARCE, Annotea does not attempt to maintain the consistency of sub-document addresses.

3.7.5. Multivalent Document Model

In the *Multivalent Document Model* (MVD) [135, 137], a document of any type is represented in an intermediate tree form, and the document is viewed and annotated in a universal browser called the *MVD Browser*. Each node in the intermediate tree for a document has a unique identifier (ID).

The annotations for a base document are stored in a "hub" document separate from the base document. Each annotation is associated with the IDs of the annotated tree nodes.

(An annotation may span multiple tree nodes.) To increase the robustness of a sub-document address, MVD includes an excerpt of the annotated region and some structural information, along with node IDs [136]. For example, when a text selection in a word-processor document is annotated, the text of the annotated region and the name of the section and the paragraph that contains the annotated region are also saved.

An annotation is displayed using one or more *behaviors*, which are pieces of software executed according to a series of protocols. Behaviors also decide what operations are permissible on an annotation and on an annotated part of the base document.

A key difference between MVD and our approach is that an MVD "hub" document, which is analogous to an SI document, contains the annotations for only one base document. In our approach, an SI document may contain annotations and other SI for any number of base documents; many SI documents may contain annotations over the same base document; and a single SI element (for example, a link) can span multiple base documents.

The behaviors MVD employs to display and operate on annotations could be viewed as "annotation agents", *a la* context agents in SPARCE, but the annotation agents are not reusable in the same manner as context agents. (Section 3.6.3.2 reviews reusability and other attributes of context agents.)

MVD assumes that all base documents can be represented in its internal tree form. This assumption may not be valid for all document types, or it may not be efficient to prepare a tree representation for all types of documents. Also, multiple tree representations may be possible for some documents, each with different strengths and weaknesses. For example, a plain text file can be mapped to different tree structures: It can be represented as a tree with one node containing the entire text, or it can be represented as a bushy tree with one leaf node for each line of text.

MVD assumes that a single document browser suffices to view and edit all documents. It also uses a single format for the hub documents to store annotations. In contrast, our approach employs existing base applications, letting SA users employ the UI base applications offer, and allowing SA developers to choose the data model that is best for their SI. We also believe (and have demonstrated in Section 1.3) that different SAs and different SI models are needed to serve different user goals.

MVD's position on robustness of sub-document addresses is similar to ours. It too promotes the use of immutable identifiers and proposes use of context information to increase the likelihood that an intended sub-document is found when a base document changes.

3.7.6. Compound Documents

A *compound document* is a document created by combining new information with parts of existing information. A *compound document system* is a collection of cooperative applications that follow a set of protocols to display, print, and store data. In this system, existing information parts appear in a compound document as if they are an integral part of the result document. For example, a research paper can be composed as a compound document. In this case, much of the paper's text would likely be written directly in the compound document, but graphs and charts are likely inserted into the compound document from existing spreadsheets. Similarly, figures can be inserted from existing image files. (Each chapter of this dissertation is composed as a compound document.)

A compound document is created in a *host application* that is responsible for providing the overall document UI. *Source applications* supply the data for different parts of the compound document, and render the parts within the document UI. When a user selects a document part, the host application interacts with the appropriate source application and presents to the user a list of operations possible on the part; the source application carries out the action the user selects. "Activating" a part included in a compound document opens the source document that the part belongs to in its original application, and "highlights" the part.

OLE 2 [18] and OpenDoc [132] are the best known compound document systems, with the former probably being in wider use. In the rest of this sub-section, we provide an overview of the OLE 2 compound document system and compare it to SPARCE.

An existing information part may be *embedded* or *linked* in an OLE 2 compound document. Embedding makes a copy of the source part, but linking retains a link (called a *moniker*) to the source rather than making a copy. (Both approaches produce the same visual result in the compound document.) Also, the content of a linked part is updated in the compound document each time the compound document is opened. Due to their similarity with marks, we discuss only linked parts in the rest of this section.

An application capable of creating an OLE 2 compound document is called a *container*. An application that is capable of supplying data to a compound document is called a *server*. An application may be both a container and a server. For example, MS Word is both a container and a server, but Adobe Acrobat is only a server [7]. To import data into a compound document, a user copies the data in a server application to the clipboard and then pastes the data into the compound document. At the time of pasting the data, the user may decide either to embed the data or to create a link.

The OLE 2 compound-document protocols require container and server applications to use *Compound Files*, a technology for persisting compound data. This technology provides a means to treat a file-system file as a collection of storages and streams. A *storage* element is analogous to a file-system directory; a *stream* element is analogous to a file. A storage element may contain streams and other storages.

To store a compound document, a container application first opens a file-system file using traditional, OS-provided file-system functions. When external data is imported into the compound document, the container creates a storage element in the document file and passes the storage to the server application responsible for the imported data. That server first creates a stream in the storage, and writes the moniker corresponding to the linked data into the stream. The server may also write an image version of the data for quick drawing when the compound document is loaded again (so as to avoid the possible delay in invoking the server application to draw the linked data).

A server application may provide the content for a linked part in multiple formats. For example, MS PowerPoint can supply the content of a linked slide as a Slide object or as a picture. When content is available in more than one format, the container or the user may choose a display format for the content. However, once given a format and a display area, the server has complete control over what information is displayed and how it is displayed in the given area.

The OLE 2 compound document system and SPARCE differ in several aspects. First, OLE 2 allows retrieval of only content from a linked part, and the container application has no control over how much data is retrieved or how the retrieved data is drawn. With SPARCE, an SA may retrieve any subset of the context information available for a mark, and display it in any manner. For example, an SA may retrieve the text of an MS Word selection and draw the text in any color. Alternatively, it may retrieve both text and color information, and draw the text in the retrieved color.

OLE 2 imposes a storage model for compound documents. A controller application has little control over how monikers are written to a file because the server applications write the monikers. SPARCE does not impose such constraints on SAs.

Developing OLE 2 container and server applications can require large development efforts. By one account [16], supporting compound documents requires implementing 13 interfaces and 126 functions (to support both container and server features). In contrast, SAs do not need to implement any particular interface, and a context agent needs to implement only four functions (listed in Section 3.6.2.7).

3.8. Summary and Conclusions

This chapter has described a flexible representation scheme for descriptors of base parts. The scheme allows a context-agent implementer to choose a descriptor structure appropriate to his needs, yet allows SAs to represent the use of marks in any data model. The URI representation of base-part references enables the use of marks in traditional applications such as web browsers and word processors, without any change to those applications.

This chapter has also described SPARCE, our middleware architecture to create base-part references, and to activate base parts and retrieve context information from the parts. The chapter has also presented an evaluation of the representation schemes for descriptors and of the middleware. The evaluation shows that support for referencing information in different base types is added easily. It also shows that SAs and context agents can evolve independently due to the abstractions SPARCE defines.

SPARCE is closely related to hypermedia systems, annotation systems, the multivalent document model, and compound document systems. These systems support subsets of the features SPARCE supports, but none supports all of SPARCE's features. No system provides the freedom SPARCE does in modeling of annotations and other information similar to SI. No system supports retrieval of context information from the base layer. The ability to access context information allows us to combine SI with base information, and to transform the combination to other forms. Chapters 6, 7, and 9 show how others parts of SASS employ SPARCE to support such transformation.

The next chapter builds on the descriptor representation schemes introduced in this chapter to model bi-level information in conceptual and logical data models.

4. Modeling Bi-level Information

This chapter describes a methodology to model bi-level information in the Entity-Relationship (ER) model [25] at the conceptual level, and in the relational [41] and XML [43] models at the logical level. A developer of a superimposed application (SA) can use the methodology to prepare a conceptual schema for only the superimposed information (SI) and indicates which parts of SI are associated with marks. The methodology includes a means to automatically extend the SI schema to include mark descriptors and context information, thus modeling *bi-level information*. The methodology also includes procedures that can automatically generate logical bi-level information schemas from a conceptual bi-level information schema. Instances of the bi-level information schemas prepared using our methodology can be declaratively queried using languages such as the Structured Query Language (SQL) [92] and XML query languages.

We present our methodology in three parts: First, we model marks and the use of marks (in Sections 4.3-4.5). Second, we model mark descriptors (in Section 4.6). Finally, we model context information (in Section 4.7). With each part, we present details of generating relational and XML schemas from ER schemas.

Section 4.8 demonstrates the ability to express declarative queries over bi-level information. Section 4.9 presents an evaluation of the methodology in the form of bi-level information schemas generated for three SAs with distinct information needs. Section

4.10 reviews four related systems and compares those systems to our methodology. Section 4.11 summarizes the chapter and presents some concluding remarks.

We begin the chapter with an introduction to the need for modeling bi-level information (in Section 4.1) and a description of a motivating example (in Section 4.2).

4.1. Introduction

An SA is different from a traditional application in two key respects: First, at run time, it uses marks to reference base-layer contents. Specifically, the SA associates marks with superimposed information (SI) elements. Second, at storage time, the SA includes mark identifiers or mark descriptors with SI elements (as described in Section 3.2). That is, the design of an SA must include representations for the use of marks in both the run-time model and the storage model of the SA.

In Section 3.2.4 (see specifically Figure 3.7), we presented a means to represent the use of marks in the run-time model of an SA. In this chapter, we present a means to represent the use of marks in the storage model—more precisely, in the information model—of an SA.

Earlier in this dissertation research, we used (and were satisfied with) ad-hoc means to represent the use of marks in the information model of an SA, but as the number of SAs grew (we know of nine SAs built thus far with our infrastructure: six due to us, and three due to our collaborators; see Section 3.6.1.3), we realized that SA developers would benefit from a systematic means to represent the use of marks. A systematic means would take into account different uses of marks, and provide uniform syntax and semantics to represent these uses of marks.

In this chapter, we describe a means of systematic conceptual modeling of the use of marks in the ER model. We model the use of marks at the conceptual level so that the resulting SI schemas are independent of logical data models (such as the relational model and the XML model). We use the ER model because it is widely used for conceptual information modeling, and SA developers are likely to be familiar with it. Additionally, by using the ER model, we are able to leverage existing procedures to automatically generate SI schemas for the relational and XML models, which eases the transition from the SA-design phase to the implementation phase.

An obstacle to representing the use of marks in the ER model, and in models like the ER-model, is that the native model constructs are not expressive enough. Specifically, they cannot express the layer-crossing property of marks. A solution would be to develop new models or modeling constructs that represent the use of marks. However, existing design methodologies and tools might not work with the extended or new model. An alternative is to use existing constructs as they are, but develop conventions to indicate the use of marks. We pursue the latter alternative.

We identify the different patterns of use of marks, and provide a set of conventions to apply the patterns in a flexible and expressive manner. The patterns we identify allow an SA developer to accomplish the following information modeling tasks at design time. In this list and in the rest of the chapter, we use the term *attribute* to mean either an ER entity's attribute or an ER relationship's attribute.

- Associate marks with entities, attributes, and relationships.
- Assign the excerpt obtained from a mark as the value of an attribute.
- Impose cardinality and other constraints when associating marks.
- Generate schemas for the relational and XML models from ER schemas.

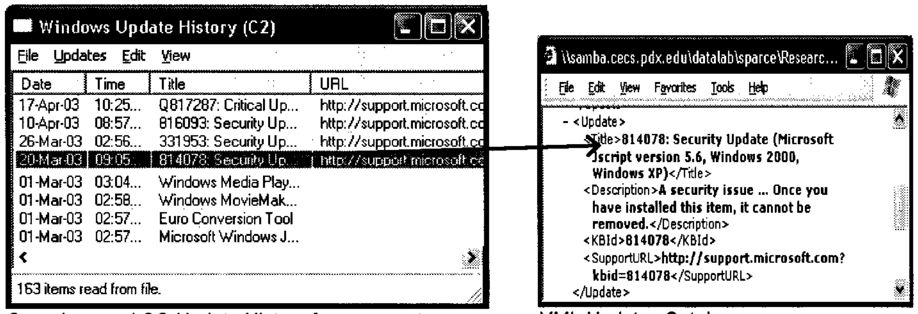
To use the patterns, the SA developers need not be aware of the information model of any base application.

The patterns make it easy to exploit the context mechanism of SPARCE and provide a way to combine SI with context information. The combined bi-level information may then be queried using structured query languages such as SQL in the relational model, and XQuery [176] or XSLT [177] in the XML model. Section 4.8 shows some example queries over bi-level information. Chapters 5 and 9 discuss execution of queries over bi-level information.

4.2. Motivating Example

In this section, we describe the *Superimposed System-Information Browser* (SSIB), an SA developed using SPARCE. We describe the information needs of SSIB and present a traditional ER schema for SSIB information. In Sections 4.3–4.7 we use our methodology to model bi-level information to express the information needs of SSIB.

SSIB allows users to browse information such as operating system (OS) updates, and application and OS events, for a collection of computers. System administrators can use this application to browse information resident on networked computers for diagnostic purposes.


Windows Update History (C2)
File Updates Edit View
\\samba.cecs.pdx.edu\datalab\sparce\Researc...
File Edit View Favorites Tools Help

Date	Time	Title	URL
17-Apr-03	10:25...	Q817287: Critical Up...	http://support.microsoft.co
10-Apr-03	08:57...	816093: Security Up...	http://support.microsoft.co
26-Mar-03	02:56...	331953: Security Up...	http://support.microsoft.co
20-Mar-03	09:05...	814078: Security Up...	http://support.microsoft.co
01-Mar-03	03:04...	Windows Media Play...	
01-Mar-03	02:58...	Windows MovieMak...	
01-Mar-03	02:57...	Euro Conversion Tool	
01-Mar-03	02:57...	Microsoft Windows J...	

163 items read from file.

```
- <Update>
    <Title>814078: Security Update (Microsoft
      Jscript version 5.6, Windows 2000,
      Windows XP)</Title>
    <Description>A security issue ... Once you
      have installed this item, it cannot be
      removed.</Description>
    <KBId>814078</KBId>
    <SupportURL>http://support.microsoft.com?
      kbid=814078</SupportURL>
  </Update>
```

Superimposed OS Update History for a computer *XML Updates Catalog*

Figure 4.1: System information displayed in SSIB. OS-update information displayed on the left, with a mark into an XML document on the right

Figure 4.1 shows some OS-update information displayed in SSIB. The window with the caption ‘Windows Update History (C2)’ displays a table structure superimposed on OS-update information for computer `C2`. The highlighted row shows the details of one OS update applied on that computer, excerpted from a set of marks. For example, the title of this update is retrieved using a mark into a shared catalog of available updates (called the *Updates Catalog*) stored on the network, shown on the right side of Figure 4.1. Though not shown in the figure, the highlighted row also contains support details such as a reason for the update and the underlying problem that necessitated the update. These details are retrieved using marks into HTML documents [61] available on the Microsoft (MS) Support web site [100]. Table 4.1 describes these and other sources that SSIB uses to display system information.

Modeling SSIB information as SI provides several benefits. It integrates disparate and distributed information without replication. It also allows structured querying over base information of varying structures. For example, an administrator can ask to see a timeline of errors on computer c2 since the last update related to MS Outlook [96] was applied on that computer. Answering this query requires looking up the support pages to discover which updates apply to MS Outlook, choosing the last such update on computer c2, and looking up error reports on computer c2 that occurred after that update. The query returns the date, time, and description of relevant errors.

Table 4.1: Base sources SSIB consults

Info. Kind	Doc. Type	Location	Description
Event log	MS Excel	Distributed	Records OS and application events, typically one event per row. Obtained using the *Event Log Viewer* built into MS Windows. Three log files per computer.
Error reports	MS Word	Distributed	Records OS and application errors. Obtained using the *System Information Viewer* built into MS Office [97]; reformatted for demonstration purposes. One document per computer.
Update log	Text	Distributed	Contains one line per OS update applied. Not all available updates might be applied on a particular computer. One log per computer.
Updates Catalog	XML	Network, shared	Contains one Update XML element per available update (see Figure 4.1). One log per network.
Support details	HTML	The web	Describes symptoms, cause, and resolution related to a problem along with a list of affected applications. Available from MS Support. Each update in the updates catalog typically references a support page.

Figure 4.2 shows an ER schema for SSIB, drawn using a syntax similar to the syntax that the Unified Modified Language (UML) [159] defines for static class diagrams. Schema elements whose names are in bold have marks associated with them. The entity Observation denotes observations that computer users record about their computers. (For example, a user might record seeing an error message related to MS Outlook.)

The entity Application represents applications such as MS Outlook. A system administrator frequently uses this entity to determine which updates need to be applied on a given computer. (A support web page for an OS update typically lists the applications to which the update applies, whereas a scan of the computer reveals the applications installed on the computer.) The other entities relate to the base sources listed in Table 4.1.

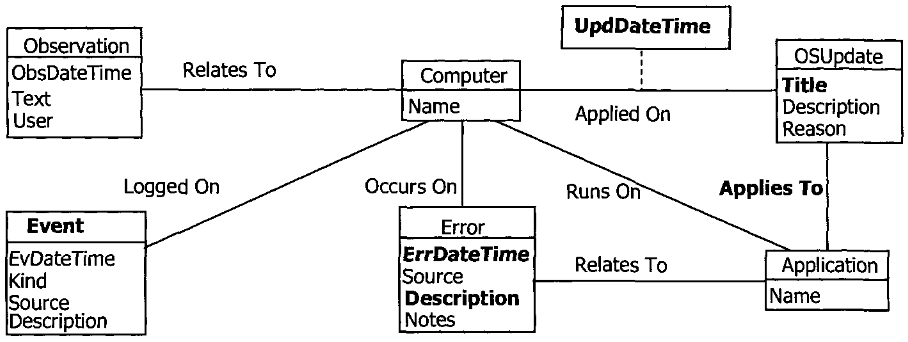


Figure 4.2: A conceptual schema for SSIB. Names in bold indicate elements with associated marks. All relationships are many-to-many; all entities have a key attribute named ID (not shown).

In the rest of this chapter we show how the mark associations in the schema of Figure 4.2 are expressed using our methodology. Unless stated otherwise, all examples in this chapter are based on this schema.

4.3. Modeling Marks and Use of Marks

We model a mark as the ER entity Mark. This entity has a key attribute named ID. Its other attributes are derived from mark descriptors (described in Chapter 3), but we omit those attributes at this stage because they are immaterial to modeling the use of marks. Section 4.6 describes modeling of mark descriptors.

We model different uses of marks as *relationship patterns* [114], which capture recurring needs or problems when establishing relationships (at design time) among information elements. (Section 4.10.1 reviews the general notion of relationship patterns.) We define a relationship pattern for each type of schema element with which marks may be associated: entity, entity attribute, relationship, and relationship attribute. Deriving attribute values from the text excerpt of a mark forms another pattern.

The relationship patterns we identify have the following informal signature:

```
<pattern>:<type> (<parameters>)
```

In this signature, <pattern> is the name of the pattern, <type> is the name of the relationship type (that represents a use of mark) as chosen by the SA developer, and <parameters> indicates attribute names, when they are needed by the pattern. A relationship pattern that represents the use of marks relates an entity of type Mark to non-Mark entities or to relationships of any type. We call a non-Mark entity type a *regular entity type* or an *SI entity type.* A relationship between regular entities is a *regular relationship.*

In the rest of this section, we describe the five relationship patterns we have identified to represent the use of marks. For each pattern, we state its signature, describe the semantics, and list the constraints on using the pattern.

4.3.1. Associating Marks with Entities

The EMark pattern associates marks with regular entities. Figure 4.3 shows the use of this pattern to associate a mark with an Event entity. EMark is the name of the relationship pattern and EventDetail is the relationship type; Logged On is a regular ER relationship type.

Signature: EMark:<type>. A relationship of EMark pattern has no parameters.

Semantics: The EMark pattern associates marks with *entire* entities, not with any particular set of entity-attributes, and no specific meaning is attached to this association. Instead, the developer interprets this association. For example, the developer might incorporate the excerpt extracted from the mark into the user interface of an SA.

Constraints: The EMark pattern may be used to associate marks with any SI entity type. The developer may impose any cardinality constraint on EMark relationships. The schema in Figure 4.3 restricts the cardinality of the EventDetail relationship type to one because an event logged on a computer has just one associated mark in the SSIB application.

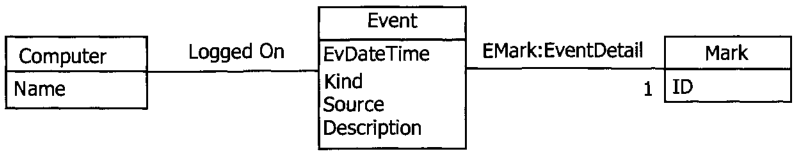


Figure 4.3: Associating marks with an entity

4.3.2. Associating Marks with Entity Attributes

The AMark pattern associates marks with attributes of an entity. Figure 4.4 shows two relationship types that associate marks with two attributes defined by the entity type Error. The relationship type ErrorTime associates the attribute ErrDateTime with a mark. The relationship type ErrorDetails associates the attribute Description with a mark. Occurs On is a regular ER relationship type.

Signature: AMark:<type>(a_1, a_2, ..., a_n), where a_1, a_2, ..., a_n ($n>0$) are distinct attributes of an SI entity.

Semantics: All attributes specified are associated with the same mark (or the same set of marks if cardinality is greater than one). Associating a mark with an attribute *does not* mean its value is obtained using the mark. Rather, it gives an SA access to excerpt and other context information of the associated mark(s), in addition to having an attribute value stored in the superimposed layer. For example, an SA may display a retrieved excerpt as a "tool tip" upon mouse rollover. An SA may also activate the associated mark.

Constraints: An AMark relationship type is always a binary relationship between an SI entity type and the Mark entity type. At least one attribute must participate in the relationship. The developer may impose any cardinality constraints. The schema in Figure 4.4 restricts the cardinality of the relationship types ErrorTime and ErrorDetails to exactly one mark to satisfy the SSIB application needs.

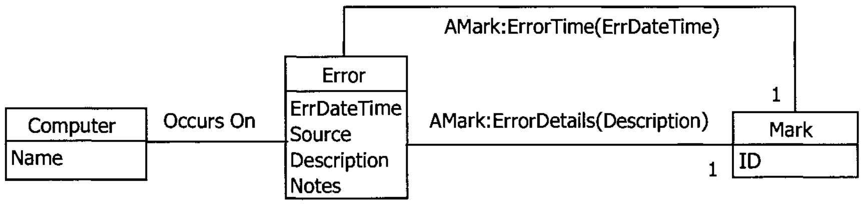


Figure 4.4: Associating marks with entity attributes

4.3.3. Deriving Attribute Values

We define the pattern AExcerpt to derive an attribute's value from the excerpt of a mark. Figure 4.5 shows a relationship type in the AExcerpt pattern to set the value of the attribute Title as the excerpt of a mark. (It is possible to define a more general pattern such as AContext to derive an attribute's value from any context element of a mark, but, for simplicity, we limit this discussion to excerpts. Chapter 7 discusses deriving a value from any part of a mark's context.)

Signature: AExcerpt:<type>(a), where a is an attribute of an SI entity.

Semantics: The value of the attribute associated with a mark using this pattern is a function of the excerpt obtained from the mark. We assume that appropriate type conversion is performed before assigning the derived value to an attribute.

Constraints: Like an AMark relationship type, an AExcerpt relationship type is always binary: between an SI entity type and the Mark entity type. Assuming that attributes are single-valued, the attribute in an AExcerpt relationship type may be associated with at most one mark. (Chapter 7 discusses deriving an attribute's value from more than one mark.)

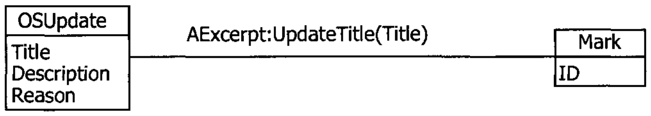


Figure 4.5: Deriving the value of an entity's attribute from a mark's excerpt

4.3.4. Associating Marks with Relationships

We use the RMark pattern to associate marks with relationships. Figure 4.6 shows a relationship type of this pattern that associates zero or more marks with relationships of the type Applies To. We use the term *anchored relationship* [17] to refer to a relationship with which marks are associated. In Figure 4.6, the relationship Applies To is anchored. We aggregate the anchored relationship (as indicated by a dashed rectangle around the relationship type [139]) to clarify that marks are associated with the relationship.

Signature: RMark:<type>. A relationship of RMark pattern has no parameters.

Semantics: The RMark pattern associates marks with entire relationships.

Constraints: A relationship of any type may be anchored (including another RMark relationship type). There are no constraints on the degree of the anchored relationship type, but an RMark relationship itself is always binary. That is, it relates an anchored relationship type with the Mark entity type. There are no constraints on the cardinality of either the anchored relationship type or the RMark relationship type, and either type may define attributes.

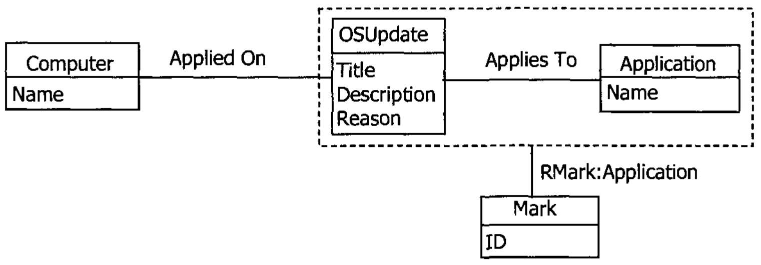


Figure 4.6: Associating marks with a relationship

4.3.5. Associating Marks with Relationship Attributes

The RAMark pattern associates marks with attributes of a relationship. Figure 4.7 shows a relationship type that associates marks with the attribute UpdDateTime of an Applied On relationship. We aggregate the anchored relationship type Applied On to clarify that marks are associated with the relationship's attribute.

Signature: RAMark:<type>(a_1, a_2, ..., a_n), where a_1, a_2, ..., a_n (n>0) are distinct attributes of a relationship.

Semantics: The semantics of the RAMark pattern are similar to that of the AMark pattern. All attributes specified are associated with the same mark (or the same set of marks if cardinality is greater than one). Associating a mark with an attribute does *not* mean its value is obtained using the mark.

Constraints: The RAMark pattern imposes constraints similar to those the RMark pattern does. The attributes of any relationship (including an RMark or RAMark relationship) may be associated with marks. There are no constraints on the degree of the anchored relationship type, but an RAMark relationship type itself is always binary. There are no constraints on the cardinality of either relationship type, and either type may define attributes.

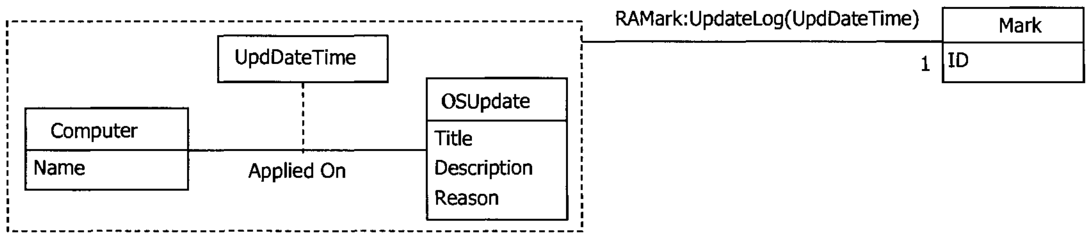


Figure 4.7: Associating marks with a relationship attribute

4.4. Generating Relational Schemas

Having covered all the patterns of use of marks employed in Figure 4.2, we now define the procedures to convert the relationship types (defined using the patterns) to relational schemas. We present relational schemas in the form of Data Definition Language (DDL) statements using SQL:1999 [92].

We represent the Mark entity type as the relation `Mark` with the key attribute `ID` (see Figure 4.8). Section 4.6 discusses the representation of other attributes of this relation.

```
CREATE TABLE Mark
(
 ID INTEGER NOT NULL PRIMARY KEY, ...
)
```

Figure 4.8: Partial relational schema for the Mark entity type

The relational schemas generated for the different patterns of use of marks involve the `Mark` relation. Specifically, the schema generation procedures generate relations that reference the attribute `Mark.ID`. The procedures to generate relational schemas are based on the procedure defined by Elmasri and Navathe [41]. In the rest of this chapter, we call their procedure the *traditional procedure.* (Briefly, the traditional procedure translates each entity to a relation and each of the entity's attribute to an attribute of the relation generated for the entity. Based on the cardinality constraints, a relationship is represented either as an attribute in a participating entity's relation, or as a separate relation.)

Figure 4.2 omits the key attribute named ID from all entity types, but we add that attribute to the relational schema generated for the entity types. We also assign a reasonable data type to each attribute.

4.4.1. Generating Schemas for the EMark and AMark Patterns

We use the traditional procedure to generate relational schemas for EMark and AMark relationship types. Figure 4.9(a) shows the relational schema for the entity type Event and the relationship type EventDetail of Figure 4.3. The attribute `EMark_EventDetail` stores the mark associated with an event.

```
CREATE TABLE Event
(
 ID INTEGER NOT NULL PRIMARY KEY,
 EvDateTime TIMESTAMP, Kind CHAR(5), Source VARCHAR(25),
 Description VARCHAR(255),
 EMark_EventDetail INTEGER REFERENCES Mark(ID)
)
```

(a)

```
CREATE TABLE Error
(
 ID INTEGER NOT NULL PRIMARY KEY,
 ErrDateTime TIMESTAMP, Source VARCHAR(25),
 Description VARCHAR(255), Notes VARCHAR(255),
 AMark_ErrorTime INTEGER REFERENCES Mark(ID),
 AMark_ErrorDetails INTEGER REFERENCES Mark(ID)
)
```

(b)

Figure 4.9: Relational schema generated for EMark and AMark relationship types. (a) Schema for the Event entity type and EMark relationship type of Figure 4.3; (b) Schema for the Error entity type and AMark relationship types of Figure 4.4

Figure 4.9(b) shows the relational schema generated for the Error entity type and the AMark relationship types of Figure 4.4. The last two attributes represent the AMark relationship types.

4.4.2. Generating Schemas for the AExcerpt Pattern

We generate the relational schema for a relationship type of the pattern AExcerpt in two steps: First, we generate the schema for a stored relation. Then we define a *view* (that is, a relation derived from other relations) over the stored relation in order to provide direct access to the excerpt retrieved from the base layer.

To generate the schema for a stored relation, we generate the schema for the entity type involved in the AExcerpt relationship type using the traditional procedure, and *remove* from the generated relational schema the attributes that participate in the AExcerpt relationship type.

Figure 4.10(a) shows the relational schema the traditional procedure generates for the entity type OSUpdate in Figure 4.5. Figure 4.10(b) shows the relational schema generated for the stored relation after removing the attribute `Title` because the value of that attribute is derived from a mark's excerpt. Note that the foreign-key attribute that represents the use of mark is present in both schemas.

```
CREATE TABLE Traditional_OSUpdate
(
 ID INTEGER NOT NULL PRIMARY KEY,
 Title VARCHAR(100),
 Description VARCHAR(255),
 Reason VARCHAR(255),
 AExcerpt_UpdateTitle INTEGER REFERENCES Mark(ID)
)
```

(a)

```
CREATE TABLE Stored_OSUpdate
(
 ID INTEGER NOT NULL PRIMARY KEY,
 Description VARCHAR(255),
 Reason VARCHAR(255),
 AExcerpt_UpdateTitle INTEGER REFERENCES Mark(ID)
)
```

(b)

Figure 4.10: Relational schema generated for an AExcerpt relationship type. (a) Traditional schema generated for the entity type participating in an AExcerpt relationship; (b) Schema generated for the stored relation for an entity type participating in an AExcerpt relationship

In the second step, we define a view over the stored relation. The view exposes *as is* the attributes whose values are not derived from a mark's excerpt, but hides the attribute that references the attribute `Mark.ID` (corresponding to the attribute whose value is derived from a mark's excerpt). Instead, the view exposes the excerpt obtained from the hidden mark ID attribute.

Figure 4.11 shows the definition of the view over the stored relation for the entity type OSUpdate in Figure 4.5. The view exposes the attributes `ID`, `Description`, and `Reason` as they are because their values are not derived from marks' excerpts. The view exposes the attribute `Title` as a result of the function `excerpt`. The function `excerpt` accepts a mark ID and returns the text excerpt (a string) retrieved from the corresponding mark. This function may be implemented as a user-defined function [147] by reusing the context mechanism of SPARCE (described in Section 3.3.2).

In the view definition that our procedure generates, the value of an attribute associated with an AExcerpt relationship is the same as the excerpt retrieved from a mark, but an SA developer may change the generated schema. For example, he might make the attribute's value only the first 10 characters of the excerpt.

```
CREATE VIEW OSUpdate (ID, Title, Description, Reason)
AS
SELECT ID, excerpt(AExcerpt_UpdateTitle), Description, Reason
FROM Stored_OSUpdate
```

Figure 4.11: View definition generated for an entity type participating in an AExcerpt relationship

Type conformance is an important consideration when assigning an attribute value from a retrieved excerpt. As described, our procedure to generate relational schemas for the AExcerpt pattern assigns a value of type string to any attribute that derives its value from a mark's excerpt. Although the string type might satisfy many modeling needs, it is necessary to consider representing excerpts as other types (such as integer and date).

An improvement to our procedure is to cast the result of the function `excerpt` to a type compatible with the type of the attribute that participates in an AExcerpt relationship type.

*4.4.3. Generating Schemas for the **RMark** Pattern*

We generate the relational schema for an RMark relationship type in two steps. In the first step, we generate the schema for the anchored relationship type using an appropriate procedure. That is, we use the traditional procedure if the anchored relationship is a regular ER relationship between SI entities; we use one of the procedures in this section if the relationship follows any of the patterns of use of marks. For example, we use the traditional procedure to generate the schema for the relationship type Applies To in Figure 4.6 because that relationship type is between SI entity types. Figure 4.12(a) shows the schema generated for that anchored relationship type (in the form of the relation `AppliesTo`). For ease of reading, we include also the schema for the related entity type Application (in the form of the relation `Application`). Figure 4.10(b) shows the schema for the other related entity type, OSUpdate.

In the second step, we augment the schema generated in the first step to represent the RMark relationship type. The augmentation procedure is based on the cardinality constraints of the RMark relationship type. If the cardinality constraints of the RMark relationship allow multiple marks (that is the relationship is 1:N or M:N), we create a new relation and perform the following actions.

1. Add the key attributes of the relation that captures the anchored relationship type, and constrain those attributes to be a foreign key.
2. Add a foreign-key attribute to reference the attribute `Mark.ID`.
3. Add the attributes of the RMark relationship.

4. Define the primary key of the new relation as the set of the foreign-key attributes.

If the RMark relationship can have at most one mark (that is the relationship is 1:1 or M:1), we perform only the aforementioned Actions 2 and 3, but with the relation that captures the anchored relationship type.

```
CREATE TABLE Application
(
 ID INTEGER NOT NULL PRIMARY KEY,
 Name VARCHAR(255)
)

CREATE TABLE AppliesTo
(
 UID INTEGER REFERENCES Stored_OSUpdate(ID),
 AID INTEGER REFERENCES Application(ID),
 PRIMARY KEY (UID, AID)
)
```

(a)

```
CREATE TABLE RMark_Application
(
 UID INTEGER,
 AID INTEGER,
 CONSTRAINT FOREIGN KEY (UID, AID)
  REFERENCES (AppliesTo.UID, AppliesTo.AID),
 RMarkID INTEGER REFERENCES Mark(ID),
 PRIMARY KEY (UID, AID, RMarkID)
)
```

(b)

Figure 4.12: Relational schema generated for an RMark relationship type. (a) Schema for the anchored relationship type of an RMark relationship type; (b) Schema for an RMark relationship type

Figure 4.12(b) shows the schema of a new relation created to represent the RMark relationship type of Figure 4.6. A new relation is created because the RMark relationship type allows any number of marks to be associated with the anchored relationship type Applies To. The schema of the new relation contains the key attributes `UID` and `AID` of the relation `AppliesTo`, the relation that captures the anchored relationship type Applies To. These attributes together are also defined as a foreign key referencing the primary key of the relation AppliesTo. The new relation also has a foreign key attribute to denote the use of a mark. The relation has no other attributes because the RMark relationship has no attributes. Finally, the set of all foreign key attributes (UID, AID, and RMarkID) is the primary key of the new relation.

4.4.4. Generating Schemas for the RAMark Pattern

Generating the relational schema for an RAMark relationship type is similar to generating the relational schema for an RMark relationship type, except that the foreign key attribute that denotes the use of a mark is associated with a relationship's attribute, not with the relationship.

```
CREATE TABLE Computer
(
 ID INTEGER NOT NULL PRIMARY KEY,
 Name VARCHAR(255)
)

CREATE TABLE AppliedOn
(
 UID INTEGER REFERENCES Stored_OSUpdate(ID),
 CID INTEGER REFERENCES Computer(ID),
 UpdDateTime TIMESTAMP,
 RAMark_UpdateLog INTEGER REFERENCES Mark(ID),
 PRIMARY KEY (UID, CID)
)
```

Figure 4.13: Relational schema for an RAMark relationship type

Figure 4.13 shows the relational schema generated for the RAMark relationship type of Figure 4.7. For ease of reading, we include also the schema for the related entity type Computer. (Figure 4.10(b) shows the schema for the other related entity type, OSUpdate.) The relation AppliedOn in this schema captures the anchored relationship type Applied On. A new relation is not needed to capture the RAMark relationship type because its cardinality is M:1.

4.5. Generating XML Schemas

In this section, we describe the procedures to generate XML schemas for the different patterns of use of marks. In XML terms, these procedures allow association of marks with elements, attributes, and text content. We describe the schemas generated for the XML model using XML Schema [170], instead of using XML Document Type Definition (DTD) [43], because the former is more expressive and permits more modular construction of schemas.

We begin this section with an overview of the procedure to generate XML schemas from ER schemas.

4.5.1. Overview of the Schema-Generation Procedure

The *representational multiplicity* (that is, the ability to express the same information in different ways) of the XML model poses special challenges when generating an XML schema from an ER schema: An ER attribute may be represented as an XML element or as an XML attribute; an ER relationship may be represented as an XML element or as an XML attribute, or using a combination of XML elements and attributes. Further, a relationship may be replicated for each participating entity, or it may be represented using a reference (for example, using an attribute of type IDREF). For reasons such as performance, an SA developer may use different representations for different applications.

Several researchers—Kleiner and Lipeck [82]; Sengupta and others [144]; Elmasri and others [42] among them—have considered the problem of generating XML schemas from ER schemas. However, they all limit the developer's choices of XML representations for ER schema elements. For example, Elmasri and others represent ER attributes as XML elements. That is, none of the current approaches to generate XML schemas from ER schemas fully handles the representational multiplicity of XML.

To leverage these and other works, and to avoid the limitations of existing procedures, we have devised a two-step procedure to generate the XML schema for a pattern of mark use. In the first step, we allow the SA developer to employ any existing procedure to generate the XML schema for the ER entity, relationship, or attribute involved in the use of mark, *excluding* the relationship type that indicates the use of marks. This step generates the schema for an XML element or an attribute (because an ER entity, relationship, or attribute can only be represented using these XML constructs).

In the second step, we add to the schema generated in the first step new XML elements (*always* elements) that represent the use of marks. The location of a new XML element that is added is determined as follows: If the first step generates an element, the new element is added as a sub-element of the element generated; if the first step generates an attribute, the new element is added as a sub-element of the element that contains the attribute generated.

Table 4.2 shows how the two steps of the procedure work together. The first column lists different ER constructs. The second column shows the relationship pattern used to represent the use of marks (in which an ER schema construct of the kind listed in the first column participates). The third column shows the XML constructs an existing procedure to generate XML schemas might generate in the first step. The last column shows the type of XML element we add to the schema generated in the first step. Figure 4.14 defines the element types we use to represent use of marks. Section 4.5.2 describes these types, including the need for the type Xml_TMark (shown in the fifth data row of Table 4.2).

Table 4.2: Correspondence of ER constructs and patterns of use of marks to XML constructs

ER construct	Relationship pattern	XML construct generated in Step 1	XML element type added in Step 2
Entity	EMark	Element	Xml_EMark
Entity	EMark	Attribute	Xml_AMark
Entity attribute	AMark	Element	Xml_EMark
Entity attribute	AMark	Attribute	Xml_AMark
Entity attribute	AExcerpt	Element	Xml_TMark
Entity attribute	AExcerpt	Attribute	Xml_AMark
Relationship	RMark	Element	Xml_EMark
Relationship	RMark	Attribute	Xml_AMark
Relationship attribute	RAMark	Element	Xml_EMark
Relationship attribute	RAMark	Attribute	Xml_AMark

Our two-step procedure allows marks to be associated with any part of an XML document (elements, attributes, and text nodes), regardless of how an ER construct is represented in XML. That is, with respect to the use of marks, our procedure fully handles the representational multiplicity of XML.

In this chapter, for simplicity, we assume that the first step in the schema-generation procedure represents ER elements and relationships as XML elements, and ER attributes as XML attributes. Consequently, we describe the procedures to generate XML schemas only for the EMark, AMark, and AExcerpt patterns. We omit discussing the procedures for the RMark and RAMark patterns (because those procedures would be the same as the procedures for the EMark and AMark patterns, respectively).

4.5.2. Element Types for Patterns of Use of Marks

The XML Schema instance document in Figure 4.14 defines the element types we use to represent association of marks with different parts of an XML document. The element types belong to the namespace "sixml" and are bound to the Uniform Resource Identifier (URI) [15] "http://schema.sixml.org". For simplicity, where possible, we use our element types without this namespace or the URI. XML Schema defines the namespace "xs" (bound to the URI "http://www.w3.org/2001/XMLSchema").

The term *Sixml* (pronounced `'siks-m&l`) [118, 120] refers to SI represented as XML. A *Sixml document* is an XML document that contains elements of the types we define for mark associations. To focus on generating XML schemas from ER schemas, we present here only the Sixml element types that can arise in our generation procedure. (The ER model cannot express all XML constructs.) Chapter 7 presents the complete set of Sixml types. Appendix A shows the XML Schema instance document containing the complete set of Sixml types. That document is also available online [119].

```xml
<xs:schema targetNamespace="http://schema.sixml.org" xmlns:sixml="http://schema.sixml.org"
                                                     xmlns:xs="http://www.w3.org/2001/XMLSchema">
 <!--Abstract base type for mark descriptors -->
 <xs:complexType name="Descriptor" abstract="true" final="" block="" mixed="true">
  <xs:complexContent mixed="true"><xs:extension base="xs:anyType"/></xs:complexContent>
 </xs:complexType>
 <!--Context information of arbitrary internal structure -->
 <xs:element name="Context">
  <xs:complexType mixed="true">
   <xs:sequence><xs:any processContents="skip"/></xs:sequence><xs:anyAttribute processContents="skip"/>
  </xs:complexType>
 </xs:element>
 <xs:element name="Descriptor" type="sixml:Descriptor"/>
 <xs:attribute name="markID" type="xs:string"/><xs:attribute name="type" type="xs:string"/>
 <xs:complexType name="Xml_EMark" final="restriction" block="#all" mixed="true">
  <xs:complexContent mixed="true">
   <xs:restriction base="xs:anyType">
    <xs:sequence><xs:element ref="sixml:Descriptor" minOccurs="0"/><xs:element ref="sixml:Context" minOccurs="0"/></xs:sequence>
    <xs:attribute ref="sixml:markID"/><xs:attribute ref="sixml:type"/>
   </xs:restriction>
  </xs:complexContent>
 </xs:complexType>
 <xs:attribute name="valueSource" type="xs:boolean"/><xs:attribute name="valueExpression" type="xs:string"/>
 <xs:complexType name="Xml_TMark" final="restriction" block="#all" mixed="true">
  <xs:complexContent mixed="true">
   <xs:extension base="sixml:Xml_EMark">
    <xs:attribute ref="sixml:valueSource"/><xs:attribute ref="sixml:valueExpression"/>
   </xs:extension>
  </xs:complexContent>
 </xs:complexType>
 <xs:simpleType name="QNameList"><xs:list itemType="xs:QName"/></xs:simpleType>
 <xs:attribute name="target">
  <xs:simpleType><xs:restriction base="sixml:QNameList"><xs:minLength value="1"/></xs:restriction></xs:simpleType>
 </xs:attribute>
 <xs:complexType name="Xml_AMark" final="restriction" block="#all" mixed="true">
  <xs:complexContent mixed="true">
   <xs:extension base="sixml:Xml_TMark"><xs:attribute ref="sixml:target" use="required"/></xs:extension>
  </xs:complexContent>
 </xs:complexType>
</xs:schema>
```

Figure 4.14: A simplified XML Schema instance document for the different patterns of use of marks

We use the type-derivation facility of XML Schema to define the element types that represent the different uses of marks. The type Xml_EMark is at the root of the type hierarchy. This type is used to associate a mark with an XML element. It includes an element named Descriptor to represent the descriptor of the associated mark, and the attribute markID to represent the ID of the mark. Both Descriptor and markID are labeled optional, but at least one of these two must be used. Section 4.6 discusses the use of Descriptor.

The optional element Context included in the type Xml_EMark represents the context information retrieved from the mark in question. Section 4.7 discusses the use of this element. Chapter 7 discusses the use of the optional attribute type.

The following example segment shows how an instance of the EMark relationship type in Figure 4.3 can be represented in XML. The element Event represents the entity Event in that figure. EMark_EventDetail represents the EMark relationship. (The relationship element's name is based on the name of the relationship pattern and the relationship type in Figure 4.3.)

```
<Event ID="..." EvDateTime="..." Kind="..." Source="..." Description="...">
  <EMark_EventDetail sixml:markID="87" xsi:noNamespaceSchemaLocation="."/>
</Event>
<xs:element name="EMark_EventDetail" type="sixml:EMark"/>
```

The attribute xsi:noNamespaceSchemaLocation in the example segment associates a schema with EMark_EventDetail. The prefix xsi indicates the XML Schema instance namespace [171]. The value (period) for xsi:noNamespaceSchemaLocation denotes that the schema for EMark_EventDetail is included in the "current" document. The element xs:element defines the schema for EMark_EventDetail. The schema simply states that EMark_EventDetail is of type EMark. (This somewhat convoluted method of associating a schema with an element is actually the simplest way of associating a schema using XML Schema.)

The second type in the type hierarchy, Xml_TMark, is used to model an AExcerpt relationship type when the ER attribute in question is represented as an element. (See the fifth data row in Table 4.2.) Xml_TMark extends Xml_EMark by two attributes. The Boolean attribute valueSource indicates whether the text content of the XML element that corresponds to an ER attribute is derived from the context of the associated mark. The string attribute valueExpression denotes the context element that supplies the value. We illustrate the use of Xml_TMark after discussing the use of the type Xml_AMark. Chapter 7 discusses the use of the attribute valueExpression.

The type Xml_AMark is used to associate a mark with XML attributes. It extends the type Xml_TMark by the attribute target. The value of this attribute is a list of qualified names. (A *qualified name* [125]—Qname for short—is a sequence of characters allowable as the name of an XML element or attribute, possibly combined with a prefix that is associated with a URI. For example, the strings sixml:markID and ErrDateTime are both QNames.) The qualified names listed as the value of the attribute target are required to identify attributes with which an Xml_AMark element associates a mark. The value of target must identify at least one attribute, and each identified attribute is associated with the same mark.

The following XML segment illustrates how instances of the AMark relationship types in Figure 4.4 might be represented in XML. The element Error denotes an Error entity. AMark_ErrorTime and AMark_ErrorDetails denote relationships. The missing attribute valueSource in each relationship element indicates that the target attribute's value is not derived from the associated mark. (Alternatively, valueSource may be set to "false".)

```
<Error ID="..." ErrDateTime="..." Source="..." Description="..." Notes="...">
  <AMark_ErrorTime sixml:markID="..." sixml:target="ErrDateTime"
                   xsi:noNamespaceSchemaLocation="."/>
  <AMark_ErrorDetails sixml:markID="..." sixml:target="Description"
                   xsi:noNamespaceSchemaLocation="."/>
</Error>
<xs:element name="AMark_ErrorTime" type="sixml:Xml_AMark"/>
<xs:element name="AMark_ErrorDetails" type="sixml:Xml_AMark"/>
```

The type Xml_AMark is also used to model an AExcerpt relationship type. In this use of Xml_AMark, the value of the attribute valueSource is always "true". For example, an instance of the AExcerpt relationship type in Figure 4.5 may be represented in XML as follows. Here, an OS update's title is represented as the attribute Title. The sub-element AExcerpt_UpdateTitle (of type Xml_AMark) associates the attribute Title with a mark. The sub-element's attribute valueSource denotes that the target attribute's value is the excerpt from the associated mark.

```
<OSUpdate Title="..." Description="..." Reason="...">
  <AExcerpt_UpdateTitle sixml:markID="146" sixml:target="Title" sixml:valueSource="true"
                        xsi:noNamespaceSchemaLocation="."/>
</OSUpdate>
<xs:element name=" AExcerpt_UpdateTitle" type="sixml:Xml_AMark"/>
```

The element type Xml_TMark handles the case of an ER attribute that participates in an AExcerpt relationship, and is represented as text content of an XML element. (This case corresponds to the fifth data row in Table 4.2.) The type Xml_AMark cannot be used in this case because the target XML attribute would not exist. For example, if the attribute Title involved in the AExcerpt relationship type in Figure 4.5 is represented as text content, we insert the element TExcerpt_UpdateTitle (of type Xml_TMark) into the element OSUpdate as follows:

```
<OSUpdate Description="..." Reason="...">
  <TExcerpt_UpdateTitle sixml:markID="146" sixml:valueSource="true"
                        xsi:noNamespaceSchemaLocation="."/>
</OSUpdate>
<xs:element name=" TExcerpt_UpdateTitle" type="sixml:Xml_TMark"/>
```

A sub-element of type Xml_TMark is only a *design-time proxy* for text content. At run time, this proxy is replaced by the excerpt retrieved from the associated mark. Chapter 7 describes how the proxy is replaced at run time.

The element types Xml_EMark, Xml_AMark, and Xml_TMark *disallow* an instance of a derived type to be used in place of an instance of the specified type (indicated by the value "#all" for the attribute block). This constraint ensures that only an element of the most appropriate type is used to represent the use of a mark. For example, an instance of Xml_AMark may not be used to associate a mark with an element.

XML Schema cannot express or enforce all the constraints we need to represent the use of marks. For example, XML Schema (more precisely, an XML Schema compliant application) can ensure that the value of the attribute target of an Xml_AMark element is a list of qualified names. However, it cannot ensure that a name mentioned in the list identifies an attribute of the containing element. Chapter 7 shows how we enforce constraints that XML Schema cannot enforce.

4.5.3. Generating Schema for the EMark Pattern

To generate the schema for an EMark relationship, we first generate the schema for the regular entity types that participate in the relationship. Then, into the schema generated in the first step, we insert the schema for a sub-element of type Xml_EMark.

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="Event">
    <xs:complexType>
     <xs:attribute name="ID" type="xs:string"/>
     <xs:attribute name="EvDateTime" type="xs:dateTime"/>
     <xs:attribute name="Kind" type="xs:string"/>
     <xs:attribute name="Source" type="xs:string"/>
     <xs:attribute name="Description" type="xs:string"/>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

(a)

```
<xs:schema xmlns:sixml="http://schema.sixml.org" xmlns:xs="http://www.w3.org/...">
  <xs:import namespace="http://schema.sixml.org"/>
  <xs:element name="Event">
    <xs:complexType>
     <xs:sequence>
      <xs:element name="EMark_EventDetail" type="sixml:Xml_EMark" maxOccurs="1"/>
     </xs:sequence>
     <xs:attribute name="ID" type="xs:string"/>
     <xs:attribute name="EvDateTime" type="xs:dateTime"/>
     <xs:attribute name="Kind" type="xs:string"/>
     <xs:attribute name="Source" type="xs:string"/>
     <xs:attribute name="Description" type="xs:string"/>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

(b)

```
<Event ID="2" EvDateTime="..." Kind="S" Source="Log" Description="Started"
        xsi:noNamespaceSchemaLocation="http://schema.sixml.org/examples/ssib.xsd"/>
    <EMark_EventDetail sixml:markID="87"/>
</Event>
```

(c)

Figure 4.15: XML schema generated for an EMark relationship type. (a) Schema for the entity type Event of Figure 4.3 excluding the EMark relationship type; (b) Schema for the entity type Event including the EMark relationship type; (c) An instance of the schema in Part (b) of this figure

Figure 4.15(a) shows the XML schema generated in the first step of the procedure for the Event entity of Figure 4.3. It represents the Event entity type as an XML element and defines the attributes of the Event entity as XML attributes. This schema does not include the EMark relationship type EventDetail of Figure 4.3.

Figure 4.15(b) shows the XML schema produced in the second step of the procedure. This schema first imports the XML schema in the namespace "sixml" (that is, the schema shown in Figure 4.14) so that it can reference the element type Xml_EMark. The schema then includes the element named EMark_EventDetail of type Xml_EMark, and sets the attribute maxOccurs appropriately.

Figure 4.15(c) shows an example instance of the schema generated in the second step (and shown in Figure 4.15(b)). The instance assumes that the generated schema is stored in the file pointed to by the attribute xsi:noNamespaceSchemaLocation. (The complete schema for the SSIB application is available online at the location indicated in Figure 4.15(c).)

*4.5.4. Generating Schema for the **AMark** and **AExcerpt** Patterns*

The procedures to generate the XML schema for the AMark and AExcerpt patterns are similar to the procedure for EMark, except that elements of types Xml_AMark are introduced. For these patterns we show only the final XML schema generated.

Figure 4.16 shows the XML schema generated for the Error entity type of Figure 4.4, including the two AMark relationship types. The ER attributes that participate in the AMark relationships are modeled as attributes. The elements AMark_ErrorTime and AMark_ErrorDetails denote the AMark relationship types.

Figure 4.16 also shows the XML schema generated for the OSUpdate entity type and the AExcerpt relationship type of Figure 4.5. The element AExcerpt_UpdateTitle denotes the AExcerpt relationship type.

Each of the AMark elements shown in Figure 4.16 is associated with a target attribute (as illustrated in the second and third example XML segments in Section 4.5.2) when the schema is instantiated.

```
<xs:schema xmlns:sixml="http://schema.sixml.org"
           xmlns:xs="http://www.w3.org/2001/XMLSchema">
 <xs:import namespace="http://schema.sixml.org"/>
 <xs:element name="Error">
  <xs:complexType>
   <xs:sequence>
    <xs:element name="AMark_ErrorTime" type="sixml:Xml_AMark" maxOccurs="1"/>
    <xs:element name="AMark_ErrorDetails" type="sixml:Xml_AMark"
               maxOccurs="1"/>
   </xs:sequence>
   <xs:attribute name="ID" type="xs:string"/>
   <xs:attribute name="ErrDateTime" type="xs:dateTime"/>
   <xs:attribute name="Source" type="xs:string"/>
   <xs:attribute name="Description" type="xs:string"/>
   <xs:attribute name="Notes" type="xs:string"/>
  </xs:complexType>
 </xs:element>

 <xs:element name="OSUpdate">
  <xs:complexType>
   <xs:sequence>
    <xs:element name="AExcerpt_UpdateTitle" type="sixml:Xml_AMark"
               maxOccurs="1"/>
   </xs:sequence>
   <xs:attribute name="ID" type="xs:string"/>
   <xs:attribute name="Title" type="xs:string"/>
   <xs:attribute name="Description" type="xs:string"/>
   <xs:attribute name="Reason" type="xs:string"/>
  </xs:complexType>
 </xs:element>
</xs:schema>
```

Figure 4.16: XML schema generated for the AMark and AExcerpt relationship types. The schema generated for the Error entity type and the AMark relationship types of Figure 4.4, and the schema generated for the OSUpdate entity type and the AExcerpt relationship type of Figure 4.5 are shown

4.6. Modeling Mark Descriptors

In Section 3.7.3 we observed that a mark describes one endpoint—a base selection—of a *potential* link. An actual link to the base selection is created when a mark is associated with an SI element (such as a Sidepad item). Thus far in this chapter, we have modeled a link's endpoint (as the Mark entity type) and a link itself (as a relationship type involving the Mark entity type). In this section, we model the *specification* of a link's endpoint: a mark descriptor.

Ideally, we like to be able to represent a descriptor for any linking technology (such as OLE 2 compound documents [18] and SPARCE), but doing so can be challenging because the structure of a mark descriptor can vary widely among linking technologies. For example, the OLE 2 compound document system allows a variety of *monikers* (a moniker encodes an address), while a URI may include a *fragment identifier* (the portion of a URI that begins with the # character) [15] containing practically any data. Further, some frameworks allow development of new kinds of endpoint specifications. For example, the XPointer framework [168] allows development of new pointer schemes. (An XPointer *pointer* specifies a fragment of a resource that is identifiable using a URI.)

In the rest of this section, we present a conceptual model for a mark descriptor and discuss the representation of a mark descriptor in relational and XML schemas. Through examples we show how our model can represent descriptors for any linking technology.

4.6.1. Conceptual Modeling

In Section 4.3 we mentioned that a Mark entity has the attribute ID as the primary key, and that other attributes of the entity are derived from its descriptor. A simple way (Alternative 1) to represent a mark descriptor for any linking technology is to add to the Mark entity type a Kind attribute to denote the linking technology, and a Descriptor attribute to store a serialized form of a mark descriptor. This approach provides a simple solution to the storage problem, but makes it hard to query the structure of a descriptor.

Alternative 2 is to define a variation of the Mark entity type for each linking technology, with attributes specific to that technology. This approach allows storing of different kinds of descriptors (for different linking technologies), and allows structured queries over descriptors. However, the SA developer would need to choose, at design time, a variation of the Mark entity type to represent each use of marks. That is, the SA developer would need to choose linking technologies at SI-design time. (There is no notion of type inheritance in this alternative. Also, the basic ER model does not support inheritance.)

Alternative 3 is to specialize the Mark entity type (that is, derive new entities from Mark) for different kinds of descriptors using the Extended-ER (EER) model [41]. This approach allows storing of different kinds of descriptors, allows structured queries over descriptors, and leaves unchanged both the model for mark and the model for use of marks. Also, the SA developer would not need to choose linking technologies at SI-design time.

While it seems to exhibit several advantages, Alternative 3 is not practical for two reasons: First, the most popular logical data model, the relational model, does not natively support specialization. (Some relational database managements systems, for example DB2 [62], do support some form of specialization.) Second, many of the queries over descriptors would need to consider descriptor kind because the attributes of descriptors tend to vary widely. Thus, for practical reasons, a Kind attribute would still be required in the Mark entity type. Example 4 in Section 4.8 demonstrates a use of this attribute.

Alternative 4 is to add an attribute Kind to the Mark entity type to denote descriptor kind, add a new entity type for each kind of descriptor to support, and relate the Mark entity type with these new entity types. This approach has all the benefits of Alternative 3, except that it changes the Mark entity (because it adds an attribute). Also, it does not require support for inheritance.

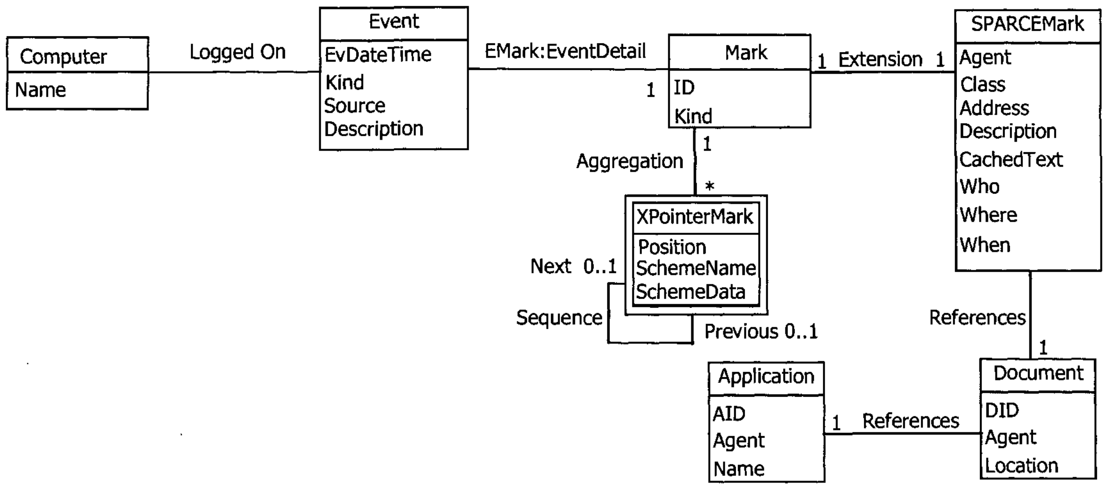


Figure 4.17: A conceptual model for a mark descriptor. Example descriptor entity types are included only for illustration

We pursue Alternative 4 to model mark descriptors. Figure 4.17 shows an ER schema in this alternative. It shows a new attribute Kind added to the entity type Mark. It also includes the EMark relationship type of Figure 4.3 to demonstrate that relationship types that indicate the use of marks are unaffected. The figure also includes entity types for mark descriptors for SPARCE and for XPointer. The entity type SPARCEMark models a SPARCE mark descriptor. The attributes of this entity type and those of the related entities Document and Application are obtained from the descriptors in Figure 3.4.

The entity type XPointerMark models an XPointer scheme-based pointer [168]. A *scheme-based pointer* in the XPointer framework is a sequence of *(scheme, fragment)* pairs, where *scheme* is the name of an addressing scheme. The term *fragment* is a specification of a fragment of data within the context of a resource pointed to by a URI. A fragment specification is a string constructed using the production rules specified in the grammar for the associated addressing scheme. The pointer schemes defined using the XPointer framework are: *element()* [173] to address XML elements using element ids and positions, and *xpointer()* [174] to address portions of XML data using an extension of the XPath syntax [166]. Different fragments in a sequence that forms a scheme-based pointer may be addressed using different pointer schemes. For example, the first fragment in a sequence may be identified using the *element()* scheme, whereas the second fragment may be identified using the *xpointer()* scheme. The entity type XPointerMark supports this kind of mixture of pointer schemes with the help of the attribute SchemeName.

The entity type XPointerMark is modeled as a weak entity in Figure 4.17 because multiple XPointer pointers (in a sequence) may be associated with a single mark ID. The attribute Position is the partial key for this entity type. The value of this attribute denotes the position of an XPointer pointer in the sequence associated with a mark ID. For each pointer in the sequence, the attribute SchemeName denotes the pointer scheme, and the attribute SchemeData denotes the fragment identified.

In this schema, we conveniently use the entity XPointerMark to represent both a URI and a sequence of XPointer pointers that specifies a fragment within the resource that the URI references. When representing a URI, we store the name of the URI scheme in the attribute SchemeName and the rest of the URI in the attribute SchemeData. In this approach, the first XPointerMark entity for a given mark ID (that is, the entity with the value zero for the attribute Position for the mark ID) represents a URI.

We have included the entity types SPARCEMark and XPointerMark in Figure 4.17 only as examples of supporting descriptors of different linking technologies.

4.6.2. Relational Schema

We revise the schema of the relation `Mark` (to add the `Kind` attribute), but the procedures to generate relational schemas for any of the patterns of use of marks do not change.

Figure 4.18(a) shows the revised schema for the relation `Mark`. Figure 4.18(b) shows the schema for the mark, document, and application descriptors of SPARCE. The attribute `MarkID` in the relation `SPARCEMark` is both a primary key and a foreign key referencing the attribute `Mark.ID`. That is, a row in this relation is an extension of a row in the relation `Mark`. Figure 4.18(c) shows the schema for the XPointer descriptor.

One or more rows (varying by the attribute `Position`) in this relation correspond to a row in the `Mark` relation.

```
CREATE TABLE Mark
(
 ID INTEGER NOT NULL PRIMARY KEY, Kind VARCHAR(50) NOT NULL
)
```

(a)

```
CREATE TABLE SPARCEMark
(
 MarkID INTEGER NOT NULL REFERENCES (Mark.ID),
 Agent VARCHAR(50), Class VARCHAR(50), Address VARCHAR(255),
 Description VARCHAR(1024), CachedText VARCHAR(1024),
 Who VARCHAR(255), Where VARCHAR(255), When TIMESTAMP,
 DID INTEGER NOT NULL REFERENCES (Document.DID),
 PRIMARY KEY MarkID
)
CREATE TABLE Document
(
 DID INTEGER NOT NULL PRIMARY KEY,
 Agent VARCHAR(50), Location VARCHAR(1024),
 AID INTEGER NOT NULL REFERENCES (Application.AID)
)
CREATE TABLE Application
(
 AID INTEGER NOT NULL PRIMARY KEY,
 Agent VARCHAR(50), Name VARCHAR(50)
)
```

(b)

```
CREATE TABLE XPointerMark
(
 MarkID INTEGER NOT NULL REFERENCES (Mark.ID),
 Position INTEGER NOT NULL,
 SchemeName VARCHAR(50) NOT NULL, SchemeData VARCHAR(1024),
 PRIMARY KEY (MarkID, Position)
)
```

(c)

Figure 4.18: Relational schema generated for mark descriptors. (a) Revised schema for the Mark entity type; (b) Schema for a SPARCE descriptor; (c) Schema for an XPointer pointer

4.6.3. XML Schema

Together, the element types Descriptor and Xml_EMark shown in Figure 4.14 allow the use of new kinds of descriptors without altering any element type defined thus far.

There are no constraints on types derived from Descriptor (because the attribute final is empty), and an instance of a derived type may be used where an instance of Descriptor is expected (because the attribute block is empty). That is, an instance of an Xml_EMark, and its derived types, may include a mark descriptor for any linking technology, by simply deriving a new type from Descriptor.

Figure 4.19 shows the element types to represent SPARCE mark descriptors and XPointer pointers. These types are derived from Descriptor. Figure 4.20 illustrates the use of the new descriptor kinds. The element EMark_EventDetail uses a SPARCE descriptor to associate the Event element with a range of cells in a spreadsheet. The element AExcerpt_UpdateTitle uses an XPointer descriptor to associate the Title attribute of the OSUpdate element with a part of an XML document (containing the SSIB updates catalog) located at `http://localhost/updates.xml`. The XPointer descriptor uses the *element()* scheme to address an element within this XML document.

The attribute xsi:type of the element Descriptor in each use of mark in Figure 4.20 indicates the type of the actual descriptor used. XML Schema [170] requires the use of this attribute whenever an instance of a derived type is used in place of an instance of an abstract base type. This attribute also models the Kind attribute of the Mark entity shown in Figure 4.17. (The attribute Kind in an Event element is SI, and denotes the kind of event recorded for a computer. It is unrelated to the kind of mark descriptor.)

```
<xs:schema xmlns:sixml="http://schema.sixml.org"
           xmlns:xs="http://www.w3.org/2001/XMLSchema">
 <xs:import namespace="http://schema.sixml.org"/>
  <xs:complexType name="Application">
   <xs:sequence>
    <xs:element name="AID" type="xs:string"/>
    <xs:element name="Agent" type="xs:string"/>
    <xs:element name="Name" type="xs:string"/>
   </xs:sequence>
  </xs:complexType>

  <xs:complexType name="Document" mixed="true">
   <xs:sequence>
    <xs:element name="DID" type="xs:string"/>
    <xs:element name="Agent" type="xs:string"/>
    <xs:element name="Location" type="xs:string"/>
    <xs:element name="Application" type="Application"/>
   </xs:sequence>
  </xs:complexType>
 <xs:complexType name="SPARCEMark" mixed="true">
  <xs:complexContent>
   <xs:extension base="sixml:Descriptor">
    <xs:sequence>
     <xs:element name="Agent" type="xs:string"/>
     <xs:element name="Class" type="xs:string"/>
     <xs:element name="Address" type="xs:string"/>
     <xs:element name="Description" type="xs:string"/>
     <xs:element name="CachedText" type="xs:string"/>
     <xs:element name="Who" type="xs:string"/>
     <xs:element name="Where" type="xs:string"/>
     <xs:element name="When" type="xs:dateTime"/>
     <xs:element name="Document" type="Document"/>
    </xs:sequence>
   </xs:extension>
  </xs:complexContent>
 </xs:complexType>

 <xs:complexType name="XPointer_Part" mixed="true">
  <xs:complexContent>
   <xs:restriction base="xs:anyType">
    <xs:sequence>
     <xs:element name="SchemeName" type="xs:string" minOccurs="1" maxOccurs="1"/>
     <xs:element name="SchemeData" type="xs:string" minOccurs="0" maxOccurs="1"/>
    </xs:sequence>
   </xs:restriction>
  </xs:complexContent>
 </xs:complexType>
 <xs:complexType name="XPointerMark" mixed="true">
  <xs:complexContent>
   <xs:extension base="sixml:Descriptor">
    <xs:sequence>
     <xs:element name="PointerPart" type="XPointer_Part" minOccurs="1"/>
    </xs:sequence>
   </xs:extension>
  </xs:complexContent>
 </xs:complexType>
</xs:schema>
```

Figure 4.19: XML schema for SPARCE descriptors and XPointer pointers

```
<Event ID="2" EvDateTime="..." Kind="S" Source="Log" Description="Started">
 <EMark_EventDetail sixml:markID="87">
  <sixml:Descriptor xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
                 xsi:type="SPARCEMark">
   <Agent>MSOfficeAgents.ExcelAgent</Agent>
   <Class>ExcelMark</Class>
   <Address>Sheet1|$A$2:$C$3</Address>
   <Description>Sheet Sheet1, Cell(s): A2:C3 in "C1Sys.xls" (MS Excel)</Description>
   <CachedText>The EventLog service was started</CachedText>
   <Who>smurthy</Who>
   <Where>TYEE</Where>
   <When>2004-05-28 14:03:02</When>
   <Document>
    <DID>D9</DID>
    <Agent>MSOfficeAgents.ExcelAgent</Agent>
    <Location>C:\C1Sys.xls</Location>
    <Application>
     <AID>A1</AID>
     <Agent>MSOfficeAgents.ExcelAgent</Agent><Name>MS Excel 2002</Name>
    </Application>
   </Document>
  </sixml:Descriptor>
 </EMark_EventDetail>
</Event>

<OSUpdate Title="..." Description="..." Reason="...">
 <AExcerpt_UpdateTitle sixml:markID="146" sixml:target="Title" sixml:valueSource="true">
  <sixml:Descriptor xmlns:xsi="http://www.w3.org/..." xsi:type="XPointerMark">
   <PointerPart>
    <SchemeName>http</SchemeName><SchemeData>localhost/updates.xml</SchemeData>
   </PointerPart>
   <PointerPart>
    <SchemeName>element</SchemeName><SchemeData>/1/3/1</SchemeData>
   </PointerPart>
  </sixml:Descriptor>
 </AExcerpt_UpdateTitle>
</OSUpdate>
```

Figure 4.20: Example use of SPARCE descriptor and XPointer pointer

4.7. Modeling Context Information

We now discuss modeling context information (that is, information related to a mark retrieved from the base layer). Text excerpt, font name, and containing paragraph are examples of context information.

As described in Section 3.3.1, the context information retrieved from a mark is organized as a hierarchy of context kinds and context elements. A context kind groups related parts of the context of a mark, whereas a context element is an atomic piece of information in the context of a mark. For example, text and image excerpts are context elements. Both context kinds and context elements have friendly names; a context element also has a value. A context kind may contain other context kinds.

Figure 4.21 shows an ER schema for a mark's context information. This schema allows an SA developer to access base information without explicitly modeling the information present in the base layer. It also makes navigation over bi-level information easy. For example, one can easily navigate from an Event entity (which is SI) to the text content for the mark associated with the entity.

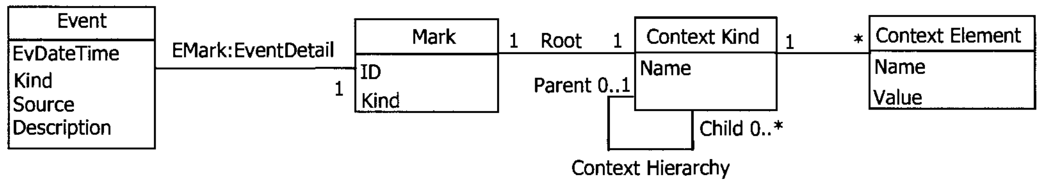


Figure 4.21: A conceptual schema for context information. The entity type Event is included to illustrate the ability to navigate bi-level information (that is, navigate from SI to context information)

Though this conceptual model enables queries over context information, expressing such queries can be quite cumbersome in the relational model (due to the nesting of context kinds). Querying in the XML model is much easier, but the queries will likely employ a large number of value predicates (to test the names of context kinds and context elements). Further, we do not expect context information to be actually stored in a database, because the complete context information for a mark can be arbitrarily large, and certain context elements might not be queried at all. (An SA might choose to cache parts of context information for performance bebefits.)

In the rest of this section, we introduce a means of retrieving context information dynamically at query execution time, and introduce a representation for context information that makes query expression easier in the XML model.

We define the function `context` to dynamically retrieve the value of an element in the context for a mark. This function accepts a mark ID, a hierarchy of context kinds specified as a path expression, and the name of a context element; and it returns the value of the specified context element. For example, the function call `context('87', 'Content', 'Text')` returns the text content for the mark whose ID is `87`. This call is equivalent to the call `excerpt('87')`. (Section 4.4.2 describes the function `excerpt`.) The call `context('87', 'Container/Row', 'Text')` returns the text content of the row that contains the region referenced by the mark with ID `87`. The function returns an empty value if the context-kind hierarchy or the context element supplied is not applicable to the specified mark.

The function `context` makes it easy to traverse context hierarchies, especially in the relational model, by eliminating the potentially large number of self joins over the relation that represents context kinds. Using it also avoids eager materialization of context information in both the relational and XML models. Chapters 5 through 9 discuss in detail the issues related to executing queries over bi-level information.

We now introduce a representation for context information to more easily query context in the XML model. This representation is a simple variation of the one generated from the conceptual schema shown in Figure 4.21. We begin with an illustration of the need for an alternative schema.

Figure 4.22 shows the Event element in Figure 4.20 with partial context information for the associated mark (to a range of cells in a spreadsheet) added in. Context information is included in the element Context, ContextKind represents a context kind, and ContextElement represents a context element. The text content of a ContextElement stores the value of the corresponding element. Instances of ContextKind and ContextElement are nested to reflect the context hierarchy. This representation for context information is faithful to the conceptual schema in Figure 4.21.

```
<Event ID="2" EvDateTime="2000-04-28T11:45:00" Kind="S" Source="Log"
       Description="Started">
  <EMark_EventDetail sixml:markID="87">
    <sixml:Descriptor xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
                  xsi:type="SPARCEMark">
     <!-- Descriptor truncated for brevity. Figure 4.20 shows the complete descriptor -->
    </sixml:Descriptor>
    <sixml:Context>
     <ContextKind name="Content">
      <ContextElement name="Text">The operation was canceled by the user.</ContextElement>
     </ContextKind>
     <ContextKind name="Container">
      <ContextKind name="Row">
       <ContextElement name="Text">
        The operation was canceled by the user. Your computer ... network address (DHCP) server
       </ContextElement>
      </ContextKind>
     </ContextKind>
     <ContextKind name="Placement">
      <ContextElement name="Sheet">Sheet1</ContextElement>
     </ContextKind>
    </sixml:Context>
  </EMark_EventDetail>
</Event>
```

Figure 4.22: Partial context information for a mark to cells in a spreadsheet represented using a generic schema. The schema used corresponds to the conceptual schema in Figure 4.21

Given this representation for context information, the following XPath expression can retrieve the text of the row (or rows) containing the spreadsheet cells associated with an event. This expression uses predicates over the names of context kinds and elements to navigate to the desired context element.

```
Event/EMark_EventDetail/sixml:Context/ContextKind[@name='Container']
/ContextKind[@name='Row']/ContextElement[@name='Text']
```

```
<Event ID="2" EvDateTime="2000-04-28T11:45:00" Kind="S" Source="Log"
        Description="Started">
  <EMark_EventDetail sixml:markID="87">
    <sixml:Descriptor xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
                      xsi:type="SPARCEMark">
      <!-- Descriptor truncated for brevity. Figure 4.20 shows the complete descriptor -->
    </sixml:Descriptor>
    <sixml:Context>
      <Content>
        <Text>The operation was canceled by the user.</Text>
      </Content>
      <Container>
        <Row>
         <Text>
          The operation was canceled by the user. Your computer ... network address (DHCP) server
         </Text>
        </Section>
      </Container>
      <Placement>
        <Sheet>3</Sheet>
      </Placement>
    </sixml:Context>
  </EMark_EventDetail>
</Event>
```

Figure 4.23: Partial context information in a schema determined by a context agent developer

A simple change in this representation for context information is to express each context kind and context element with an element whose name is the same as the value of its name attribute. Figure 4.23 shows the context information in Figure 4.22 in the revised representation. With this revision, the following expression retrieves the text of the rows containing the cells associated with an event, without using value predicates:

```
Event/EMark_EventDetail/sixml:Context/Container/Row/Text
```

In addition to being simpler, the revised expression can potentially execute more efficiently because the query processor can use *structural indexes*. Also, the original approach fixes a schema for context information, but, in the revised approach, a context-agent implementer may choose a schema. A potential disadvantage with the revised approach is that an SA developer might tightly couple SI with the schema that a particular context agent employs. (The revised approach does let the context-agent developer model context information as in the original approach.)

In the rest of this dissertation, we use the functions `excerpt` and `context` to explicitly retrieve context information in the relational model. These functions can be used in the XML model as well, but we use only the representation in Figure 4.23 when discussing the XML model. (Section 4.8 does use these functions with XML for illustration.) Chapter 5 discusses strategies to realize the representation scheme in Figure 4.23 when executing bi-level queries.

4.8. Querying Bi-level Information

We now demonstrate the ability to express *bi-level queries* (that is, queries over bi-level information) using the logical schemas we generate. The examples in this section are based on the schema for the SSIB application. They demonstrate the ability to express structured queries over the combined SI and base information, though the base information may be heterogeneous, distributed, loosely structured, and not stored in a traditional database. Chapter 5 introduces the process and performance of bi-level-query execution.

Example 1: List all OS updates related to security.

The following SQL query suffices for the relational model (see Figure 4.11):

```
SELECT * FROM OSUpdate WHERE Title LIKE '%Security%'
```

The following XPath expression suffices for the XML model (see Figure 4.16):

```
//OSUpdate[contains(@Title, 'Security')]
```

In both queries, the descriptions of OS updates are *automatically* obtained from the base layer because the attribute Title is modeled using the AExcerpt pattern (see Figure 4.5).

Example 2: List errors related to the application MS Word (see Figure 4.4).

The following SQL query suffices for the relational model (see Figure 4.9(b)):

```
SELECT *
FROM Error
WHERE excerpt(AMark_ErrorDetails) LIKE '%word.exe%'
```

The following XQuery expression may be used in the XML model (see Figure 4.16):

```
<Errors> {
 FOR $e IN document("SSIB.xml")//Error
 LET $d = sixml:excerpt($e/AMark_ErrorDetails/@sixml:markid)
 WHERE contains($d, 'word.exe')
 RETURN $e
} </Errors>
```

The following XPath expression may also be used in the XML model, if the XPath processor permits user-defined functions (*extension functions* in XML terminology):

```
//Error[contains(sixml:excerpt(AMark_ErrorDetails/@sixml:markID),
'word.exe')]
```

All three queries in this example employ the user-defined function `excerpt` (introduced in Section 4.4.2) to retrieve, at query-execution time, the text excerpt from the mark associated with the Description attribute of each Error entity. All queries return the stored error information if the text excerpt retrieved for the error description contains the string `word.exe`.

The explicit use of the function `excerpt` can be avoided if the `Description` attribute is conceptually modeled after the AExcerpt pattern instead of the AMark pattern. However, this choice is left to the SA developer.

Example 3: For each mark associated with events, retrieve the text of the row containing the marked region. For example, if a mark points to a few cells within a row of an MS Excel spreadsheet, return the text of the row that contains the marked cells.

The following SQL query suffices for the relational model (see Figure 4.9(a)):

```
SELECT context(EMark_EventDetail, 'Container/Row', 'Text') FROM Event
```

Assuming the XPath processor permits extension functions, the following XPath expression suffices for the XML model:

```
sixml:context(//Event/EMark_EventDetail/@sixml:markID,
'Container/Row', 'Text')
```

Both these queries employ the user-defined function `context` (described in Section 4.7) to retrieve context information at query-execution time. The following simpler XPath expression may be used in the XML model if context information is represented within the element that represents a use of a mark (as in Figure 4.23):

```
//Event/EMark_EventDetail/sixml:Context/Container/Row/Text
```

Example 4: List the number of each kind of mark descriptor in use.

The following SQL query may be used in the relational model (see Figure 4.18).

```
SELECT 'SPARCE', COUNT(MarkID) FROM SPARCEMark
UNION
SELECT 'XPointer', COUNT(DISTINCT MarkID) FROM XPointerMark
```

The following is an alternative SQL query.

```
SELECT Mark.Kind, COUNT(*) FROM Mark GROUP BY Mark.Kind
```

The first SQL query counts the number of *unique* mark IDs in the relations maintained for each kind of mark descriptor. The number of unique mark IDs in the `SPARCEMark` relation is the same as the number of mark IDs because the attribute `MarkID` is the primary key. On the other hand, the `XPointerMark` relation may have multiple rows per mark ID. Thus, writing this query requires *a priori* knowledge of the kinds of descriptors in use and the schema of the relation for each kind of descriptor.

The second query also counts the unique marks per descriptor kind, but writing it does not require the knowledge of the schema of the relation for each kind of descriptor (because it uses the `Kind` attribute of the relation `Mark`).

The following XSLT 2.0 [178] template suffices for the XML model (see Figure 4.20). For simplicity, we assume that all uses of marks include mark descriptors, not mark IDs.

```
<xsl:template match="/">
 <Counts>
  <xsl:for-each-group select="//sixml:Descriptor" group-by="@xsi:type">
   <xsl:element name="{string(current-grouping-key())}">
    <xsl:value-of select="count(current-group())"/>
   </xsl:element>
  </xsl:for-each-group>
 </Counts>
</xsl:template>
```

The XSLT 2.0 template is similar to the second SQL query. It groups `Descriptor` elements by the attribute `xsi:type` and outputs one XML element for each distinct value of that attribute. The content of each element output is the number of descriptors of the descriptor kind corresponding to the element.

We use XSLT 2.0 in this example, rather than using XQuery, XPath, or XSLT 1.0 [177], because only XSLT 2.0 has the features needed to express the query in a declarative manner. Also, the use of XSLT 2.0 demonstrates that the XML schema our methodology generates can be used with different XML query languages.

4.9. Evaluation

We evaluate the relationship patterns to represent the use of marks by applying the patterns to three SAs with distinct information needs. The SAs are: Sidepad (introduced in Section 1.2.1), the Superimposed Scholarly Review System (SISRS) [109] (introduced here in Section 4.9.2), and the Superimposed System-Information Browser (SSIB) introduced in Section 4.2.

4.9.1. Sidepad

Section 1.2.1 introduced a simple scratchpad SA called Sidepad, and Figure 1.3 shows an instance of a document created in Sidepad. Figure 4.24 shows a conceptual model for the SI created with Sidepad. According to this schema, a Sidepad *document* owns items and groups. An *item* has a user-assigned name and a descriptive text. It may be associated with a mark (indicated by the EMark relationship ItemMark). A *group* contains items and other groups. It has only a user-assigned name, and it too may be associated with a mark (represented by the relationship GroupMark). An item is owned either by a group or by a document, but this constraint cannot be expressed in the ER model. Items and groups have display attributes such as shape and color, but we omit modeling those attributes for simplicity.

The relationship Nests represents the possible nesting of groups. Section 4.10.1 describes the relationship pattern Hierarchy that Nests follows.

Because the conceptual schema for Sidepad is quite simple, we omit showing the logical schemas in the relational and XML models.

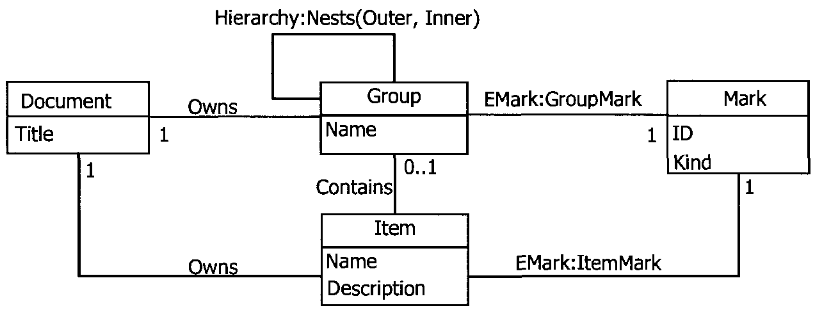


Figure 4.24: A conceptual schema for SI created using Sidepad

4.9.2. The Superimposed Scholarly Review System (SISRS)

The Superimposed Scholarly Review System (*SISRS*, pronounced *scissors*) [109] is an SA that assists in a peer-review process (such as that an academic conference might use). SISRS helps a reviewer superimpose comments on an electronic version of a paper, and prepare a review report. It also helps collate review reports for a paper from different reviewers, and prepare feedback to authors. SISRS uses bi-level queries to prepare both reviewer reports and author feedback. Chapter 9 shows some bi-level queries executed over SISRS documents.

Figure 4.25 shows a conceptual schema for SISRS. In this schema, each paper has a title. The AExcerpt relationship type TitleSource indicates that a paper's title is obtained from the base paper. The EMark relationship type Applies To represents the region(s) of the paper with which a comment is associated. The AMark relationship type References allows the *text of a comment* to be associated with marks. For example, a reviewer might cite passages from related work. The AExcerpt relationship type Help allows easy access to the excerpt of the commented region.

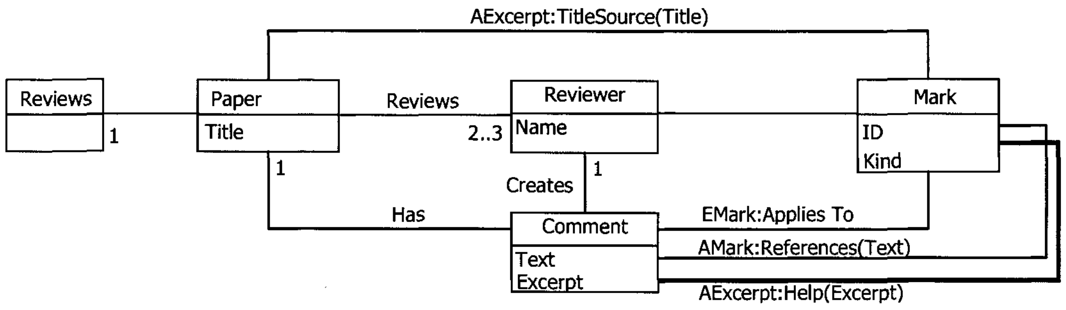


Figure 4.25: A conceptual schema for SI created using SISRS. The bold line distinguishes the AExcerpt relationship type from the AMark relationship type

Figure 4.26 shows a document that conforms to the XML schema generated for SISRS. The attribute xsi:noNamespaceSchemaLocation points to the generated schema. The element EMark_AppliesTo associates the lone comment shown with the commented region. AExcerpt_Help assigns the excerpt of the *same* commented region to the attribute excerpt. Thus, both these mark association elements use the mark ID "23". TMark_References associates the comment text (modeled as the text content of the element Comment) with a mark. AExcerpt_TitleSource indicates that the commented paper's title is a mark's excerpt. Chapter 7 describes how an attribute is assigned an excerpt at run time.

We now illustrate the ability to express a bi-level query over the XML schema generated from the conceptual schema. Figure 4.27 shows two XSLT templates to generate an HTML document containing author feedback for each paper. For brevity, the query does not cluster comments by reviewer.

```
<Reviews xsi:noNamespaceSchemaLocation="http://schema.sixml.org/examples/sisrs.xsd"
          xmlns:xsi= "http://www.w3.org/2001/XMLSchema-instance"
          xmlns:sixml="http://schema.sixml.org">
 <Paper title="">
  <Comment excerpt="" reviewer="r1">
   <TMark_References sixml:markID="...">Text of the comment</TMark_References>
   <AExcerpt_Help sixml:markID="23" sixml:target="excerpt"
                 sixml:valueSource="true"/>
   <EMark_AppliesTo sixml:markID="23"/>
  </Comment>
  <AExcerpt_TitleSource sixml:markID="..." sixml:target="title"
                       sixml:valueSource="true"/>
 </Paper>
</Reviews>
```

Figure 4.26: XML representation of a SISRS document. Elements in bold show use of marks

The template for the element Paper writes out one HTML document for each reviewed paper. This template writes the current paper's title directly from the attribute title even though the title is an excerpt from the reviewed paper. This operation is possible because title participates in an AExcerpt relationship in the conceptual schema.

```
<xsl:template match="Paper">
 <xsl:document method="html" href="{@title}">
  <HTML><BODY>
   <P style="font-size:32"><xsl:value-of select="@title"/></P>
   <xsl:apply-templates select="Comment">
    <xsl:sort select="EMark_AppliesTo/sixml:Context/Placement/Page"
            data-type="number"/>
   </xsl:apply-templates>
  </BODY></HTML>
 </xsl:document>
</xsl:template>

<xsl:template match="Comment">
 <P>
  <B><xsl:text>Page: </xsl:text></B>
  <xsl:value-of select="EMark_AppliesTo/sixml:Context/Placement/Page"/>
 </P>
 <P>
  <B><xsl:text>Excerpt: </xsl:text></B><I><xsl:value-of select="@excerpt"/></I>
 </P>
 <P><B><xsl:text>Comment: </xsl:text></B><xsl:value-of select="text()"/></P>
</xsl:template>
```

Figure 4.27: Bi-level XSLT templates to generate author feedback in HTML format from SISRS data. Key parts of the query are in bold font. Output HTML elements are in upper case

After writing the title, the template for Paper triggers a template for each contained Comment such that the comments are processed in the order of the page containing the commented regions. The template for Comment first writes the page number of the commented region, then writes (in italics) the text excerpt of the commented region, and then writes the comment text. The page number and the text excerpt for a commented region are obtained from the base layer at query-execution time.

4.9.3. The Superimposed System-Information Browser (SSIB)

Section 4.3 has described how the various uses of marks in the SSIB application are represented using relationship patterns. Sections 4.4 and 4.5, respectively, show the logical schemas generated for SSIB in the relational and XML models. Figure 4.28 shows the complete ER schema for SSIB.

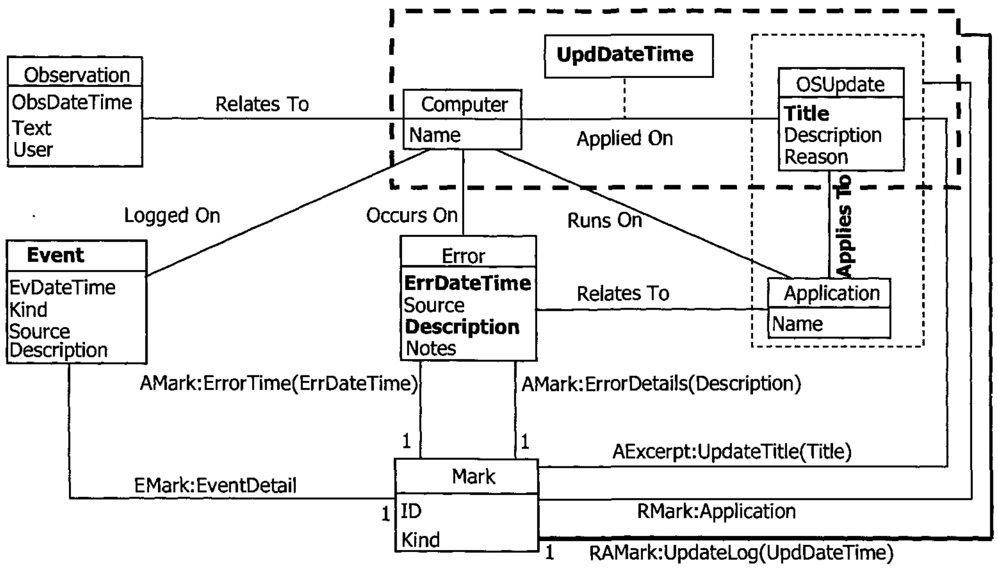


Figure 4.28: The complete conceptual schema for SSIB. All relationships between SI entities are many-to-many. All entities have a key attribute named ID (not shown). Names in bold in the original ER schema of Figure 4.2 are retained for ease of comparison.

4.10. Related Work

In this section, we review the notion of relationship patterns, the building block of our methodology to represent the use of marks. We also review four systems that conceptually model links using a form of the ER model and compare them to our methodology.

4.10.1. Relationship Patterns

Many researchers have extended the ER model, mostly by adding new constructs to the model. For example, Elmasri and Navathe [41] introduce new constructs to support specialization; Tanaka and others [152] add constructs to express application semantics; and Cysneiros and others [29] add constructs to express non-functional requirements.

Our methodology to represent use of marks in the ER model does not introduce new constructs, but uses a set of conventions for existing constructs based on the notion of relationship patterns [114]. A *relationship pattern* is an abstraction of a recurring need when establishing relationships among information elements in specific contexts. A relationship pattern is similar to a software-design pattern [47], except that it focuses on relationships. Like software-design patterns, relationship patterns are independent of modeling languages (although a particular modeling language may not have the power to express certain relationship patterns).

Defining a relationship pattern allows developers to solve a kind of problems once (rather than solving repeatedly), and it helps developers understand many relationship types at once. It also lets developers customize how relationships are treated in various stages of the information life cycle. Finally, it allows developers to leverage known patterns and existing solutions.

A relationship-pattern specification describes the contexts in which the pattern applies; the syntax to express relationship types of the pattern; the semantics and constraints of the pattern; and the consequences of using the pattern. For example, Section 4.3 informally describes the syntax, semantics, and constraints for each pattern of mark use. The name of each pattern (and the heading of the section in which the pattern is discussed) conveys the context in which that pattern applies. Sections 4.4 and 4.5 describe a consequence (in the form of the effect on logical schemas) of using the patterns to represent the use of marks.

Relationship patterns may be used to specify patterns of any kind of relationships, not just to specify the use of marks. When a developer recognizes a relationship pattern, he simply needs to describe it by specifying the context, syntax, semantics, constraints, and consequences of the pattern. For example, Figure 4.29(a) shows a relationship type Manages that an entity type named Employee has with itself. This relationship type models a hierarchical relationship among employees in a company. Developers encounter such relationships frequently, for example, when modeling bill-of-material and supply-chain relationships. The relationship type Context Hierarchy in Figure 4.21 is another example. Such relationship types have many things in common. First, they all represent hierarchies. The role names of relationships may change, but the roles have the same cardinality constraints: an entity (called Employee in Figure 4.29) has zero or one other entity playing the role of a "parent" (called Manager in Figure 4.29), and zero or more "children" entities (called Subordinate in Figure 4.29). Ignoring the labels, all these types of hierarchical relationships lead to the same general logical schema.

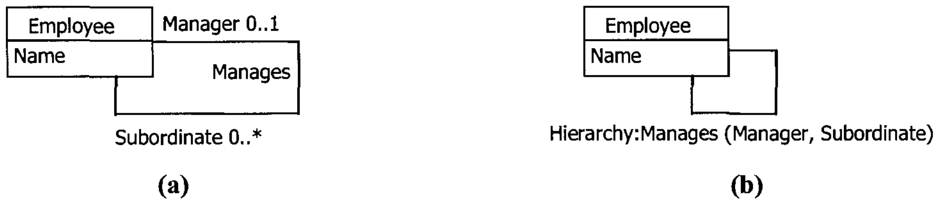


Figure 4.29: Example application of relationship patterns. (a) A hierarchy of manager and sub-ordinates; (b) The hierarchy of manager and sub-ordinates modeled after a relationship pattern called Hierarchy

A developer can capture the commonality among the hierarchical relationship types by defining a relationship pattern called Hierarchy. He can define the syntax to express relationships of this pattern, define the semantics of the pattern, impose cardinality constraints, and define the procedures to generate logical schemas. He can then get consistent representations of hierarchies by simply instantiating the Hierarchy pattern. The signature for this pattern could be: Hierarchy:<type>(<parent role>, <child role>), where <parent role> and <child role> are names of roles of "parent" and "child" entities, respectively, in a relationship.

Figure 4.29(b) shows the relationship type of Figure 4.29(a) expressed using the aforementioned Hierarchy pattern. Cardinality constraints are not shown because the relationship-pattern specification (which is omitted for simplicity) automatically assigns the cardinality 0..1 to the Manager role (that is, to the "parent") and the cardinality 0..* to the Subordinate role (that is, to the "child").

The relationship Nests in Figure 4.24 illustrates another use of the Hierarchy pattern.

4.10.2. Conceptual Models for Links

In this section, we review four systems that conceptually model links using a form of the ER model: structured maps [31], superimposed schematics [17], the nested-context model [24], and the hypertext design model [48]. Structured maps and superimposed schematics are models for SI, developed by colleagues in our research group. The nested-context model and the hypertext design model are models for hypertext, and are developed by others. We also review topic maps [68] (though they do not use the ER model) because structured maps are based on topic maps.

We compare the five systems (including topic maps) with our methodology using the following criteria: Expressive power, independence from a linking technology, agnosticism toward the content and granularity of linked data, and ability to generate (or express) schemas in implementation-friendly logical data models.

A note on expressive power: In this section, we consider the ability to express relationships similar to the ones possible with our patterns of use of marks. For example, with the AExcerpt pattern, we consider the ability to *express* (not implement) that the value of (an attribute of) an entity is derived from another entity.

4.10.2.1. Topic Maps

A *topic map* [68] (also called a topic navigation map [67, 158]) represents the structure of groups of addressable information objects called *topics*, and the relationships (called *associations*) between topics. An association is specified as a hyperlink in the Hypermedia/Time-based Structuring Language (HyTime) [64].

A topic has a set of properties called *facets*. "Title" and "Description" are commonly occurring facets. Special associations called *anchors* may be used to link a topic to its facets. An anchor may have a role name.

Topic maps are expressed using the Standard Generalized Markup Language (SGML) [51, 66]. The use of SGML and HyTime makes a topic map very expressive, but it also makes a topic map hard to comprehend: Popular web browsers do not support browsing SGML documents and they do not handle HyTime links. An XML syntax [69] has also been published for topic maps. That syntax uses XLink [164] to create links, but an XLink link can only address XML content.

A HyTime link is agnostic toward content and granularity of linked data, but it cannot express assignment of context information to an attribute. Finally, the topic map model does not define a means to generate schemas in logical data models (but the DTD of a topic map may be viewed as a logical schema).

4.10.2.2. Structured Maps

The *structured map model* [31] is based on the topic-navigation-map model [67, 158], and is expressed using the ER model or as an SGML document. In structured maps, a topic is called an *entity*, a topic relation is called a *relationship*, and an anchor role is called a *facet*. A structured-map entity has only one attribute, the entity's topic text.

When expressed in the ER model, structured-map entities and relationships are expressed using ER constructs of the same name. Because a structured-map entity has only one attribute, the structured map model does not use the full expressive power of the ER model.

The structured-map model does not explicitly state how facets are represented in the ER model, but in the examples Delcambre and others provide [31], a new kind of construct, akin to a relationship type involving only one entity type, is used to represent a facet. The value of a facet implicitly defines the other entity type in the relationship (which is an information selection in the base layer).

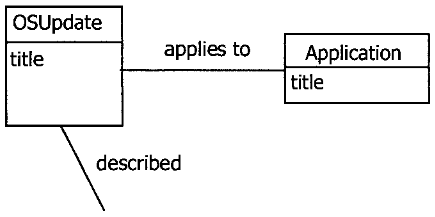


Figure 4.30: A structured map for OS updates

Figure 4.30 shows a structured map describing a relationship between an OS update and an application. The facet type described denotes an anchor into the HTML support page that describes the OS update.

Viewed as an ER schema, the structured-map model can express only the EMark pattern. However, viewed as an SGML document, the structured-map model can express the EMark, AMark, RMark, and RAMark patterns. The structured-map model cannot express the AExcerpt pattern.

A facet in the structured map model is independent of a linking technology, and is agnostic toward linked content and granularity. A structured map expressed in the ER model may be transformed into relational and XML schemas. (We assume that a facet may be represented as a relationship with an entity similar to the Mark entity in our methodology.)

4.10.2.3. Superimposed Schematics

Bowers and others [17] have proposed the *superimposed-schematic model*, an extension to the ER model to represent the use of marks. In this model, any entity or relationship may be associated with *one* mark. A relationship must be binary and it cannot have attributes. An entity's attribute may also be associated with a mark, but the value of an attribute associated with a mark is *always* the excerpt retrieved from the mark. That is, the superimposed-schematic model supports a limited form of our EMark, RMark, and AExcerpt patterns. It cannot express the AMark and RAMark patterns.

Our methodology improves upon the superimposed schematic model by removing the limitations on cardinality, and by allowing marks to be associated with attributes of both entities and relationship. We do not require the value of an attribute associated with a mark to be a base-layer excerpt. In general, we do not impose any limitations on ER-model constructs.

The superimposed-schematic model is independent of a linking technology and is agnostic towards content and granularity of linked data. The model supports bi-level querying at the conceptual level, but the extent of the base data that may be queried is limited to excerpts. The superimposed-schematic model does not include procedures to generate schemas in logical data models.

4.10.2.4. The Nested-Context Model

Casanova and others have proposed the *nested-context model* to model the structure, presentation, and navigational aspects of hypertext documents [24]. A sub-model addresses each of these aspects. The *definition sub-model* deals with the structural aspect and is related to the conceptual modeling of links. In this section, we discuss the definition sub-model due to its similarity with our work.

In the nested-context model, a *node* is an information element with a unique id. There are two kinds of nodes: terminal nodes and context nodes. A *terminal node* is a node whose content is determined and interpreted by some application. For example, a terminal node may be an image or a video. A terminal node may have *attributes* containing user-defined or application-defined information. The attribute named *contents* describes the actual (application-specific) data of the node. A terminal node is analogous to a base document.

A *context node* is a collection of terminal nodes and possibly other context nodes. A node may be contained in any number of context nodes, but it does not belong to any context node. Thus, context nodes provide a means to create multiple simultaneous organizations of nodes, and are thus analogous to SI. A context node has no attributes.

A *link* connects two nodes. In contrast to a node, a link always belongs to a specific context node. A link's endpoint is specified using an *anchor*, which is a pair *(N, s)*, where *N* is a node that forms the *base* of the anchor, and *s* is an *offset* into the content of the base. The offset may be *null* to indicate a link to an entire node. If the base is a context node, the offset is another anchor; otherwise the offset specifies a displacement within the content of a terminal node.

Links in the nested-context model support EMark and AMark relationships. (The latter type of relationship is supported only to a limited extent.) The AExcerpt pattern cannot be expressed. RMark can be simulated by first creating a context node containing the entities that participate in an anchored relationship, and then linking the newly created context node with another node. RAMark cannot be expressed, because a context node cannot have attributes.

Links in the nested-context model are independent of a linking technology, and are agnostic towards the content and granularity of linked data. However, links are not typed. In contrast, by virtue of using the ER model, relationships in our methodology are typed.

4.10.2.5. The Hypertext Design Model

The Hypertext Design Model [49] (HDM) is an extension to the ER model for modeling the structural and navigational aspects of hypertext. In this section, we review HDM2 [48], the second iteration of HDM.

The authors of HDM2 take the position that extending and reusing existing modeling techniques, and leveraging others' experience, is a better way to model hypertext systems, instead of creating new models. They extend the constructs entity type and relationship type of the ER model, and add new constructs called index type and guided tour type to facilitate easier access and focused navigation, respectively.

In HDM2, an *entity type* is a tree structure of *components,* which are sets of information elements called units. A *unit* is a concrete representation of a component. For example, an OS update component may have two units: An executable file for installation, and an HTML page with support information.

An entity type defines a named structure that is either an aggregate or a homogeneous tree. An *aggregate structure* has a *root component* and a list of member structures. A *homogeneous tree structure* has only a set of homogeneous components organized as a tree, or as a sequence, or as a singleton (one component). HDM2 does not define in what sense components may be homogeneous, but it is expected that homogeneous components define different parts of the same larger component.

A relationship is called an *application web* (or just a *web*) in HDM2. A web can relate entities and other webs, and includes a *center component* that annotates or otherwise describes the relationship. A *web type* (analogous to an ER relationship type) is the schema of a web. It has a name, a list of destination specifications, and the specification of the center component. A *destination specification* identifies an entity type or a web type, and an optional path expression to identify a particular component or unit within a destination, but only when the destination is an entity. A cardinality specification may be associated with a web type, but not with a destination. That is, cardinality constraints have different semantics than in the ER model: In HDM2, a cardinality constraint limits the number of instances of a web type; in ER, a cardinality constraint limits the number of relationships of a given type in which an entity may participate.

HDM2 introduces two constructs to enable easier access and focused navigation of a hypertext. An *index* defines a possibly heterogeneous collection of entities and components, making it easier to access specific elements and components *directly*, without traversing intervening webs. A *guided tour* is a linear path through entities and components. Indexes and guided tours may be recursive.

HDM2 can express the EMark and AMark relationship patterns, but not the AExcerpt pattern. It can also express RMark relationships because a relationship may relate entities with other relationships, but it cannot express the RAMark pattern, because a web cannot have attributes.

The entity construct of HDM2 is richer than that of the ER model, and hence richer than ours. However, the entity structure of HDM2 is motivated by the needs of hypertext networks, and the structure can become unwieldy for other (simpler) classes of applications. The framework of relationship patterns we use provides a more lightweight method of representing hierarchical structures in the ER model. Figure 4.29 gives an example.

Addressing a portion of an entity's content in HDM2 requires the use of a specific form of path expression. Also, a path expression can only select a unit (that is, a representation of a component), but it cannot select a region within a unit. For example, a path expression can select the HTML support page unit of an OS update, but it cannot select a region within the HTML page.

HDM2 does not define a way to generate schemas in logical data models, but we believe it is possible to generate an XML schema from an HDM2 schema.

4.11. Summary and Conclusions

We have presented a methodology to explicitly represent marks and the use of marks in ER schemas using a set of conventions to augment the semantics of existing ER model constructs. An SA that realizes an ER schema with these conventions can easily access the context information associated with a mark; can activate marks; and can readily express queries over combined SI and associated base information.

Our methodology strictly extends the ER model. That is, it does not reduce the expressive power of any of the traditional ER constructs. Existing tools that operate on ER schemas will be unaffected, but the tools would need to be extended to exploit the notion of relationship patterns. (The tools would not need to be "mark aware"; they just need to be "relationship-pattern aware".)

Our methodology to represent the use of marks has three *independent* parts: a model for marks and use of marks, a model for mark descriptors, and a model for context information. The model for marks and use of marks allows new relationship types and patterns of relationship types to be defined without affecting the model for mark descriptors and the model for context information. The model for mark descriptors allows new kinds of mark descriptors to be added without affecting the other two models. Similarly, the model for context information may be changed without affecting the other parts. Each part of the methodology provides a systematic way to transform a conceptual schema to logical schemas in one or more data models.

We have described five patterns of use of marks, but other patterns may emerge as SAs are developed. SA developers may define new patterns using the framework for relationship patterns. Also, we have omitted discussing a few obvious patterns. For example, we have described the AExcerpt pattern to derive the value of an entity's attribute from the excerpt of a mark. A similar pattern may be defined for relationship attributes. It is also possible to define a pattern to represent that an attribute's value is derived from a context element's value. Such a pattern would generalize the AExcerpt pattern. (In Section 4.7 we described the use of the function `context` to explicitly retrieve a context element's value from the context of a mark.)

Our model for mark descriptors can represent the specification of a link's endpoint in any linking technology. See Section 4.6. This ability allows an SA developer to choose a linking technology appropriate for SA needs. For example, the developer can choose a technology based on factors such as address robustness, granularity of information addressed, and kinds of contents addressed. He can also mix and match the linking technologies. For example, he may use XPointer pointers to address XML content, and use SPARCE to address a selection inside a PDF document [6].

Our model can represent an embedded link in any linking technology. A link is called an *embedded link* (or an *inline link*) if the specification of its endpoints is included in one of the linked entities. An *n-way* embedded link specifies *n*–1 endpoints; the last endpoint is implicitly the point of inclusion. Embedded links are directed (away from the point of inclusion) and tend to be binary. A link specified using the A tag in HTML, and a mark employed in SI, are examples of embedded links.

In our model, only the EMark pattern can express an n-way embedded link, because relationship types of that pattern may be of any degree; all other patterns express binary embedded links. (See constraints specification for each pattern in Section 4.3.)

Our model for the use of marks, and the SI systems we reviewed in Section 4.10.2, *cannot* represent stand-off links. In a *stand-off link*, the link specification is maintained separately from the linked data. Consequently, an *n-way* stand-off link specifies *n* endpoints. Our model could be extended to represent stand-off links by allowing expression of relationships among marks. Such a representation would allow different endpoints of a link to be expressed using different linking technologies.

Currently, we use a single entity type Mark to represent any mark. It is possible to extend that entity type according to base type or domain-specific type. A domain-specific type can abstract over base types, yet support additional semantics (and behavior) specific to a domain. For example, marks into patent applications may be defined as a domain-specific type, such as a *claim mark*, regardless of the format (such as HTML and PDF) of the base patent documents.

In this chapter, we have used the ER model to represent the use of marks in conceptual schemas, but our approach may be used in other models (such as the UML model) as well, because our representation is simply an application of relationship patterns, and relationship patterns are independent of modeling languages [114]. (It is possible that a modeling language does not have the power to express certain relationship patterns.)

In Section 4.8 we provided some examples of bi-level queries, but did not describe how those queries are executed. Chapters 5–9 show how bi-level queries can be executed over the information represented in logical schemas generated using our methodology.

5. Transforming Bi-level Information

In Chapter 4, we discussed modeling superimposed information (SI), marks and their use, mark descriptors, and mark contexts. We also illustrated how SI and base information (BI) may be combined, and how the combined *bi-level information* may be filtered and transformed in the relational [41] and XML [43] models using *bi-level queries* expressed in existing languages.

In this chapter, we consider a means of realizing bi-level queries. We introduce the notion of a *bi-level query system* (which is a representation scheme, or schemes, for bi-level information together with a processor for bi-level queries); set goals for a bi-level query system; and identify a strategy to meet the goals in the XML model.

Specifically, we discuss two representation schemes for XML bi-level information and analyze the effect of these schemes on query expression and query execution, especially when a large number of marks and base documents are involved. We also provide a reference model for an XML *bi-level query processor*. Chapters 6 through 9 describe in detail the different parts of a bi-level query processor.

Although we focus on bi-level querying in the XML model, many of our techniques for bi-level querying apply in other data models as well. For example, Chapter 6 illustrates how context information can be retrieved dynamically from within a relational database management system.

5.1. Introduction

Consider the following retrieval tasks in relation to a Sidepad document (such as that shown in Figure 1.3).

Q1: List the base documents that the Sidepad document references.

Q2: Extract excerpts from marks associated with items in the group named Garlic.

Q3: List the names of items in the Sidepad document.

Q4: Find the number of marks that the Sidepad document uses.

Q5: Create an HTML page from the contents of the Sidepad document.

Task Q1 requires examining the descriptor of each mark used in the Sidepad document (to obtain the path to the base document with which a mark corresponds). Q2 requires examining the context of marks attached to items in a particular group. Q3 requires examining just the name of each item in the Sidepad document, but requires no access to mark associations, descriptors or context information. Q4 requires counting the number of mark associations, but does not require examining the descriptors or context information. Q5 requires transforming the contents of the Sidepad document, possibly along with some context information, to an HTML page (such as that shown in Figure 1.5).

In general, tasks such as Q1 through Q5 require the user to filter and transform bi-level information. A user can prepare and transform bi-level information manually when the quantity of information is relatively small, but he could benefit from an automated approach when processing large information sets.

An SA developer can facilitate automation of tasks such as Q1 through Q5 by providing an API to the SA. A user can then develop *scripts* (which are interpreted programs, expressed in languages such as JavaScript [73] and VBScript [160]) that use the SA's API to examine SI and the referenced base information. For example, Sidepad can expose an API that allows a user to navigate the groups and items (contained in a document), and the referenced base information. In this approach, an end user of an SA might develop custom scripts or he might execute canned scripts that the SA developer incorporates into the SA.

With the scripting approach to automation, scripts can be developed external to an SA and executed *without* changes to the SA. However, executing a script requires a script interpreter that interacts with the SA's API using a specific technology, on a specific platform. For example, if Sidepad exposed an OLE Automation API [130], the script interpreter must be able to interact with OLE automation objects, probably on the MS Windows platform. If another SA exposed a Java [71] interface, the script interpreter would need to be capable of invoking Java methods.

An alternative is to expose bi-level information so it can be filtered and transformed using *queries* expressed in a language appropriate to the SI model. For example, as illustrated in Section 4.8, SQL [92] might be the query language if SI is in the relational model, whereas XQuery [176] or XSLT [177] might be the query language if the SI model is XML (or a model that readily maps to XML). As with the scripting alternative, an end user might develop these queries or he might execute canned queries the SA-developer incorporates into the SA.

Unlike the scripting alternative, the querying alternative allows bi-level information to be processed on any operating platform, using any implementation technology (with the data model being the only limiting factor). For example, an SA user can use any XML query language to query Sidepad data exposed in XML format, and he can use any XML query processor available on his favorite platform.

The two alternatives also differ in the style of programming a user would likely employ to automate transformation of bi-level information. The scripting approach likely requires the use of an *imperative* language such as VBScript and JavaScript (which requires the description of how a task is performed). In contrast, query languages (such as SQL) tend to be *declarative* (requiring the user to only describe what task needs to be performed; not how the task is to be performed.) For example, consider the Task Q1 to list the base documents that a Sidepad document references. In the scripting approach, the user expresses how duplicate document locations are eliminated (because a Sidepad document may contain multiple marks into a base document.) In SQL, the user simply uses the `DISTINCT` qualifier in the `SELECT` clause to eliminate duplicates.

Due to the benefits the SA developers and users can derive from it, we pursue the querying alternative to filter and transform bi-level information.

Figure 5.1 shows a reference model for a bi-level query system. Dashed arrows indicate data flow. The bi-level query system accepts SI, the referenced descriptors, and base information. It uses a set of transformers to represent the descriptors and base information in the same data model as SI, according to a schema that is conducive for bi-level querying. As mentioned in Section 3.2.2, SI may include mark descriptors directly, or include only mark IDs. Figure 5.1 shows a descriptor repository to accommodate SI that uses mark IDs.

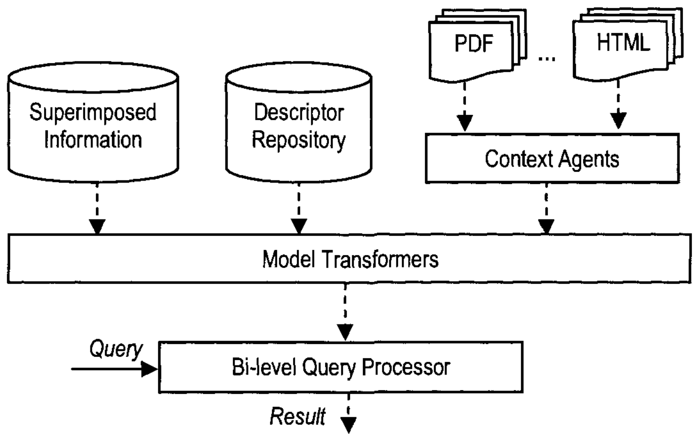


Figure 5.1: A reference model for a bi-level query system. Dashed arrows indicate data flow

We restrict a bi-level query processor to use only one data model at a time, because, in practice, choosing a query language and the data model for the result can be hard if data models are mixed.

5.2. Representing Bi-level Information

For a given data model, several logical schemas are possible for bi-level information, and schemas can vary in their support for bi-level query processing. Some schemas can make query expression easier, but can cause execution inefficiency, whereas other schemas can restrict querying capabilities.

In this section, we introduce two XML schemas for bi-level information with different degrees of support for bi-level querying. The first schema, called the *nested schema*, integrates SI, the marks referenced in the SI, and the contextual information for the marks; and presents the integrated information as a single XML document for querying. The second schema, called the *normalized schema*, separates SI, the descriptors, and the context information; and requires the user to *explicitly join* the different kinds of information (as needed) in queries.

In this section, we compare the effect of the two schemas on query expression and query-execution performance. We use the comparison to present our goals and strategies for bi-level querying (in Section 5.3).

The representation schemes are based on the developments in Sections 4.5 through 4.7. The examples we use are based on the conceptual schema presented in Figure 4.24 for the Sidepad application. As in Chapter 4, the *Sixml element types* (that is, the element types used to represent mark associations) belong to the namespace "`sixml`" and are bound to the URI "`http://schema.sixml.org`". Also as in Chapter 4, for simplicity, we use the Sixml element types without a namespace.

5.2.1. Nested Schema

In the nested schema, a *mark-association element* (that is, an element that represents the use of a mark) is nested inside an SI element. The mark-association element in turn contains the mark descriptor and the complete context information retrieved from the mark.

Figure 5.2 shows an example XML fragment in the nested schema. Some elements are shown in bold to highlight the nesting of information. The element Item and the text directly contained in that element represent SI. The element EMark_ItemMark represents a mark association. The sub-element Descriptor of the mark-association element contains a description of the associated mark. The sub-element Context represents the context information retrieved from the mark. For illustration, this sub-element includes only three kinds of context information (content, containment, and placement).

The nested schema allows a user to easily query bi-level information because SI, its associated marks, and the context information for the marks are all available together. The nesting of information allows "natural" navigation from the SI layer to the base layer. For example, the Task Q1 (list the base documents used) can be accomplished using the following XPath 1.0 [166] (henceforth referred to simply as *XPath*) expression:

```
//Item/EMark_ItemMark/sixml:Descriptor/Document/Location
[not(.=preceding::Location)]
```

This XPath expression navigates from the root of the XML document to SI (the element Item), to a mark association (EMark_ItemMark), to the descriptor of the base document (the nested element Document), and finally to the element Location that contains the path to the referenced base document. The predicate in this expression (that is, the portion enclosed in brackets) eliminates duplicate base-document locations.

```xml
<?xml version="1"?>
<SidepadDoc title="Data Integration">
 <Item name="Goal">Mediate heterogeneous data sources without replicating data.
  <EMark_ItemMark sixml:markID="23">
   <sixml:Descriptor xmlns:xs="http://www.w3.org/2001/XMLSchema-instance"
                     xmlns:sixml="http://schema.sixml.org" xs:type="SPARCEMark">
    <Agent>AcrobatAgents.PDFAgent</Agent>
    <Class>AcrobatPDFTextMark</Class>
    <Address>2|395|439</Address>
    <Description>Page 3 in f.pdf (Adobe Acrobat)</Description>
    <CachedText>provide applications and ...</CachedText>
    <Who>smurthy</Who>
    <Where>TYEE</Where>
    <When>2004-05-28 14:03:02</When>
    <Document ID="D6">
     <Agent>AcrobatAgents.PDFAgent</Agent>
     <Location>E:\Base\f.pdf</Location>
     <Application ID="Acrobat5">
      <Agent>AcrobatAgents.PDFAgent</Agent>
      <Name>Adobe Acrobat 5.0</Name>
     </Application>
    </Document>
   </sixml:Descriptor>
   <sixml:Context>
    <Content><Text>provide applications and ...</Text></Content>
    <Containment>
     <Section><Heading>3: Garlic Overview</Heading></Section>
    </Containment>
    <Placement><Page>3</Page></Placement>
   </sixml:Context>
  </EMark_ItemMark>
 </Item>
</SidepadDoc>
```

Figure 5.2: Example bi-level information in the nested schema

The nested schema has two obvious problems. First, the details of a mark (including the descriptor and the context information) are represented redundantly if the mark is used more than once. Second, the descriptors for all marks, documents, and applications, along with the context information for marks are *eagerly* materialized regardless of query needs. For example, the Task Q3 (list names of Sidepad items) can be accomplished without consulting mark descriptors or context information (using the

XPath expression `//Item/@name`), yet the descriptors and context information are materialized in this approach.

The nested schema can be inefficient also because the context information for some marks can be rather large. Depending on what context elements a context agent provides, the size of the complete context for a mark could exceed the size of its base document.

In summary, the nested schema makes it easy to express bi-level queries, but it can potentially affect query-execution performance, especially when the number of marks is relatively large.

5.2.2. Normalized Schema

The normalized schema is a *normalization* [12] of the nested schema to eliminate redundancy. In this schema, bi-level information is represented in *five* documents, one each for SI and mark associations (together), mark descriptors, document descriptors, application descriptors, and context information. Figure 5.3 shows the bi-level information of Figure 5.2 represented in the normalized schema. Dashed lines separate the documents. The elements in bold indicate references between the XML documents.

A query (such as Q3) over just SI executes efficiently in the normalized schema because the query is executed over just the SI document, but navigating from the SI layer to the base layer is cumbersome because the user must explicitly join different documents: Join queries are harder to express and they tend to be error-prone [70]. For example, completing Task Q1 would require the following XQuery query:

```
<result> {
 fn:distinct-values(
  for $a in fn:doc("SI")//Item/EMark_ItemMark,
      $m in fn:doc("Marks")//sixml:Descriptor[@ID=$a/@sixml:markID],
      $d in fn:doc("Documents")//Document[@ID=$m/DID]
  return $d/Location
 )
} </result>
```

In this query, the `for` expression ranges over mark-association elements in the SI document, and binds the variable `$a` to an EMark_ItemMark element in each iteration. For each EMark_ItemMark element in the SI document, the `for` expression binds the variable `$m` to the matching Descriptor element in the mark-descriptors document; and the variable `$d` to the matching Document element in the document-descriptors document. The expression then returns the Location sub-element from the matching Document element. The function `fn:distinct-values` eliminates duplicates in the sequence of nodes that the `for` expression returns. The namespace `fn` is bound to the URI http://www.w3.org/2005/xpath-functions [176].

Clearly, the XPath expression to perform Task Q1 (shown in Section 5.2.1 for the nested schema) is more compact, and is easier to develop and comprehend, than the XQuery query.

The normalized schema solves the problem of redundant representation of descriptors and context information, but it still eagerly materializes the complete context information for the referenced marks whenever a query references the context document. (In this discussion, for simplicity, we ignore the issue of granularity of materialization.)

```
<!-- SI document -->
<?xml version="1"?>
<SidepadDoc title="Data Integration">
 <Item name="Goal">Mediate heterogeneous data sources without replicating data.
   <EMark_ItemMark sixml:markID="23"/><!-- References the Mark-descriptors document-->
 </Item>
</SidepadDoc>
-----------------------------------------------------------------------------------------
<!-- Mark-descriptors document -->
<?xml version="1"?>
<Marks>
  <sixml:Descriptor xmlns:xs="http://www.w3.org/2001/XMLSchema-instance"
                   xmlns:sixml="http://schema.sixml.org" xs:type="SPARCEMark" ID="23">
    <Agent>AcrobatAgents.PDFAgent</Agent>
    <Class>AcrobatPDFTextMark</Class>
    <Address>2|395|439</Address>
    <Description>Page 3 in f.pdf (Adobe Acrobat)</Description>
    <CachedText>provide applications and ...</CachedText>
    <Who>smurthy</Who>
    <Where>TYEE</Where>
    <When>2004-05-28 14:03:02</When>
    <DID>D6</DID><!-- References the Document-descriptors document-->
  </sxml:Descriptor>
 ...
</Marks>
-----------------------------------------------------------------------------------------
<!-- Document-descriptors document -->
<?xml version="1"?>
<Documents>
 <Document ID="D6">
  <Agent>AcrobatAgents.PDFAgent</Agent>
  <Location>E:\Base\f.pdf</Location>
  <AID>Acrobat5</AID><!-- References the Application-descriptors document-->
 </Document>
 ...
</Document>
-----------------------------------------------------------------------------------------
<!-- Application-descriptors document -->
<?xml version="1"?>
<Applications>
 <Application ID="Acrobat5">
  <Agent>AcrobatAgents.PDFAgent</Agent>
  <Name>Adobe Acrobat 5.0</Name>
 </Application>
 ...
</Applications>
-----------------------------------------------------------------------------------------
<!-- Context document -->
<?xml version="1"?>
<sixml:Contexts>
 <sixml:Context ID="23"><!-- The ID associates context with a descriptor -->
  <Content><Text>provide applications and ...</Text></Content>
  <Containment>
    <Section><Heading>3: Garlic Overview</Heading></Section>
  </Containment>
  <Placement><Page>3</Page></Placement>
 </sixml:Context>
 ...
</sixml:Contexts>
```

Figure 5.3: Example bi-level information in the normalized schema

5.2.3. Impact of Representation Scheme on SI-only Queries

In this section, we consider the effect of the two representation schemes on *SI-only queries*, which are queries over just SI, returning only SI (possibly with newly constructed information). For example, Task Q3 is accomplished with an SI-only query (`//Item/@name`). We consider the issues in expressing and executing SI-only queries because we expect a significant number of queries to read and return only SI, and we wish to use the same mechanism to execute both bi-level queries and SI-only queries.

Expressing an SI-only query can be harder in a bi-level setting because non-SI data might need to be (explicitly) excluded from the results. For example, the result of the XPath expression `//Item` to return all Sidepad items also returns mark-association elements (and the nested descriptors and context information if applied to the nested schema) embedded inside each Item element. Thus, the user needs to write the following, more complex, XQuery query (because XPath cannot exclude the contents of an element it returns [166]).

```
<result> {
  for $i in fn:doc("SI")//Item
  return <Item name="{$i/@name}">{@i/text()}</Item>
} </result>
```

An SI-only query might execute poorly in a bi-level setting because the query processor examines unwanted non-SI information. For example, when executing the SI-only query `//Item/@name` (for Task Q3), the query processor examines all 26 elements in the nested schema (for the data in Figure 5.2), but only two of these elements represent SI. The processor examines three elements (SidepadDoc, Item, and EMark_ItemMark) in the normalized schema (applied to the SI document in Figure 5.3), though only two elements represent SI. In both cases, only one element satisfies the query.

In summary, expressing SI-only queries (and a few other classes of queries as illustrated in Chapter 8) can be hard in both schemas. Also, the queries can execute inefficiently in both schemas, and the inefficiency increases with the number of mark associations.

5.3. Goals and Strategy for Bi-level Querying

Section 5.2 has provided sufficient information to help the reader get a feel for the key issues in bi-level query processing. With that background information, we now present our goals for a bi-level query system and outline our strategy to meet the goals in the XML model.

5.3.1. Goals

We see four roles for people interacting with a bi-level query system: SA users, SA developers, bi-level query developers (that is, people who develop queries to accomplish tasks such as Q1 to Q5), and implementers of bi-level query processors. Our goal is to design a system such that the activities of all roles are made easier. We identify seven sub-goals to reach that larger goal:

G1: SI-schema independence: A representation scheme for bi-level information should not curb SI modeling. For example, it should not force the SA developer to include or exclude particular SI elements.

G2: Diversity and multiplicity: It should be possible to associate zero or more marks with any *conceptual* SI element. For example, zero or more marks should be possible for both Sidepad groups and items. It should be possible to associate zero or more marks with any *logical* SI element. For example, in the XML model, it should be possible to associate zero or more marks with elements, attributes, and text.

G3: Execution efficiency: A bi-level query system should aid efficient query execution in terms of speed. Specifically, it should not significantly hurt the performance of SI-only queries.

G4: Scalability: A bi-level query system should be able to handle queries that involve a large number of marks or marks over a large number of base documents. Specifically, the system should provide a reasonable response even for queries that involve 100,000 marks.

G5: Ease of query expression: A bi-level query system should aid "natural" navigation from the SI layer to the base layer.

G6: SI-only-query preservation: Imagine an *SI-only schema* obtained by removing the mark-association elements from the normalized schema. The result of a query executed over an instance of such a schema (for example, the SI document in Figure 5.3, but without the `EMark_ItemMark` element) must be preserved when the same query is executed over bi-level information. For example, in the XML model, the XPath expression `//Item` executed as an SI-only query must return only SI when executed over the information in either Figure 5.2 or 5.3. We focus on SI-only-queries because facilitating bi-level querying should not be at the cost of SI-only-queries (and we expect a good portion of queries to be of the SI-only kind).

G7: Language compatibility: New operators or functions should not be required in an existing query language to express a bi-level query or an SI-only query. Language compatibility ensures that query developers do not need to learn anything new to express bi-level queries, and that (parts of) existing query processors can be reused to process bi-level queries.

Our goals for a bi-level query system apply to any logical data model (such as relational and XML, with appropriate substitution and interpretation of terms). Whichever logical model is chosen, *any* SI schema in that model should be supported. However, it is possible that the logical model influences the degree to which a sub-goal is met. For example, Goal G5 might be met to different degrees in the XML and the relational models because the mechanism of navigation (setting aside its naturalness) is quite different between the two models.

Though our goals for a bi-level query system are not specific to a logical data model, we focus on designing and implementing a system for the XML model, because of the increasing popularity of XML. In addition, XML bi-level queries pose some unique problems (such as those related to SI-only queries).

With respect to the XML model, both the nested schema and the normalized schema (presented in Section 5.2) help meet Goals G1, G2. The normalized schema helps meet G3, but conflicts with G5. The nested schema helps meet G5, but conflicts with

G3. Neither schema helps Goal G4, largely because the query-execution strategy influences scalability more than a schema does. As described in Section 5.2.3, both schemas conflict with G6. Both schemas help meet G7 with respect to bi-level queries, but neither helps meet this goal with respect to SI-only queries.

Many normalized schemas are possible for bi-level information, with differences in the degree to which the sub-goals are met, but we limit our discussion to the schema used in Figure 5.3.

5.3.2. Strategy for the XML Model

Within the XML model, we use a combination of SI design-time modeling solutions and SA run-time solutions to meet the goals identified. Table 5.1 summarizes the goals and shows the different parts of the strategy employed to reach each goal.

Table 5.1: Summary of goals and strategy for bi-level querying. A number in parentheses indicates the chapter or section where the related discussion can be found

Goal	Means (Chapter or Section)
G1. SI-schema independence	Modeling (7), Sixml DOM (7)
G2. Diversity and multiplicity	Modeling, Sixml DOM
G3. Execution efficiency	Normalized schema (5.2.2), Cloaking (8), Bi-level navigator (9)
G4. Scalability	Bulk accessor (6), Cloaking, Bi-level navigator
G5. Ease of query expression	Nested schema (5.2.1), Cloaking, Bi-level navigator
G6. SI-only-query preservation	Normalized schema, Cloaking, Bi-level navigator
G7. Language compatibility	Modeling, Cloaking, Sixml DOM, Bi-level navigator

To meet Goals G1 (SI-schema independence) and G2 (diversity and multiplicity), we use the Sixml element types introduced in Sections 4.5 through 4.7 to represent bi-level information. We extend the element types for mark associations defined in Section 4.5.2 to indicate marks associated with parts of an XML document (for example, with text content and processing instruction) that the ER model cannot represent.

We also define *Sixml DOM*, an extension of the W3C XML Document Object Model (DOM) [34] to represent and manipulate a Sixml document (that is, an XML document containing instances of Sixml element types) at run time. Chapter 7 describes the extended Sixml element types and Sixml DOM.

To meet Goals G3 (execution efficiency) and G5 (ease of query expression), we require SI to include just the mark associations as in the normalized schema (for example, the first document shown in Figure 5.3), but we allow bi-level queries to be expressed over the nested schema (such as that used in Figure 5.2). If a query involves only the SI elements and the mark associations, no new data is materialized; if a query examines descriptors or context information, the necessary information is materialized *just in time*.

We design and implement a *bi-level navigator* that implements just-in-time materialization of descriptors and context information. The navigator uses Sixml DOM to *internally* represent bi-level information, but uses the W3C XPath data model [166] to *externally* represent the same information for querying purposes. With the bi-level navigator, bi-level queries may be expressed in existing languages and executed with existing traditional XML query processors. Chapter 9 describes the bi-level navigator, its data model, and its use with existing query processors.

To meet Goal G4 (scalability), we use a *bulk accessor* component to efficiently retrieve context information from a large number of marks and from marks in a large number of base documents. Chapter 6 describes the bulk accessor. Sixml DOM (and thus, indirectly, the bi-level navigator) employs the bulk accessor.

To meet Goals G6 (SI-only query preservation) and G7 (language compatibility with respect to SI-only queries), we *cloak* (that is, make invisible) mark associations from SI-only queries, and exclude cloaked information from query results. Chapter 8 describes a formal model for cloaking data and for executing queries over cloaked data. The bi-level navigator supports cloaking.

Figure 5.4 shows a reference model for an XML bi-level query processor that employs the strategy outlined. The modules shaded gray denote traditional XML query processors using the bi-level navigator to support bi-level querying.

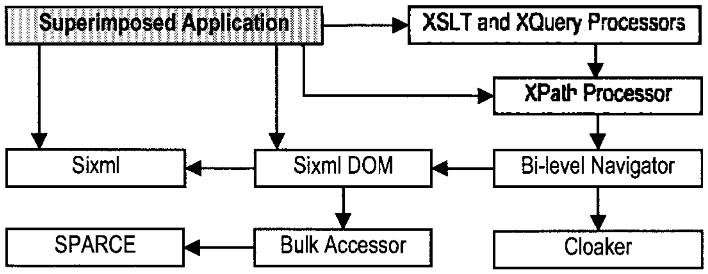


Figure 5.4: A reference model for an XML bi-level query processor. Arrows denote dependency. A gray module denotes an existing traditional XML query processor

5.4. Summary and Conclusions

In this chapter, we have introduced the notion of a bi-level query system to help filter and transform bi-level information using queries in existing languages. We have presented two alternative representation schemes (namely, nested and normalized schemas) for XML bi-level information, and explored how each scheme aids or affects query expression and execution. We have also illustrated that SI-only queries deserve special attention when designing a bi-level query system.

We have identified seven goals for a bi-level query system, and presented a strategy to meet the goals in the XML model. At the heart of the strategy is the bi-level navigator, which allows an SA developer to model SI and mark associations in the normalized schema, but permits queries expressed over the nested schema. The navigator employs the other solutions identified to allow expression and execution of queries using existing query processors unchanged.

Chapters 6 through 9 describe the different parts of a bi-level query system. Each of those chapters includes a description of related work.

6. Optimizing Bulk Access to Context Information

Scalability (the ability to extract context information from a large number of marks and base documents for a single query) is one of the goals we set in Chapter 5 for a bi-level query processor. In this chapter, we describe a component called the *bulk accessor* [121] that is specifically designed to achieve this goal. The bulk accessor supports different policies that a query processor can exploit to improve performance depending on data characteristics such as clustering of marks.

In this chapter, we illustrate the use of the bulk accessor from within a relational query processor. Chapter 9 shows the use of the bulk accessor in an XML query processor.

6.1. Introduction

Imagine that the peer-review commenting of papers submitted to a conference is managed using the Superimposed Scholarly Review System (*SISRS*) application introduced in Section 4.9.2. Assume that the conference receives 500 submissions (which is plausible: The conference *VLDB 2006* had 624 submissions [30]). If each paper is reviewed by three reviewers, and if each reviewer comments on 10 distinct regions of each paper, 15,000 marks would be created in total. In this setting, some queries may combine the superimposed comments with context information retrieved from the commented regions. For example, the query shown in Figure 4.27 to prepare a draft of feedback to authors retrieves excerpts from commented regions. As illustrated in Sections 4.7 and 4.8, a query processor can use the functions `excerpt` and `context` to retrieve context information, but retrieving context information from 15,000 marks can consume an unacceptable amount of time, if these functions are implemented naïvely.

In the rest of this section, we establish the need for giving special consideration to retrieving context information, using bi-level queries, for marks stored in a database (DB). Specifically, we show that the process a typical superimposed application (SA) uses to retrieve context information in an *interactive setting* is impractical for bulk access in a DB setting. For simplicity, we limit this discussion to retrieving excerpts using the function `excerpt`, but a similar discussion holds for the function `context`.

Figure 6.1 shows a sequence diagram drawn using the Unified Modeling Language (UML) [159] syntax. It shows the *interactive sequence* of tasks an SA and our middleware *SPARCE* (described in Chapter 3) perform to retrieve the text excerpt from a mark, as well as the task sequence to release a context-agent instance. Tasks initiated by non-human actors are numbered. A total of eight tasks are involved in retrieving an excerpt. Releasing a context-agent instance involves three tasks.

All 11 tasks in the interactive sequence may not be needed to retrieve the excerpt from every mark because the typical SA releases a context-agent instance only after the user closes the SA, not after each use of the instance. Consequently, later instances of a context-agent class may benefit from the work done by earlier instances. For example, only the first instance of the context agent for marks into a PDF document [6] might load the Adobe Acrobat (Acrobat) [8] application, eliminating Task 4 for later instances. Similarly, a base document loaded for one mark may be reused for other marks into that document (avoiding Task 5). Some choices in context-agent implementation and constraints in base applications can influence the specific set of tasks performed, but in general, applications and documents—once loaded—can be reused.

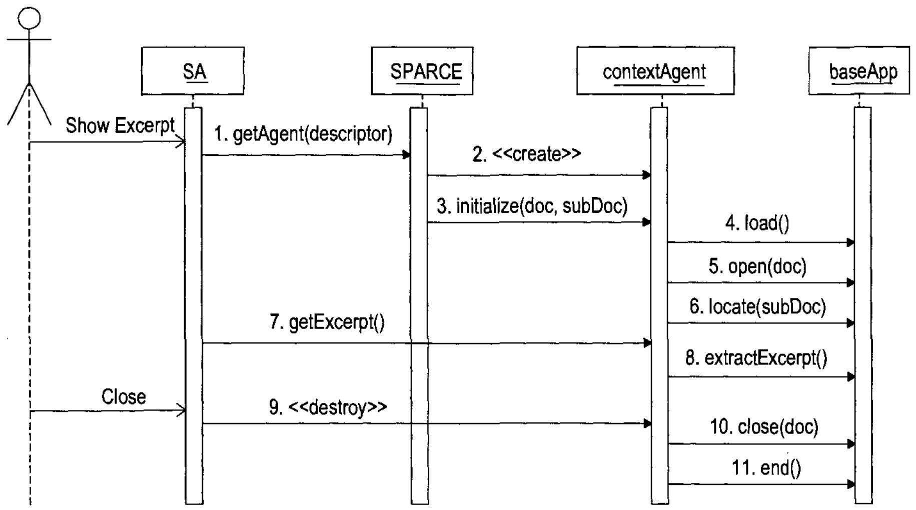


Figure 6.1: Sequence of tasks to retrieve an excerpt from a mark in an interactive setting

The interactive sequence is practical for use by an SA, but it is impractical for bulk access needed to execute a query. For example, consider the following query in the Structured Query Language (SQL) [92] to retrieve the ID and excerpt from each mark, using the schema shown in Figure 4.18(a):

```
SELECT ID, excerpt(ID) FROM Mark
```

To execute this query, the query processor invokes the user-defined function (UDF) `excerpt` for *each* mark. Assume that the UDF naïvely performs all 11 tasks of the interactive sequence, so that it operates correctly regardless of the calling context and the number of invocations.

Table 6.1 shows the time (measured) to retrieve excerpts from four marks using the interactive sequence. The first two marks reference distinct regions of a PDF document; the last two marks reference distinct ranges in a Microsoft Excel (Excel) [96] workbook. The column "SA" indicates the time required to retrieve excerpts using the interactive sequence via an SA; the column "DB" denotes the time to retrieve excerpts with the aforementioned SQL query, using a naïve implementation of the function `excerpt` in Microsoft SQL Server 2005 (MSSQL) [99]. Table 6.1 also shows the total time and the average time to retrieve excerpts for the four marks. The time to initialize context agents shown in the table is discussed in Section 6.2.

Table 6.1: Time (in milliseconds) to retrieve excerpts and to initialize context agents using the interactive sequence

		Time to retrieve excerpt		Time to initialize context agent	
Mark	**Document**	**SA**	**DB**	**SA**	**DB**
M1	P1.pdf	2172	2281	2157	2141
M2	P1.pdf	79	2218	79	2078
M3	E1.xls	250	329	234	250
M4	E1.xls	15	297	15	234
Total time (ms)		2516	5125	2485	4703
Average time (ms)		629	1281	621	1176

According to Table 6.1, the SA and the DB approaches consume about the same time to retrieve the excerpt for the first mark of each document in this dataset. However, the naïve DB approach consumes far more time for the second mark of each document, because it repeatedly opens and closes the base application and document. At the rate shown for PDF marks in Table 6.1 (2.2 seconds per PDF mark), the naïve DB approach would need over nine hours to retrieve excerpts from 15,000 PDF marks.

6.2. Bulk Access Considerations

We now analyze the similarities and the differences between the SA and naïve DB approaches to retrieving excerpts using the interactive sequence. We use this analysis to formulate the key considerations for bulk access.

For each mark, both approaches instantiate a context agent (Tasks 1–6), retrieve the excerpt (Tasks 7 and 8), and release the context-agent instance (Tasks 9–11). However, they differ in the amount of work performed when a context agent is instantiated (Tasks 4–6) and when a context-agent instance is released (Tasks 10 and 11). They also differ in the ordering of these tasks.

Our observations show that Tasks 4, 5, 10, and 11 consume a majority of the time needed to retrieve the excerpt from a mark. Consequently, we use the following equations to approximate the time taken to retrieve excerpts for all marks.

$$t_{SA} = \sum_{a=1}^{A} t_{Load(a)} + \sum_{d=1}^{D} t_{Open(d)} + \sum_{d=1}^{D} t_{Close(d)} + \sum_{a=1}^{A} t_{End(a)} \qquad \textit{Equation} \quad 5.1$$

$$t_{DB} = \sum_{a=1}^{M} t_{Load(a)} + \sum_{d=1}^{M} t_{Open(d)} + \sum_{d=1}^{M} t_{Close(d)} + \sum_{a=1}^{M} t_{End(a)} \qquad \textit{Equation} \quad 5.2$$

Equation 5.1 estimates the total time t_{SA} needed to retrieve excerpts from M marks using the interactive sequence via an SA. Equation 5.2 estimates the total time t_{DB} needed to retrieve excerpts from M marks using the naïve DB approach. A is the number of distinct base applications, and D is the number of distinct base documents. The inequalities $M \geq D \geq A$ hold because each mark is made in exactly one document; and each document is opened using exactly one application. *Load* and *End* are functions over base applications; *Open* and *Close* are functions over base documents.

In Equations 5.1 and 5.2, the first two terms correspond to Tasks 4 and 5, respectively. These tasks are performed as a part of Task 3, initialization of a context-agent instance. The third and fourth terms correspond to Tasks 10 and 11, respectively, and are performed as a part of Task 9, destruction of a context-agent instance. These equations show that the SA and DB approaches differ in the number of times base applications are loaded (and ended) and in the number of times base documents are opened (and closed).

Although not captured by these equations, the SA and DB approaches also differ in the number of *simultaneous* instances of marks and context agents in memory. At the end of retrieving excerpts from all *M* marks, the SA approach maintains *M* context-agent instances. In contrast, the DB approach maintains only one context-agent instance in memory at any time.

Table 6.1 includes the time taken to perform Tasks 3–6 (initialize context agents) for the four marks described in Section 6.1. It shows that in both the SA and DB approaches, initializing the context agent consumes a significant portion of the time needed to retrieve the excerpt from a mark.

With this information at hand, we discuss two conflicting considerations for bulk access: delaying context agent destruction (to reduce the time taken to repeatedly initialize context agents) and limiting the number of context-agent instances (to reduce memory consumption). We also discuss clustering marks by base documents as a means of balancing the resource tradeoffs due to the conflicting considerations.

The repeated destruction and initialization of context-agent instances is one reason the naïve DB approach consumes more time. Thus, the approach could perform better if context-agent destruction is delayed until the end of a query, but a database management system (DBMS) might limit the number of object instances that may be created within a query batch. (A *query batch* is a sequence of queries executed as one unit.) For example, MSSQL limits the number of ActiveX object instances per batch to 256 [147]. That is, a query can retrieve excerpts for at most 256 marks using the interactive sequence, if context agents are implemented as ActiveX classes [93].

Also, delaying context-agent destruction until the end of a query may not scale up for a large number of marks. For example, in our implementation, a context-agent instance requires at least 512 bytes of memory. At this rate, maintaining 15,000 context-agent instances simultaneously would require over 7 MB of memory *per query*, excluding the memory needs of base applications, base documents, and the DBMS. Ordinarily, a DBMS can easily afford such amounts of memory, but the operating system (OS), not the DBMS, manages the memory for context agents and base applications. As a result, the number of simultaneously executing queries might be limited.

In addition to the potential problem with scaling, too many simultaneous context-agent instances can also adversely affect the speed of bulk access: As the number of simultaneous context-agent instances, loaded base applications, and open base documents increases, the OS furnishes the various processes with more and more virtual memory, which can induce overhead due to disk operations. The amount of overhead induced depends on factors such as data-access patterns and the virtual-memory caching policy. For example, the overhead might be small if consecutive marks reference the same base document, and the base document is already resident in physical memory.

Clustering marks by base documents is a way to manage the tradeoff between delaying destruction of context-agent instances and the number of simultaneous agent instances. Clustering allows a context-agent instance to be reused for all marks in one document before the instance is reused for marks in other documents. It also exploits the OS's affinity for locality of reference because the base document is more likely to be resident in physical memory for the entire duration of its use. Thus, clustering can reduce the time taken to initialize an agent instance.

Clustering marks by base documents also allows a base document to be closed immediately after all its marks are processed, potentially reducing the stress on memory.

6.3. Design

In this section, we present the design of a bulk accessor that has the following features:

- Requires the query processor to create only one object instance (that of the bulk-accessor) per query batch, thus avoiding DBMS limits on the number of active objects.
- Pools context agents to share base applications and documents.

- Offers different pooling policies the query processor may exploit to improve performance depending on data characteristics such as clustering of marks.

Figure 6.2 shows the architecture of the bulk accessor as a UML class diagram. The shaded classes are existing components of a DBMS. The class Bulk Accessor maintains a pool of context agents. The pool can be implemented as a hash table. The hash key depends on the pooling policy used. Section 6.3.1 describes the available pooling policies.

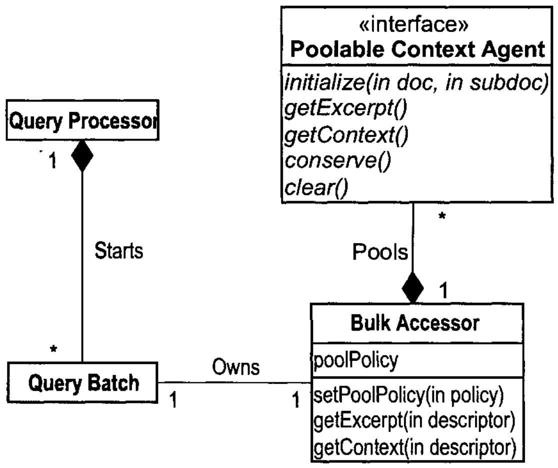


Figure 6.2: Architecture of the bulk accessor

The interface Poolable Context Agent defines the methods a context agent must implement in order to support bulk access. The method initialize assigns a document location and a sub-document address to a context-agent instance. This method informs the context agent which mark the bulk accessor intends to access. The bulk accessor may invoke this method several times in the same context-agent instance, and the values for document location and sub-document address can vary with each invocation. When this method is invoked, the context-agent instance should "smartly" reuse results of previous invocations. For example, if the document location remains constant (but the sub-document address varies) between successive invocations, the context agent should attempt to reuse the previously opened base-document instance.

The bulk accessor typically invokes the method getExcerpt or getContext after invoking initialize in a context-agent instance. These methods may be called any number of times between successive invocations of initialize, and initialize is always invoked (at some point) before retrieving excerpt or other context information.

The bulk accessor uses the methods conserve and clear to manage memory. When the method conserve is invoked, a context-agent instance should release "non-essential" resources, but be able to retrieve context information without the bulk accessor invoking initialize again. In principle, a context-agent instance can release all information except the document location and sub-document address supplied in the most recent invocation of initialize, but the instance may choose to retain other information as well (at its discretion). The bulk accessor may invoke conserve several times in a row to indicate that more resources be freed, if possible.

When the method clear is invoked, a context-agent instance should release *all* resources. After invoking clear in a context-agent instance, the bulk accessor must invoke the method initialize before using the instance to retrieve context information. Invoking clear is equivalent to destroying and recreating a context-agent instance, but without incurring the complete destruction and construction expenses.

Figure 6.3 shows the *bulk-access sequence* of tasks performed to retrieve text excerpts from marks using the bulk accessor. The figure shows three groups of tasks. Each task is numbered to denote the group to which it belongs. The first group of tasks is carried out when the query processor receives a query batch; the second group of tasks is performed for each mark; and the third group of tasks is performed when the excerpts for all marks have been retrieved, and the query batch is completed. The task startBatch creates an instance of the bulk accessor, which in turn creates an empty pool of context agents. The task endBatch destroys the bulk accessor, which in turn clears the agent pool. Clearing the agent pool involves closing base documents and ending base applications.

We first discuss different pooling policies and then discuss how a query processor may choose a pooling policy.

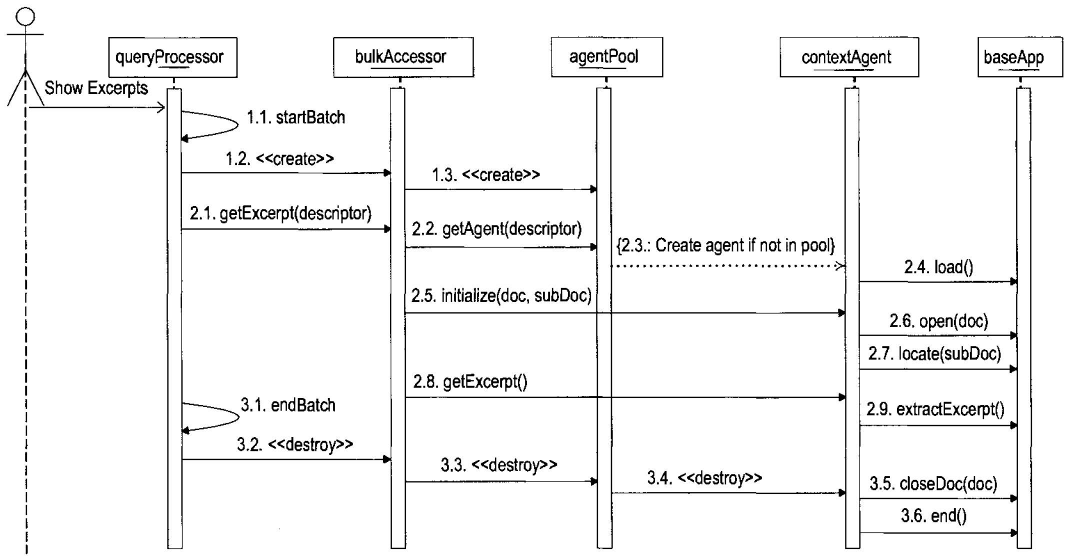


Figure 6.3: Sequence of tasks to retrieve excerpts from marks using the bulk accessor

6.3.1. Pooling Policies

The bulk accessor offers *five* pooling policies: Context-agent class, Document, Sub-document, Interactive SA, and Interactive DB.

The policy *Context-agent class* (P_{Agent}) uses one instance of each context-agent class. The same instance is used to retrieve excerpts for all marks that use that class. The name of the context-agent class is used to determine if two marks use the same class.

The policy *Document* (P_{Doc}) uses one context-agent instance per combination of base-document location (for example, a file path) and context-agent class. Excerpts for all marks of a base document that use the same context-agent class are retrieved using a single context-agent instance. The location of the document is used to determine if two marks belong to the same base document. We use the combination of document and context-agent class because marks into the same document may employ different context-agent classes. See Section 3.3.2.

The policy *Sub-document* (P_{Sdoc}) uses one context-agent instance for a combination of base-document location, sub-document address, and context-agent class. That is, the excerpts for different marks pointing at the same region of a base document, and using the same context-agent class, are retrieved using a single context-agent instance. (Two users creating marks independently might create distinct marks pointing at the same region of a base document.)

The policy *Interactive SA* (P_{SA}) uses one context-agent instance per mark. It creates two context-agent instances even if two marks point at the same region of a base document and use the same context-agent class. This policy emulates the SA approach in the interactive sequence.

The policy *Interactive DB* (P_{DB}) creates one context-agent instance for each call to retrieve an excerpt, and destroys the context-agent instance soon after retrieving the excerpt. This policy emulates the naïve DB approach in the interactive sequence.

In all policies, except P_{DB}, each base application is loaded (and ended) only once, and each base document is opened (and closed) only once. Some base applications, for example Acrobat, limit the number of base documents that may be open at once, but, for simplicity, we ignore that case for now (and consider it in the experimental evaluation described in Section 6.4.2).

To estimate the time to retrieve excerpts for all marks, for all policies except P_{DB}, we consider only the tasks in the second and third group in Figure 6.3. We disregard the first group of tasks because the time to execute them is negligible. The following equation approximates the total time t_{BA} needed to retrieve excerpts from M marks using the bulk-access sequence:

$$t_{BA} = \sum_{a=1}^{A} t_{Load(a)} + \sum_{d=1}^{D} t_{Open(d)} + \sum_{d=1}^{D} t_{Close(d)} + \sum_{a=1}^{A} t_{End(a)} + \sum_{\theta=1}^{\Theta} t_{Switch(\theta)} \qquad \textit{Equation } 5.3$$

The first four terms in Equation 5.3 are the same as those in Equation 5.1. The fifth (new) term indicates the effort to reuse existing context-agent instances to retrieve excerpts. The function *Switch* denotes the process where a context-agent instance is reinitialized for use with a mark: When the policy is P_{Agent}, a context-agent instance may need to switch to a different document or sub-document before extracting the excerpt for a mark; when the policy is P_{Doc}, a context-agent instance may need to switch to a different sub-document. The symbol Θ denotes the total number of switches context-agent instances make to retrieve the excerpts for all marks. This parameter depends on the pooling policy. For P_{Agent} and P_{Doc} this parameter is also dependent on the order in which marks are processed.

The tasks in the third group shown in Figure 6.3 are executed *after* retrieving excerpts from all marks. Consequently, the query processor can perform this group of tasks after delivering query results, without affecting the user's ability to process the results. Thus, we can simplify Equation 5.3 as follows:

$$t_{BA} = \sum_{a=1}^{A} t_{Load(a)} + \sum_{d=1}^{D} t_{Open(d)} + \sum_{\theta=1}^{\Theta} t_{Switch(\theta)} \quad \textit{Equation} \quad 5.4$$

In addition to affecting the time to retrieve excerpts, a pooling policy also affects the *pool size,* which is the maximum number of simultaneous context-agent instances maintained, denoted by the symbol K. The number of switches Θ is inversely proportional to the pool size K. If marks are uniformly distributed, the number of switches Θ is $\lceil M/K \rceil$. By uniform distribution of marks we mean the following: Marks are uniformly distributed among context-agent classes when the policy is P_{Agent}; marks are uniformly distributed among base documents when the policy is P_{Doc}; and so on.

Table 6.2 shows the relationship between pool size and the number of context-agent switches for different pooling policies. We assume that marks are uniformly distributed and that the excerpt from a mark is retrieved exactly once for each query. We use the symbol C to denote the number of distinct context-agent classes employed in a dataset, and the symbol S to denote the number of distinct sub-documents. The inequalities $M \geq S \geq D \geq C$ hold. The number of switches for the policies P_{SA} and P_{DB} is zero because a context-agent instance is never reused in these cases.

Table 6.2 shows that the policy P_{Agent} is likely to have the most switching cost, but it results in the fewest simultaneous context-agent instances after P_{DB}. The policy P_{SA} has no switching cost, but it maintains the most number of simultaneous context-agent instances.

Table 6.2: Pool size and the number of context-agent switches for different pooling policies. Uniform distribution of marks is assumed. Example values for 1000 marks are also shown

Policy	Pool size K	Example K for 1000 marks	Number of switches Θ	Example Θ for 1000 marks
P_{Agent}	C	4	$\lceil M/C \rceil$	250
P_{Doc}	D	10	$\lceil M/D \rceil$	100
P_{Sdoc}	S	100	$\lceil M/S \rceil$	10
P_{SA}	M	1000	0	0
P_{DB}	1	1	0	0

6.3.2. Choosing a Pooling Policy

We first discuss choosing a pooling policy heuristically and then discuss some of the issues in making the choice analytically.

6.3.2.1. Choosing a Pooling Policy Heuristically

Table 6.3 lists some data characteristics and predicts pooling policies that will result in the fastest execution time. The column "Clustering" denotes the attributes by which marks are clustered. The column "Distribution" describes some aspect of distribution such as number of marks per document and number of documents. The column "Policy" lists pooling policies determined heuristically.

Table 6.3: Data characteristics and pooling policies predicted using heuristics

Clustering	Distribution	Policy
Sub-document	Does not matter	P_{Agent}
Document	Many marks per document, few documents	P_{Doc}
Document	Many documents	P_{Agent}
Any other	Many uses of the same mark	P_{Sdoc}
Any other	Many references to the same sub-document	P_{Sdoc}
Any other	Many marks, few marks per document	P_{Agent}
Any other	Few marks, few marks per context-agent class	P_{SA}
Any other	Few marks	P_{Agent}

The policy P_{Agent} can provide the best performance when marks are clustered by document location and sub-document address (called "clustering by sub-document" for simplicity) for two reasons: It maintains a small pool, and it reduces the switching cost because all marks into a document are processed completely before processing marks into another document.

The policy P_{Agent} can also provide the best performance when marks are clustered by document only. However, if the number of distinct base documents is small (especially, if the number of documents is not much more than the number of context-agent classes), the policy P_{Doc} may be better as it would reduce the switching cost.

If marks are not already clustered by sub-document or document, they may be clustered appropriately before retrieving marks. In many cases, the savings obtained by clustering marks can far exceed the cost of clustering. However, it may not always be beneficial, or possible, to cluster marks. For example, clustering marks early in the query process may prevent the later use of some efficient join algorithms. The clustering of marks represented in XML cannot be changed using XPath 1.0 (because an XPath 1.0 expression cannot reorder its input; Hlousek [59] has demonstrated that an XPath 2.0 expression can reorder its input, albeit in an imperative fashion).

When marks cannot be clustered, an appropriate pooling policy can be determined based on the distribution of marks. If the cost of estimating (or computing) the distribution of marks is excessive, the policy P_{Agent} is probably the best default choice.

6.3.2.2. Issues in Choosing a Pooling Policy Analytically

We do not build or evaluate an analytical model to choose a pooling policy because there are several impediments to building such a model. Instead, we use the heuristics described in Section 6.3.2.1, and our experiments with the bulk accessor (described in Section 6.4.2) show that the heuristics produce satisfactory results in many cases.

To choose a pooling policy analytically, the query processor needs to only compare the value of the last term in Equation 5.4 among the pooling policies, because the first two terms are generally independent of the pooling policy. However, there are two impediments to computing the last term: estimating the number of context-agent switches Θ, and estimating the time to switch (per individual mark). In the rest of this section, we provide an overview of these impediments and some possible means to overcome them.

To compute the number of switches Θ, the query processor must estimate the number of marks, context-agent classes, base documents, and sub-documents at query-optimization time, but it can estimate only the number of marks (as a part of estimating selectivity). It cannot estimate the other parameters because doing so requires an examination of the descriptors of the marks involved in the query, but the exact set of marks involved in the query is known only at query-execution time.

A solution to the problem with estimating the number of switches is to index the mark descriptors and use the index to estimate the values required to predict the number of switches, regardless of the set of marks involved in the query. This solution assumes that the distribution of marks involved in a query is similar to the distribution of all known marks.

Estimating the time to switch a context agent from one mark to another is a hard problem because the switching time depends on parameters such as the number of marks, the distribution of marks, and the order in which marks are processed. The query processor can estimate the number of marks and enforce the order in which marks are processed, but it cannot estimate the distribution of marks.

The time to switch a context agent also depends on the corresponding base application. The query processor can use a table of time estimates to switch between marks for different base applications, but this approach requires knowledge of intimate details of context-agent implementations, in conflict with our desire to separate the details of retrieving context information from the actual query processing. Even then, this approach requires that the query processor know the exact set of marks that are involved in a query.

An alternative solution is to maintain profiles of query workloads with associated pooling policies. The query processor can then choose a pooling policy based on the profile that matches a given query (or based on the profile the user assigns the query). This alternative has some of the elements of a *learning query optimizer* such as LEO [150]. (A learning query optimizer improves its estimates by comparing estimated values with actual values.)

6.4. Evaluation

In this section, we provide an overview of an implementation of the bulk accessor; show how it is integrated into a traditional relational query processor; and present the results of an experimental evaluation of our bulk-accessor implementation under the various pooling policies.

6.4.1. Implementation

We have implemented the design for the bulk accessor described in Section 6.3 as an ActiveX server using Microsoft Visual Basic 6.0 [101]. The implementation supports all the pooling policies described in Section 6.3.1, and allows a query processor to indicate if, and how, marks are clustered.

Context-agent implementations are not required to implement the interface Poolable Context Agent (described in Section 6.3) but implementing it can improve bulk-access performance. We have extended all our context-agent implementations mentioned in Section 3.6.1.2 to implement this interface.

The following list provides some high-level implementation statistics (as of this writing).

- Number of interfaces: 2
- Number of classes: 1 (new, the bulk accessor); 6 (context-agent classes extended)
- Number of source files for the new classes and interfaces only: 3
- Number of new lines of code (new code and extended code): 1010
- Estimated time spent on implementing the bulk accessor and extending the context-agent implementations: 112 hours

We have used the bulk accessor to execute bi-level queries using the MSSQL relational query processor and using the XML query processors included in Microsoft's distribution of the .NET Framework [129]. Here, we provide an overview of the integration of the bulk accessor into MSSQL and illustrate its use in SQL queries.

Figure 6.4 shows a simplified version of the Transact-SQL [147] code used to integrate the bulk accessor into MSSQL. (*Transact-SQL* is Microsoft's implementation of SQL.) The text with gray background is comments. Key parts of the implementation are shown in bold.

```
--Return a handle to a new instance of the bulk accessor. Return 0 on failure
CREATE FUNCTION dbo.bulkAccessor(@policy int) RETURNS int AS
BEGIN
  DECLARE @object int, @hr int

  -- Instantiate bulk accessor (implemented as an ActiveX class), set pooling policy
  EXEC @hr = sp_OACreate 'SPARCEBulkAccess.Accessor', @object OUT
  IF @hr <> 0 RETURN 0 --bulk accessor creation failed
  EXEC @hr = sp_OASetProperty @object, 'poolPolicy', @policy
  IF @hr = 0 RETURN @object ELSE RETURN 0
END
```

(a)

```
--Return a table with one row and one column ('bulkAccessor')
--The one row has a handle to a new instance of the bulk accessor
CREATE FUNCTION dbo.bulkAccessorTable(@policy int) RETURNS TABLE  AS
RETURN (SELECT dbo.bulkAccessor(@policy) AS bulkAccessor) --reuse the scalar UDF
```

(b)

```
--Return the text excerpt from the specified mark using the bulk accessor supplied
CREATE FUNCTION dbo.excerpt(@doc varchar(1024), @sDoc varchar(256), @bulkAccessor int)
RETURNS varchar(max) AS
BEGIN
  DECLARE @result varchar(max)    --result of this function
  DECLARE @hr int --result of OLE Automation functions
  DECLARE @src varchar(255), @desc varchar(255) --error information

    --use the bulk accessor method to retrieve text excerpt for mark
  EXEC @hr = sp_OAMethod @bulkAccessor, 'getExcerpt', @result OUT, @doc, @sDoc
  IF @hr <> 0
  BEGIN
    --call failed, retrieve error information
    EXEC sp_OAGetErrorInfo @bulkAccessor, @src OUT, @desc OUT
    SELECT @result = 'Error: ' + @desc + ' (' + CONVERT(varchar, @hr) + '; ' + @src +')'
  END

  RETURN @result
END
```

(c)

```
--Example query: For each SPARCE mark, list the ID and the retrieved text excerpt.
--The last SELECT clause uses the same bulk accessor instance for each mark.
--The DBMS destroys the bulk accessor instance after the query batch is completed.
DECLARE @bulkAccessor int
SELECT @bulkAccessor = dbo.bulkAccessor(0)
SELECT MarkId, dbo.excerpt(Location, Address, @bulkAccessor) As Excerpt
FROM SPARCEMark JOIN CONTAINER ON CID
```

(d)

```
--Example query: For each SPARCE mark, list the ID and the retrieved text excerpt.
--The UDF in the FROM clause creates a bulk accessor instance for the entire query.
--The UDF in the SELECT clause uses the same bulk accessor instance for each mark.
--The DBMS destroys the bulk accessor instance after the query is completed.
SELECT MarkId, dbo.excerpt(Location, Address, bulkAccessor) As Excerpt
FROM SPARCEMark JOIN DOCUMENT ON DID, dbo.bulkAccessorTable(0)
```

(e)

Figure 6.4: Simplified Transact-SQL code to integrate the bulk accessor into Microsoft SQL Server 2005. Text with gray background is comments. Bold text shows code that operates on the bulk accessor. Code to set the pooling policy is omitted for brevity: (a) Scalar UDF to instantiate the bulk accessor; (b) Table-valued UDF to instantiate the bulk accessor; (c) UDF to retrieve text excerpt using an instance of the bulk accessor; (d) Example use of the bulk accessor in a query expressed over the schema in Figure 4.18; (e) A query equivalent to the query in Part (d), but expressed using the table-valued UDF

Figure 6.4(a) shows the definition of the UDF `dbo.BulkAccessor` to create an instance of a bulk accessor and return a handle to the new instance. The parameter `@policy` indicates the pooling policy to use. The functions `sp_OACreate` (instantiate an ActiveX class) and `sp_OASetProperty` (set a property of an ActiveX object) that this UDF uses are built into MSSQL.

Figure 6.4(b) defines the table-valued UDF `dbo.BulkAccessorTable`. This UDF defines a table with one column and returns a table with one row. The lone cell in the returned table will contain a handle to a new bulk accessor instance. This UDF is useful in associating a bulk accessor instance with a query (as Figure 6.4(e) illustrates).

Figure 6.4(c) defines the UDF `dbo.excerpt` to retrieve the text excerpt from a mark, via the bulk accessor. This UDF accepts the location of a document, the address of a sub-document, and a handle to a bulk accessor instance. The function `sp_OAMethod` built into MSSQL is used to invoke the method getExcerpt in the bulk accessor.

Figure 6.4(d) illustrates the use of the bulk accessor to retrieve excerpts from all marks in the table `SPARCEMark`. (See the relational schema in Figure 4.18.) The query in this figure first obtains a handle to an instance of the bulk accessor and sets the pooling policy to P_{Agent} (denoted by the value `0` for the parameter `@policy`). It then uses the handle repeatedly to retrieve text excerpts. The attributes `Location` and `Address` denote base-document location and sub-document address, respectively.

Figure 6.4(e) shows another use of the bulk accessor to retrieve excerpts from all marks in the table `SPARCEMark`. This query batch is equivalent to the batch in Figure

6.4(d), except that it uses the table-valued UDF to create and initialize the bulk accessor. The attribute `bulkAccessor` references the lone attribute that the table-valued UDF defines.

6.4.2. Experiments

We now present the results of experimentally evaluating the bulk accessor with four datasets: tiny, Sidepad, SISRS, and SSIB. Table 6.4 gives an overview of the four datasets. The *tiny dataset* has only eight marks, but it demonstrates the utility of the bulk accessor even for small datasets. The *Sidepad dataset* involves marks over a variety of base types used in different Sidepad documents (created over a 3-year period). The *SISRS dataset* corresponds to the application Superimposed Scholarly Review System (SISRS) introduced in Section 4.9.2. The *SSIB dataset* corresponds to the SA Superimposed System Information Browser (SSIB) outlined in Section 4.2.

Though our design allows different context-agent implementations for different marks over the same base type (and for different marks into the same base document), in our experiments, we used only one context-agent class per base type. For example, we used one context-agent class for PDF marks, and one class for Excel marks.

In all experiments, the PDF context agent used Acrobat 6.0 (Professional Edition) [8] to retrieve excerpts; the Excel, Microsoft Word (Word), and Microsoft PowerPoint (PowerPoint) agents used applications from the Microsoft Office 2002 suite [96]; and the XML agent used Microsoft XML Software Development Kit 4.0 [107].

We used a standalone *driver application* to collect experimental data, instead of using a DBMS, because the query processor in MSSQL does not allow us to collect performance data at the granularity we need for evaluation. For example, we cannot collect the data needed to plot Figure 6.8. We cannot instrument the query processor because we do not have access to its source code. However, we have verified that the results presented in this section are consistent with the results obtained by running retrieval queries within MSSQL. Section 6.4.2.2 provides example results of using the bulk accessor in MSSQL.

All experiments were run on an Intel Core Duo 1.66 GHz processor [65] with 1 GB of main memory. The OS was Microsoft Windows XP (Service Pack 2) [104]. Each experiment was run thrice, and the average result for each experiment is presented.

Table 6.4: Overview of the datasets used to evaluate the bulk accessor

Dataset	Context-agent classes (C)	Documents (D)	Sub-documents (S)	Marks (M)	Characteristics
Tiny	2	4	8	8	Few marks per context-agent class
Sidepad	4	56	490	2735	Many marks to the same sub-document
SISRS	1	426	15,336	15,336	Many documents, many marks
SSIB	3	25	105,678	107,622	Many marks per document

6.4.2.1. The Tiny Dataset

Table 6.5 lists the eight marks in the tiny dataset and shows the time (in milliseconds) to retrieve the excerpt from each mark for three pooling policies. The column "Sub-document" shows the addresses of the marked sub-documents in the dataset. For PDF marks, this column shows the page number and the index of the first and last words in the marked region. For Excel marks, it shows the spreadsheet name and the name of the cell in the marked region. (Only one cell was marked in each case.) The annotations in the first four rows describe the behavior of each policy. The last row shows the time to clear the pool after excerpts are retrieved from all marks. The marks were processed in the order shown.

The policy P_{DB} requires 10.3 seconds to retrieve excerpts from all marks in the tiny dataset; P_{Agent} requires 3.7 seconds; and P_{Doc} requires 3 seconds. On average, P_{Agent} saves about 65% of the time over P_{DB}; P_{Doc} saves about 72% of the time. However, P_{DB} consumes the least memory (one context-agent instance), whereas P_{Doc} consumes the most memory (four context-agent instances). Consequently, the different policies take different amounts of time to clear the pool.

Figure 6.5 shows the average time (in milliseconds) to retrieve excerpts for the marks in the tiny dataset. The first set of bars show the average time to retrieve an excerpt when the marks are submitted to the bulk accessor in a *shuffled* order such that two consecutive calls to the bulk accessor retrieve marks from different documents. (Marks are shuffled *before* they are submitted to the bulk accessor. Shuffling increases the number of context-agent switches in the policy P_{Agent}.)

Table 6.5: Time (in milliseconds) to retrieve excerpts for the tiny dataset

		Pooling Policy		
Document	**Sub-document**	P_{DB}	P_{Agent}	P_{Doc}
P1.pdf	1, 41-97	2,281	2,156	2,159
		The Acrobat application is loaded for the first time, and a context-agent instance is created, in each policy		
P2.pdf	1, 61-93	2,282	219	172
		Destroy current agent, create new agent instance, load Acrobat	Switch to 2nd document	Create new agent instance, open document
P1.pdf	3, 395-439	2,218	78	54
		Destroy current agent, create new agent instance, load Acrobat	Switch to 1st document	Instance 1, switch to new sub-document
P2.pdf	2, 17-31	2,203	78	54
		Destroy current agent, create new agent instance, load Acrobat	Switch to 2nd document	Instance 2, switch to new sub-document
E1.xls	S1, A11	329	266	219
E2.xls	S1, B5	312	546	296
E1.xls	S1, A6	297	188	31
E2.xls	S1, F5	391	172	31
Total time (ms)		10,312	3,704	3,016
Average time (ms)		1,289	463	377
Pool size		1	2	4
Time to clear pool (ms)		0	281	375

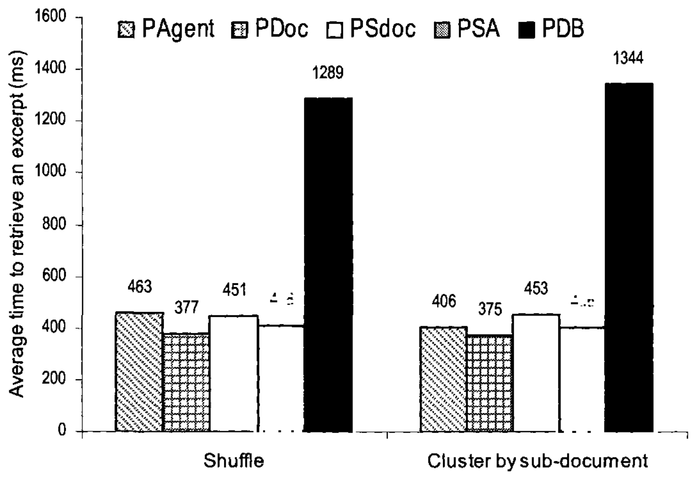


Figure 6.5: Average time (in milliseconds) to retrieve an excerpt for the tiny dataset. The first set of bars is for the case of shuffled marks; the second set is for marks clustered by sub-document

In Figure 6.5, the second set of bars shows the average time when the marks are submitted to the bulk accessor clustered by sub-document. With this clustering, P_{Agent} uses 12% less time per excerpt compared to the case when marks are shuffled. P_{Agent} even uses 10% less time compared to P_{Sdoc}, because with clustering, P_{Agent} consumes much less memory than P_{Sdoc}: P_{Agent} requires only two context-agent instances (one per base type), whereas P_{Sdoc} uses eight instances (one per sub-document).

The average time to retrieve an excerpt using the policy P_{Agent} is more than that needed with the policy P_{Doc} even with clustering, because the tiny dataset has only two documents and involves only two context-agent classes. (We will show that this behavior reverses for a larger number of documents.)

6.4.2.2. The Sidepad Dataset

The Sidepad dataset contains 2735 marks into 490 distinct sub-documents in 56 distinct documents. The document types are PDF, Excel, Word, and PowerPoint. The best performance for any policy was obtained when the marks were clustered by document. The total time to retrieve excerpts for all marks for the best case was: P_{Agent} 62.9 seconds; P_{Doc} 54.7 seconds; P_{Sdoc} 136.8 seconds; P_{SA} 568.9 seconds; and P_{DB} 1115.9 seconds.

Figure 6.6 shows the average time (in milliseconds) to retrieve an excerpt for the Sidepad dataset. The first set of bars shows the average time to retrieve an excerpt when the marks are shuffled; the second set of bars shows the average time when the marks are clustered by document; and the third set of bars shows the average time when the marks are clustered by sub-document.

The summary observations based on the average time to retrieve an excerpt shown in Figure 6.6 are:

- P_{Doc} provides the best performance for both shuffled and clustered marks.
- P_{Doc} saves 90% over P_{SA}, and 95% over P_{DB}.
- P_{Doc} always outperforms P_{Sdoc}.
- P_{Agent} saves 69%–90% over P_{SA} and 84%–94% over P_{DB}.
- Clustering helps P_{Agent} perform 38% better than when the marks are shuffled.
- Clustering also helps P_{Agent} perform almost as well as P_{Doc} because clustering reduces the number of context-agent switches for P_{Agent} from 2098 to just 55. (Figure 6.6 does not show the number of context-agent switches.)

The average execution times (in milliseconds) when marks are clustered by sub-document and the bulk accessor is invoked from within MSSQL are: P_{Agent} 25.18; P_{Doc} 24.73; P_{Sdoc} 54.82; P_{SA} 208.73; and P_{DB} 409.1. The ranking of pooling policies based on these times is the same as the ranking of the policies in the third set of bars in Figure 6.6. For example, P_{Doc} and P_{DB} have the best and worst average time, respectively, in both cases.

The pool size for the Sidepad dataset for the various policies was as follows: P_{Agent} 4; P_{Doc} 56; P_{Sdoc} 490; P_{SA} 2735; and P_{DB} 1. The pool size is the same with or without clustering.

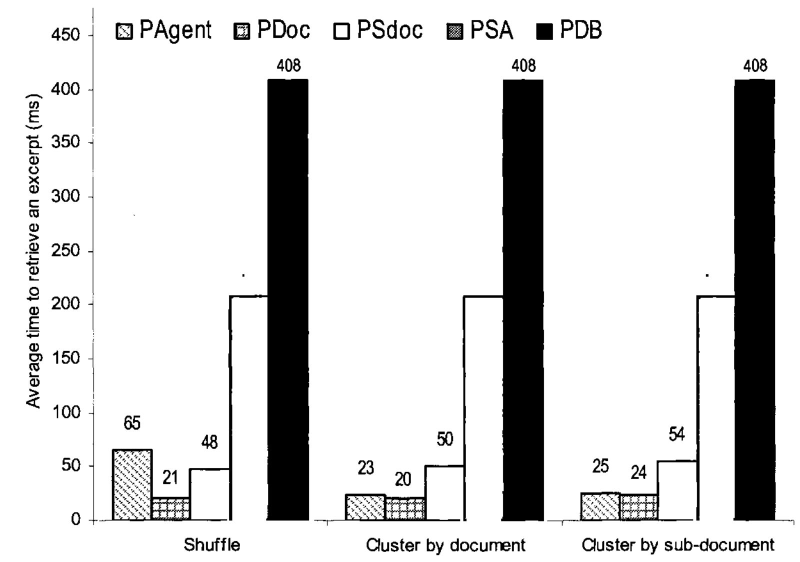


Figure 6.6: Average time (in milliseconds) to retrieve an excerpt for the Sidepad dataset. The three sets of bars are for marks shuffled, clustered by document, and clustered by sub-document, respectively

Based on Figure 6.6 and the pool sizes, for the Sidepad dataset, the query processor needs to choose only between the policies P_{Agent} and P_{Doc}. If the marks are clustered by document, the processor may prefer P_{Agent} as its performance is comparable to that of P_{Doc}, and its memory footprint is lower.

6.4.2.3. The SISRS Dataset

The SISRS dataset contains marks as might be created in a peer-review process. This dataset consists of 426 papers in PDF format obtained from the proceedings of a few of the past Computer Science conferences. Each document has exactly 12 pages, and each page has three marks, for a total of 15,336 marks. The marks were generated programmatically as follows: One mark was created in each third of a page. The location of the marked regions—the start of a region and its length—were determined using a random-number generator. The lengths of the marked regions range between 3 and 20 words.

For this dataset, we report only the performance of the policies P_{Agent} and P_{Doc} when the marks are shuffled and when the marks are clustered by document. The other pooling policies performed poorly. As with the Sidepad dataset, clustering by sub-documents did not provide much benefit over clustering by documents.

P_{Agent} produced the best total time (6.2 minutes) to retrieve all excerpts with marks clustered; P_{Doc} produced the best total time (7.9 minutes) when marks were shuffled. P_{SA} could process only 59% of the dataset due to its excessive memory needs. We did not measure the performance of P_{DB} for the entire dataset, but based on the performance for a part of the dataset, we estimate that it needs over *nine hours* to retrieve all excerpts.

Figure 6.7 shows the average time (in milliseconds) to retrieve an excerpt for the SISRS dataset. The first two bars show the average time when shuffled marks are processed using the policies P_{Agent} and P_{Doc}, respectively. The third and fourth bars show the average time when marks clustered by document are processed using these policies. We discuss the fifth bar after analyzing the first four bars.

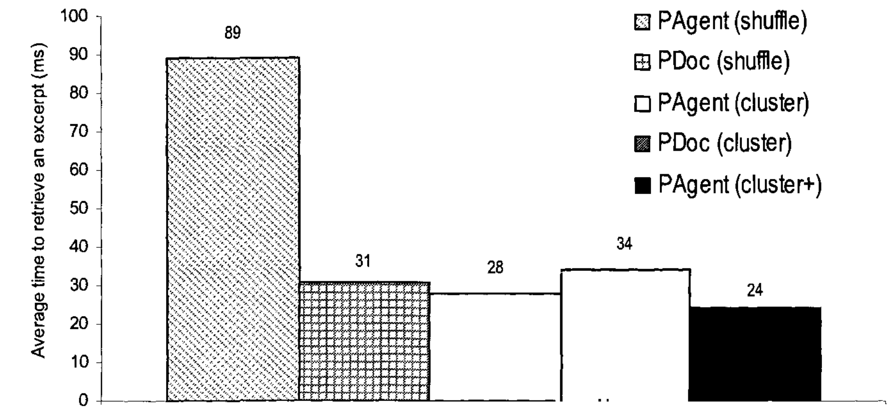


Figure 6.7: Average time (in milliseconds) to retrieve an excerpt for the SISRS dataset, with and without clustering

According to Figure 6.7, on average, the policy P_{Doc} performs better than P_{Agent} (65% savings) when marks are shuffled, but P_{Agent} performs better than P_{Doc} (18% savings) when marks are clustered. The better performance of P_{Agent} when marks are clustered is attributed to its memory efficiency: P_{Doc} employs 426 context-agent instances, whereas P_{Agent} employs only one context-agent instance. Each instance of the PDF context-agent references six objects in the Acrobat library, resulting in 2556 Acrobat objects for P_{Doc}, but only 6 Acrobat objects for P_{Agent}. Acrobat is unable to handle the volume of data P_{Doc} generates and triggers the *conservation procedure* of the bulk accessor. During this procedure, the context-agent instances for PDF marks release all references to

Acrobat objects, and, if more memory is needed, Acrobat is restarted (forcing Acrobat to release resources). Consequently, P_{Doc} consumes more time on average to retrieve an excerpt than P_{Agent} does.

The performance of the bulk accessor for the SISRS dataset when marks are shuffled deserves special attention: There are different degrees of shuffling of marks. *Simple shuffling* orders marks such that alternate calls to the bulk accessor retrieve excerpts from the same document. *Extreme shuffling* retrieves excerpts from the first mark of all documents, followed by the second mark of all documents, and so on.

The first two bars in Figure 6.7 show the average time for simple shuffling. Executing the complete workload (of 15,536 marks) using this policy under extreme shuffling triggered the conservation procedure too frequently, and the average response time for the policy P_{Agent} tended towards that expected for P_{DB}. We believe that potential issues in Acrobat may have exacerbated the situation because the performance did not degrade so drastically for similar workloads containing only Excel marks or only Word marks.

Figure 6.7 shows that the policy P_{Doc} performs slightly better when marks are shuffled than when marks are clustered. We attribute this difference to the order in which the marks were processed. Our logs show that the time to retrieve excerpts for some marks was much higher when marks were clustered than when the marks were shuffled. Examining the order in which the marks were processed, we found three clusters of marks into graphics-intensive pages that were responsible for much of the difference in the performances. Although in this case P_{Doc} performed slightly slower when marks were clustered than when marks were shuffled, we believe that there is nothing inherent in clustering marks by document that can hurt the performance of P_{Doc}.

The fifth bar, labeled "$P_{Agent\ (cluster+)}$", in Figure 6.7 shows the average time when marks are clustered by document for the policy P_{Agent}, but in this case, a document is closed *immediately* after all its marks are processed, before the marks in the next document are processed. This approach results in a savings of 14% compared to P_{Agent} when documents are not immediately closed, and a savings of 29% compared to P_{Doc}.

The time shown in the fifth bar *includes* the time to close base documents, whereas for the other bars in Figure 6.7, the times shown *exclude* the time to close base documents. That is, $P_{Agent\ (cluster+)}$ does more work than the other approaches, yet it consumes the least amount of time to retrieve all excerpts. If the time to close base documents is included, the average time to retrieve each excerpt increases to 28.13 and 34.21 respectively for $P_{Agent\ (cluster)}$ and $P_{Doc\ (cluster)}$.

Figure 6.8 shows the moving average of the time to retrieve excerpts for $P_{Doc\ (cluster)}$, $P_{Agent\ (cluster)}$, and $P_{Agent\ (cluster+)}$, computed for every 252 marks (that is, for every seven documents). The topmost line corresponds to $P_{Doc\ (cluster)}$. The moving average for this case has a rising trend until Document #251 (the *x* axis shows Document #), because with each new document encountered, a new context-agent instance is created, along with the creation of references to various Acrobat objects. When the 251st document is encountered, the bulk accessor's conservation procedure forces context-agent instances to release all open Acrobat objects, and restarts Acrobat. This process causes the spike seen in the average time to retrieve excerpts. (The time to complete the conservation procedure was over 5 seconds.)

The second and the third line in Figure 6.8 correspond to $P_{Agent\ (cluster)}$ and $P_{Agent\ (cluster+)}$, respectively. These two lines have similar shape, but the line for $P_{Agent\ (cluster+)}$ shows that closing a document when it is no longer needed saves time consistently (because closing a document increases available memory).

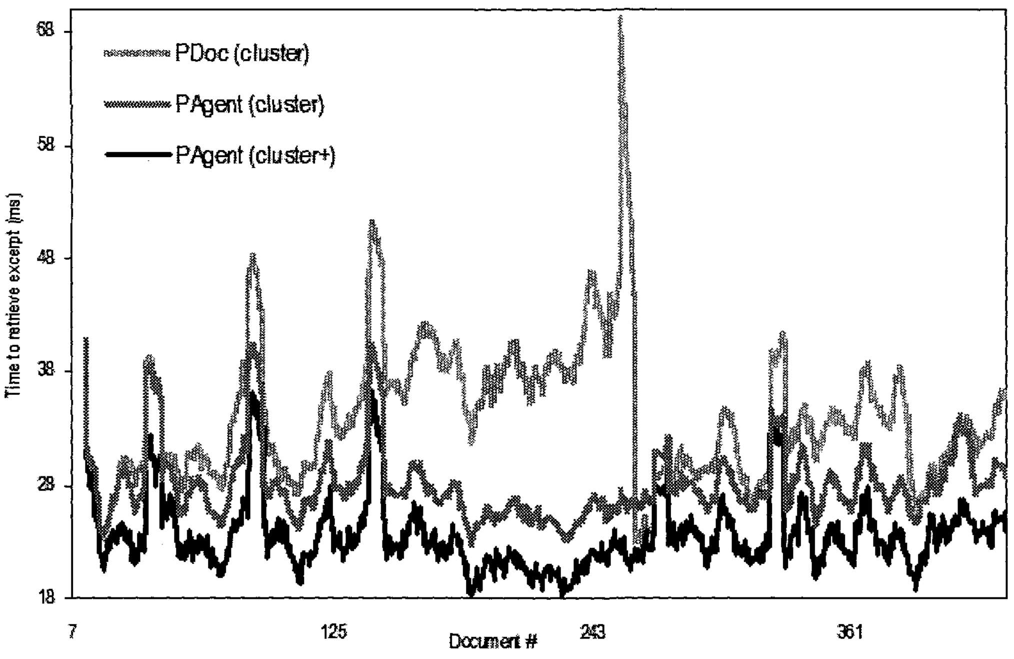


Figure 6.8: Moving average of time (in milliseconds) to retrieve excerpts for the SISRS dataset

6.4.2.4. The SSIB Dataset

The SSIB dataset contains marks to events, errors, and updates related to nine computers. (Section 4.2 describes the SSIB application.) It consists of 25 documents: 18 Excel spreadsheets containing event logs (two per computer), six Word documents containing errors reported (one per computer; not all computers had reported errors), and one XML document with details of available updates. Marks were created programmatically into each of these documents using the following criteria: one mark per event, three marks per reported error, and one mark per update applied on a computer. A total of 107,622 marks were created over 105,678 distinct sub-documents. The difference between the number of marks and sub-documents is due to the same update being applied on multiple computers.

For this dataset, we report the performance of the policies P_{Agent} and P_{Doc} when marks are clustered by document. As with the other datasets, clustering by sub-documents did not provide much benefit over clustering by documents, and P_{Doc} was the best choice when marks were shuffled.

The first set of bars in Figure 6.9 shows the average time (in milliseconds) to retrieve an excerpt for the SSIB dataset. (We use a non-zero baseline to highlight the difference in performance among the policies.) The policy P_{Agent} saves 10.4% of the time on average compared to P_{Doc}. The savings increase to 12.9% if a document is closed immediately after processing its marks (indicated by the bar labeled "$P_{Agent\ (cluster+)}$"). Marks were processed in the following order: all marks into event log, followed by all marks into error reports, followed by all marks to the updates catalog. The total time (in minutes) to retrieve excerpts for all marks was: P_{Agent} 9.14, P_{Doc} 10.29, and $P_{Agent\ (cluster+)}$ 8.97.

We also measured the performance of the policies P_{Agent} and P_{Doc} when the marks are clustered by base documents, but all marks for a computer are processed completely before marks for another computer are processed. For example, marks into the event logs for computer C1 are processed first, followed by marks into error reports for C1, followed by updates for C1. This pattern then repeats for computer C2, and so on.

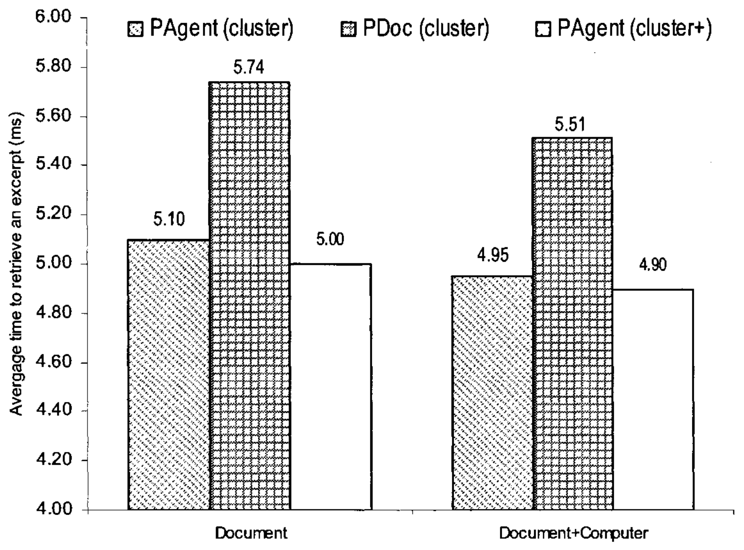


Figure 6.9: Average time (in milliseconds) to retrieve an excerpt for the SSIB dataset. The first set of bars shows the average time when marks are clustered by document. The second set shows the average time when marks are clustered by document and by computer

The second set of bars in Figure 6.9 shows the average time (in milliseconds) to retrieve an excerpt when marks are clustered by document and are grouped by computer. In this case too, P_{Agent} performs better than P_{Doc}, and closing a document soon after processing all its marks results in additional savings with P_{Agent}. That is, changing the order of marks did not change the ranking of the performance of the pooling policies (because the marks are still clustered by base documents).

6.4.2.5. Evaluation Summary

Our experiments provide the following insights into the use of the bulk accessor to retrieve excerpts:

- The bulk accessor performs better than the naïve methods even for a very small number of marks. It can save 69%–90% of execution time in comparison to the interactive SA approach, and 84%–95% when compared to the naïve DB approach, even when a query involves only a few thousand marks.
- When marks are clustered by documents, P_{Agent} provides the best response time and consumes the least memory.
- When marks are shuffled, P_{Doc} provides the best response time, but it consumes more memory than P_{Agent}. Limitations of some base applications can affect the response time of P_{Doc}.
- The bulk accessor supports five pooling policies, but, generally, a query processor needs to choose only between the policies P_{Agent} and P_{Doc}.

6.5. Related Work

In this section, we provide an overview of a system of optimizing access to data resident in files stored outside a traditional DB. We also briefly relate parts of the bulk accessor component to object management systems.

6.5.1. Structuring Schemas and Region Indexes

Consens and Milo [27] consider the problem of optimizing access to regions of file data using indexes over data described using structuring schemas. A *structuring schema* [1] is a grammar and a set of programs that describe the content of a file. Structuring schemas are used to present a structured view of data stored in files. The grammar component of a structuring schema defines the structure of the file, and the programs implement the grammar specification. For example, in the SSIB application (described in Section 4.2), the contents of an event-log file could be described using the following grammar. (Only a part of the grammar is shown.)

```
<Events> = <Event> <Events> | ε
<Event> = <EvDateTime> <Kind> <Source> <Description>
<EvDateTime> = <Date> <Time>
<Date> = <Month> <Day> <Year> | ε
<Time> = <Hour> <Minute> <Second> | ε
```

Some of the non-terminal symbols in this grammar can be exposed as DB elements. For example, a relation named `Events`, with rows of type `Event`, can be exposed. The row type `Event` can expose the attributes `EvDateTime`, `Kind`, `Source`, and `Description`. The non-terminal symbols `Date` and `Time` need not be exposed. Programs associated with this grammar can parse an event-log file and load the relation `Events`, or the programs can provide a view over the event-log file.

A structuring schema is not a mark, but a mapping from a file's content to a relation's content. When the mapping is applied, the file is scanned sequentially and its contents are exposed as a row set (assuming the relational model). If a query over the file's content involves a predicate over the attributes the file exposes, the predicate is pushed down to the program associated with the structuring schema, as an optimization.

The structuring-schema approach provides a means to mix DB data with external data, but it fully scans the external sources involved in a query. Consens and Milo [27] address this problem by maintaining an index over the structure of a file's content. The index may contain information about some or all components of a structuring schema.

An index entry indicates either a *match point* (which is the position of the indexed component) or a *region* (which is the span of the indexed component) in the file. Consens and Milo optimize access to indexed regions using region-inclusion graphs. A *region-inclusion graph* (RIG) is a directed graph with nodes representing indexed regions and edges denoting inclusion. An edge from region *r* to *s* means *r* includes *s*.

For example, consider the aforementioned grammar for a structuring schema over an event-log file. Assume that all the non-terminal symbols are exposed and indexed. Figure 6.10 shows a RIG for this structuring schema. This graph shows that the region containing the information about an event in turn contains the regions with the date and time, kind, source, and description of the event. Also, the region containing event date and time is broken into two regions: one for event date, another for event time.

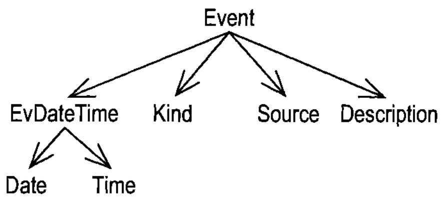


Figure 6.10: A region-inclusion graph for the event-log structuring schema

Now, consider the path expression `Event.EvDateTime.Date` to retrieve the date of an event. This expression can be evaluated by finding an event region that contains an event date and time region, which in turn contains a date region. Because the regions of all three attributes in question are indexed, the objects that satisfy the path expression can be found by evaluating the index expression *Event* $\supset$ *EvDateTime* $\supset$ *Date*, where the symbol $\supset$ denotes range inclusion. If the region containing the attribute

`EvDateTime` is the only container of the region containing the attribute `Date`, the index expression can be rewritten as *Event* $\supset$ *Date*. Evaluating the rewritten expression requires consulting the indexes for only two attributes, not three attributes.

Maintaining a region index provides two key benefits in this example. First, candidate event records that satisfy a query can be determined without consulting the full event log. Second, the event log does not need to be scanned sequentially (assuming the event-log file supports random access).

Region indexes can also reduce the number of file reads in some cases. For example, if a query needs the attributes `Kind` and `Source`, it would be possible to read the region encompassing both attributes at once (and separate the attributes in memory) instead of reading the two attributes separately.

Region analysis can be useful in bi-level query execution, but it cannot be performed completely by a query processor in our approach because mark descriptors are opaque (by design) to the processor. The query processor can coordinate the analysis, but context agents would need to provide the functionality to compare marked regions. For example, an agent could test if a marked region contains (or overlaps) another region.

An index over a file described using a structuring schema is a superimposed structure, but the data in a relation (or a view) obtained using a structuring schema contains only external data. That is, it does not allow the mixing of DB data and external data in the same schema instance. For example, in the event-log example for the relational model, the relation `Events` would contain only information from the event-log file. In contrast, as shown in Sections 4.4 and 4.5, our approach allows a mark to be mixed with SI in the same schema instance. This approach allows a developer to easily combine SI and base information.

A region index over a file's content indicates the exact portions of a file to read, and it can help reduce the number of file reads, but it does not address the issue of accessing a large number of regions in a file.

6.5.2. Object Management Systems

The pool of context-agent instances the bulk accessor uses is similar to *object pools* used by object management systems (OMSs) such as Enterprise JavaBeans [71] and BEA Tuxedo [90]. The methods initialize and clear (shown in Figure 6.2) correspond to the activation and passivation mechanisms, respectively, in an OMS. In an OMS, *activation* initializes an object before the object is used in providing a service; *passivation* saves the state of an object and deactivates the object. In contrast to a typical OMS, the bulk accessor does not save the state of a context-agent instance after deactivation because that functionality is generally not needed for bulk access to context information.

A typical OMS does not have an equivalent to the method conserve the bulk accessor uses to conserve memory without deactivating context-agent instances. Instead, an OMS deactivates objects.

6.6. Summary and Conclusions

In this chapter, we have isolated the problem of efficiently accessing context information for a large number of marks when executing a bi-level query. We have proposed a component called the *bulk accessor* as a solution. A key part of this solution is to pool context-agent instances so that the cost of accessing base sources is amortized over the entire set of marks involved in a query. We have identified several pooling policies, and provided heuristics to choose a policy based on certain data characteristics.

We have also described an implementation of the bulk accessor and showed experimentally that the accessor provides significant improvement over naïve methods for even a small number of marks. However, when a query involves thousands of marks, even with a bulk accessor, the query can take minutes to complete.

We see several opportunities to improve the performance of the bulk accessor. These opportunities lie in different realms: context-agent implementations and their interface with base applications; the query processor; and the interface between the query processor and the bulk accessor.

The performance of the UDFs `excerpt` and `context` depends on the context-agent implementation, base type, and base application. Using light-weight wrappers to retrieve context information instead of using full-blown applications can improve performance in some cases. For example, the open-source library PDFBox [134] can be used to retrieve context information from PDF marks instead of using Acrobat. In general, loading only the parts of a base application and document necessary to retrieve the requested information would provide better performance.

Indexing (or caching) often-used context information can improve the overall performance of a bi-level query, but doing so requires that the DBMS be able to detect changes in base documents. Interestingly, the bulk accessor itself can be useful in updating an index on base documents.

Another means of improving the overall performance of a bi-level query is to *eagerly* perform operations (for example, *push down* selections over SI) on SI stored in a traditional DB to possibly reduce the number of base accesses. Making such decisions requires that a query processor distinguish DB-resident SI from information in the base layer, but current query processors do not possess this capability. For example, current processors treat the UDF `excerpt` on par with other internal UDFs.

As illustrated in Section 6.4.2, shuffled marks can be a performance bottleneck. One way to process shuffled marks better is to submit marks in batches to the bulk accessor. The bulk accessor can internally cluster the marks by document, retrieve excerpts, and return a batch of results to the query processor. However, using this approach requires a significantly different interface (than the current one) between the query processor and the bulk accessor.

In this chapter, we have used the bulk accessor in a traditional relational query processor to execute bi-level queries. Chapters 7 and 9 show the use of the bulk accessor in the XML model.

7. Representing and Manipulating XML Bi-level Information

In this chapter, we describe *Sixml* and *Sixml DOM*, two parts of our strategy (outlined in Section 5.3.2) to transform XML bi-level information using queries in existing languages.

Section 4.5.2 introduced Sixml element types to associate marks with XML content that can be represented in the Entity-Relationship (ER) [25] model. In Section 7.3, we present element types to associate marks with XML content (such as a *CData section* [43]) that cannot be directly expressed in the ER model. (An ER attribute may be represented as a CData section in XML, but ER cannot distinuguish an attribute from a CData section.) The new element types, along with those introduced in Section 4.5.2, serve to meet our goal of SI-schema independence (Goal G1 in Section 5.3.1), and the goal of diversity and multiplicity of mark associations (Goal G2) for a bi-level query system.

In Section 7.4, we describe *Sixml DOM* [120], an extension to the XML Document Object Model (*DOM*) [34], to manipulate Sixml data at run time. (DOM provides a means of manipulating a tree-like view of an XML document.) This extension is needed because DOM and its application-programming interface (API) do not adequately meet the run-time needs of Sixml data management. For example, DOM cannot automatically assign a mark's excerpt to an attribute (as the attribute's value).

Sixml DOM allows an input Sixml document to be in the *normalized schema* (for example, the first document in Figure 5.3), but permits navigation over the document as if the document is in the *nested schema* (for example, the document in Figure 5.2). By retrieving mark descriptors and context information *just in time*, and by supporting navigation in the nested schema, Sixml DOM can help a bi-level query processor meet the goals of query-execution efficiency (Goal G3) and ease of query expression (Goal G5). Chapter 9 describes a query processor that uses Sixml DOM.

The XML representation schemes used in this chapter are based on the developments in Sections 4.5 through 4.7. As in those sections, the Sixml element types belong to the namespace "sixml" and are bound to the Uniform Resource Identifier (URI) [15] "http://schema.sixml.org". However, for simplicity, we refer to the Sixml element types and attributes without using a namespace.

The names of the Sixml element types to associate marks introduced in Section 4.5.2 have the prefix "Xml_". In the rest of this dissertation, we drop that prefix from type names (because we used that prefix in Chapter 4 only to distinguish XML element types from relationship patterns). Also, for simplicity, we give a mark-association element the same name as its type. For example, we give the name "EMark" to an instance of the element type EMark.

7.1. Introduction

In this section, we outline our motivation to define Sixml DOM as an extension of DOM to manipulate mark associations, descriptors, and context information. Section 7.4 describes Sixml DOM in detail.

A Sixml document can be manipulated using DOM (because a Sixml document is an XML document), but doing so can be challenging because DOM cannot distinguish mark associations from other information. We illustrate some of these challenges using the Sixml document shown in Figure 7.1. This document is based on the element Comment in the Sixml document shown in Figure 4.26. For simplicity, the name of each mark-association element is changed to match its element type. The attribute xsi:noNamespaceSchemaLocation associates a schema with Comment. The prefix xsi indicates the XML-Schema-instance namespace [171].

```
<?xml version="1.0" ?>
<Comment excerpt=""
            xsi:noNamespaceSchemaLocation="http://schema.sixml.org/examples/sisrs.xsd"
            xmlns:xsi= "http://www.w3.org/2001/XMLSchema-instance"
            xmlns:sixml="http://schema.sixml.org">
  <sixml:TMark sixml:type="sixml:TMark" sixml:markID="45">
    Contradicts prior work
  </sixml:TMark>
  <sixml:AMark sixml:type="sixml:AMark" sixml:markID="23" sixml:target="excerpt"
               sixml:valueSource="true"/>
  <sixml:EMark sixml:type="sixml:EMark" sixml:markID="23"/>
</Comment>
```

Figure 7.1: A Sixml document in the normalized schema showing marks associated with an element, an attribute, and text content. SI parts are in bold. This document is based on the document in Figure 4.26. For simplicity, only the element Comment is shown, and the name of each mark-association element is changed to match its element type. The namespace prefix xsi is used to associate a schema with Comment

Knowledge of schema: A developer needs to know mark association schemas to manipulate mark associations using DOM. For example, accessing the marks associated with the element Comment shown in Figure 7.1 would require the following code.

```
NodeList markAssociations = comment.getElementsByTagName("EMark");
```

Here, the variable `comment` holds a reference to Comment. DOM [35] defines the function `getElementsByTagName` and the type `NodeList`. To develop this code, the developer must know that mark associations for Comment are represented as elements named EMark, and that those elements are sub-elements of Comment.

In contrast, Sixml DOM allows access to mark associations without the knowledge of their schema. For example, the list of marks associated with the element Comment can be accessed using the simple expression `comment.markAssociations`.

Creating and serializing mark associations: Attaching a mark association to a part of an XML document, and *serializing* (that is, writing out) the association are both tedious with DOM. With Sixml DOM, a mark association is added directly and serialized *automatically* using only the syntax recommended [43] by the World Wide Web Consortium (W3C) [163] for serialization of XML data. For example, the Sixml document in Figure 7.1 is serialized according to the recommended syntax.

Accessing context information: Accessing mark descriptors and context information is tedious with DOM. For example, the following code would be needed to retrieve the context information from the first mark associated with the element Comment in Figure 7.1. The types `Element` and `string`, and the function `getAttribute` used in this code are defined by DOM. Our middleware to access marks and context information (described in Sections 3.2–3.4) defines the other types and functions used.

```
Element firstMA = comment.getElementsByTagName("EMark").item[0];
string markID = firstMA.getAttribute("sixml:markID");
ContextAwareObject cao = repository.GetCAOFromID(markID);
Context c = cao.ContextAgent.GetContext();
```

The first two lines of this code extract the mark ID from the first mark-association element. The code then uses the extracted mark ID to retrieve a mark object from a descriptor repository SPARCE maintains. Finally, the code uses the context agent associated with the retrieved mark object to get context information. However, context information thus retrieved cannot be manipulated using DOM because SPARCE returns context information in its own model; not in the XML model. See Section 3.3.

In comparison, accessing context information is much easier with Sixml DOM. For example, the expression `comment.markAssociations.item[0].Context` returns the context information for the first mark associated with Comment. Also, this information will be in the XML model, and it can be retrieved *just in time* from the base layer.

As with retrieving context information, it is tedious to use DOM to assign a value from the context of a mark to some part of a document. For example, the value of the attribute excerpt seen in Figure 7.1 should be, at run time, the text excerpt obtained from a mark. Using DOM, the developer would need to *explicitly* retrieve the text excerpt and assign it to the attribute. Sixml DOM automates this task. With Sixml DOM, the developer can *declaratively* specify (in the Sixml document) that the attribute's value should be a mark's text excerpt. Section 7.4.3.4 provides the details.

Navigating bi-level information: With Sixml DOM, a Sixml document can be navigated in the *nested schema* (described in Section 5.2.1), though the document is in the *normalized schema* (described in Section 5.2.2). For example, the Sixml document of Figure 7.1 does not include mark descriptors and context information, but those parts can be accessed as if they were included (using the mark-association properties `Decriptor` and `Context`, respectively). DOM cannot provide such access to mark descriptors and context information because it treats mark associations as traditional elements.

Enabling bi-level querying: Code expressed against Sixml DOM tends to be similar to query expressions we wish to support over bi-level information. For example, the mark associations for `Comment` can be accessed using the XPath [166] expression `/Comment/EMark`. Likewise, the context information for the first associated mark is accessed using the expression `/Comment/EMark[position()=0]/Context`. (Both these expressions require knowledge of the mark-association schema. Chapter 9 discusses querying mark associations without this knowledge.) A query processor can exploit this similarity to use Sixml DOM to help execute bi-level queries. Using Sixml DOM can make the query processor much simpler because Sixml DOM hides the details of retrieving mark descriptors and context information (and it retrieves them just in time), and exposes the retrieved information in the XML model.

In the rest of this chapter, we first provide an overview of DOM and then briefly revisit the issue of diversity and multiplicity of mark associations. We then present a detailed design of Sixml DOM. We also share the results of an experimental evaluation of Sixml DOM, review related work, and present some concluding remarks.

In this chapter, we refer mainly to the class diagram for Sixml DOM shown in Figure 7.2. (This diagram is drawn using the syntax for static class diagrams as defined in

UML, the Unified Modeling Language [159].) The classes shaded gray are defined in DOM. Only the DOM classes, methods, and relationships needed to describe Sixml DOM are shown.

A note on terminology: DOM is specified in the Interface Definition Language (*IDL*) [131], a language to describe an API independent of an implementation language. The classes in Figure 7.2 are actually defined using the *interface* construct of IDL, but, for simplicity, we represent and refer to the interfaces as classes. In practice, an IDL interface can be expressed using the constructs *class* and *interface* in languages such as Java [71] and C# [148].

7.2. Overview of DOM

DOM is defined in three numbered parts called *levels*. *Level 1* [35] is the most basic level. A DOM level consists of one or more modules. A *module* specifies a narrow set of functionality. The module *Level 1 Core* [35] defines the core functionality needed to create different parts of an XML document; *Level 2 Core* [36] adds support for namespaces; and *Level 3 Core* [37] adds support for type information (that is, for schema information). The module *Level 3 Load and Save* [38] defines the functionality to parse and serialize XML data, including the classes LSParser and LSSerializer in Figure 7.2. Level 1 Core defines the other shaded classes in this figure.

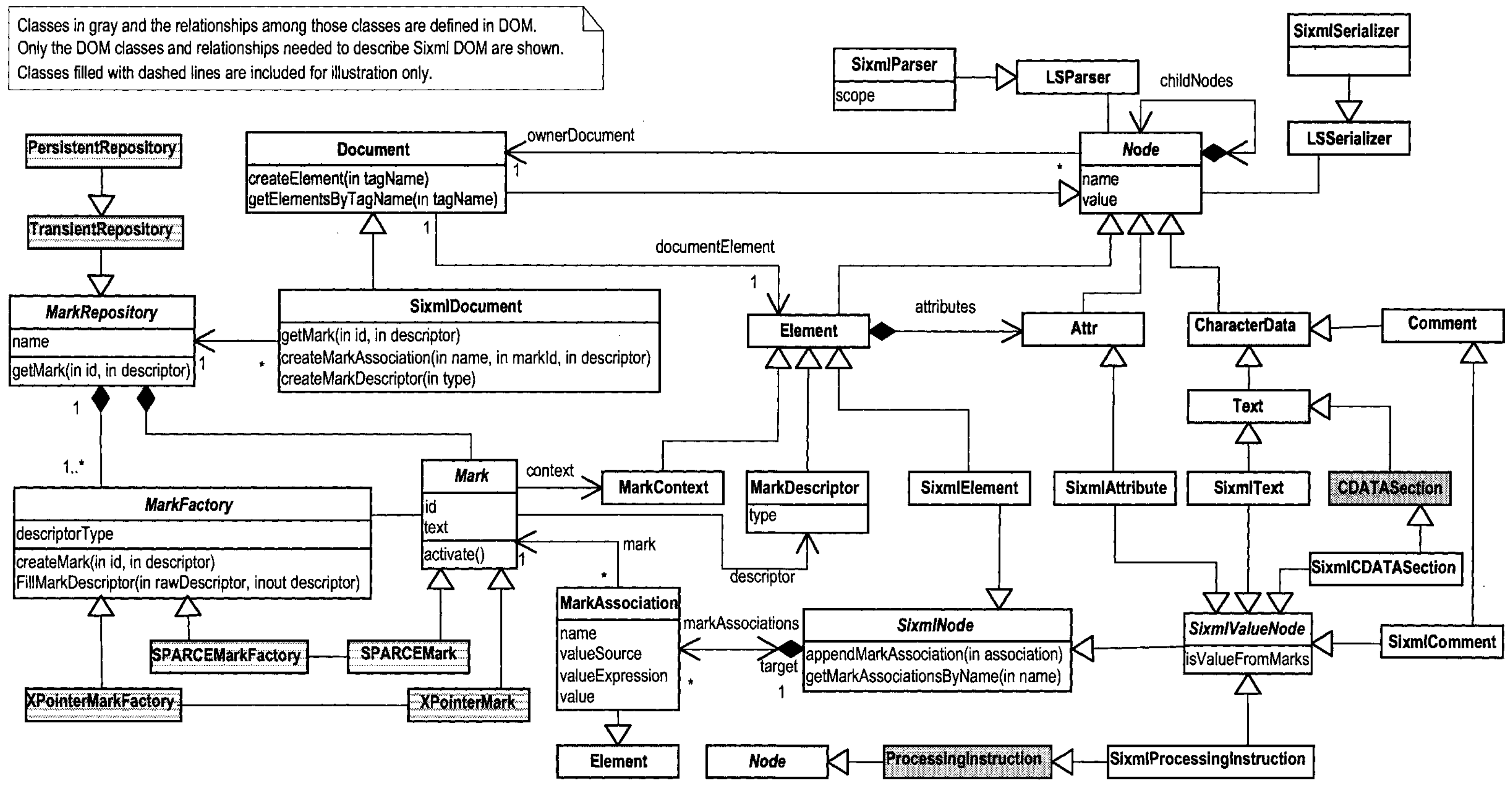

Classes in gray and the relationships among those classes are defined in DOM.
Only the DOM classes and relationships needed to describe Sixml DOM are shown.
Classes filled with dashed lines are included for illustration only.
PersistentRepository
TransientRepository
MarkRepository
name
getMark(in id, in descriptor)
Document
createElement(in tagName)
getElementsByTagName(in tagName)
SixmlDocument
getMark(in id, in descriptor)
createMarkAssociation(in name, in markId, in descriptor)
createMarkDescriptor(in type)
ownerDocument
documentElement
SixmlParser
scope
LSParser
childNodes
Node
name
value
SixmlSerializer
LSSerializer
Element
attributes
Attr
CharacterData
Comment
Text
MarkFactory
descriptorType
createMark(in id, in descriptor)
FillMarkDescriptor(in rawDescriptor, inout descriptor)
Mark
id
text
activate()
context
MarkContext
MarkDescriptor
type
SixmlElement
SixmlAttribute
SixmlText
CDATASection
SixmlCDATASection
mark
descriptor
MarkAssociation
name
valueSource
valueExpression
value
markAssociations
target
SixmlNode
appendMarkAssociation(in association)
getMarkAssociationsByName(in name)
SixmlValueNode
isValueFromMarks
SixmlComment
SPARCEMarkFactory
SPARCEMark
XPointerMarkFactory
XPointerMark
Element
Node
ProcessingInstruction
SixmlProcessingInstruction
1
*
1..*

Figure 7.2: A class diagram for Sixml DOM. The diagram is also available online [117]

Table 7.1: Types of DOM nodes

Node type	Has value?	Has parent?
Element	No	Yes. The document is the parent of *document element*; otherwise another element is the parent
Attribute	Yes	Yes, the parent is always an element, but an attribute is not a child of its parent
Text	Yes	Yes
CDATA section	Yes	Yes
Comment	Yes	Yes
Processing instruction	Yes	Yes
Document type	No	Yes, the parent is always the document
Notation	No	No
Entity reference	No	Yes
Entity	No	No
Document	No	No
Document fragment	No	No

DOM represents an XML document as an ordered tree of nodes. (The order is called *document order*, which is the order in which the nodes are serialized.) It defines 12 types of nodes (listed in Table 7.1). Features common to most types of nodes are included in the class Node. This class is specialized for each type of node. Figure 7.2 shows the specialized classes for six node types: element, attribute, text, CData section, comment, and processing instruction (PI).

Some DOM nodes (for example, an attribute) may have a value. The column "Has Value?" in Table 7.1 indicates which node types may have a value. An attribute uses an additional text node to represent its value, but the other node types maintain a value without using additional nodes.

Some nodes (for example, an element) may own other nodes, but other nodes (for example, text) may not. Also, some node types cannot be owned by other nodes. (See the column "Has Parent?" in Table 7.1.)

An element may have attributes, but it does not own its attribute nodes. An attribute has a "parent" element, but it is not a child of any element. That is, the collection induced by the relationship childNodes (in Figure 7.2) does not include attributes. (This relationship between an element and its attributes is contrary to the common expectation that the parent and child relationships are inverses.)

An XML document is represented by a node called the *document node.* A document node is an instance of the class Document. A DOM node is created in the context of a document using special methods called *factory methods* (one method per node type) defined in Document. For example, the method createElement creates an element node.

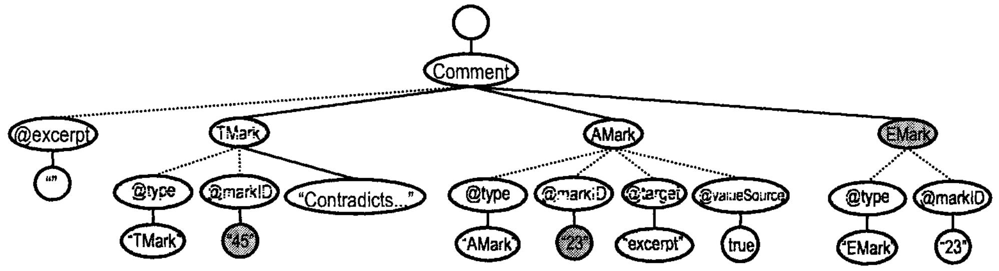


Figure 7.3: A simplified DOM tree for a Sixml document. The tree corresponds to the document in Figure 7.1. The symbol @ denotes an attribute, quotes denote a text node, and the unlabeled node is the document node. Namespace information is omitted for simplicity. A solid line denotes a parent-child relationship. A dotted line connects an attribute to its element.

The classes LSParser and LSSerializer are used to read and write, respectively, a node from or to an external source such as a disk file. These classes can also read and write a document because DOM represents a document as a node. Reading a document builds a tree. Figure 7.3 shows a simplified DOM tree built from document in Figure 7.1. The unlabeled root node is the document node. The node labeled Comment is the *document element* (that is, the top-level element). TMark, AMark, and EMark represent mark-association elements. The value of the attribute excerpt should be the text excerpt from the associated mark, but it is not, because DOM is unaware of the semantics of mark associations.

7.3. Diversity and Multiplicity of Mark Associations

Section 4.5.2 illustrated how marks may be associated with XML content that can be represented in the ER model. However, an SA developer might wish to associate marks with content that cannot be represented in that model, or he might model SI directly as XML (without first using a conceptual methodology).

In this section, we discuss associating marks with different DOM node types, independent of the ER model. We highlight two key considerations in associating marks, and introduce new element types (in addition to the types discussed in Section 4.5.2) to represent mark associations. Section 7.4 discusses how a document containing instances of these element types is manipulated at run time using Sixml DOM.

The developments in this section help meet our goal of diversity and multiplicity of mark associations (Goal G2 in Section 5.3.1), with respect to the XML model.

7.3.1. DOM Node Types and Mark Associations

DOM can be extended such that marks can be associated, at run time, with any of the 12 DOM node types, but serialization and validation considerations limit the node types with which marks may be associated. We now examine these considerations and determine the DOM node types with which marks may be associated.

7.3.1.1. Serialization and Validation Considerations

DOM is designed to interoperate with the syntax [43] W3C recommends for XML serialization. That is, a DOM implementation can read and write a document serialized according to this syntax. With Sixml DOM, we wish to maintain this interoperability with the W3C serialization syntax. Also, we would like a serialized Sixml document to contain markup that is uniform and comprehensible, and be amenable to validation using standard schema constructs. The serialized Sixml document shown in Figure 7.1 satisfies these criteria.

Encoding a mark association is the main problem in serializing a Sixml document. One solution is to develop conventions (for example, use comments with specific structure and contents) to encode mark associations, but conventions cannot be validated using standard schema constructs.

We choose to serialize a mark association as an *element* for the following reasons:

- The element construct is defined in both DOM and the serialization syntax.
- In both DOM and serialized forms, an element can contain most kinds of XML content, including another element.
- When serialized, an element allows the markup for mark associations to be placed in close proximity to the data that is associated with marks, thus improving comprehension.

- An element may be associated with a type via an XML Schema [170] instance document or a document type definition (DTD) [43] and hence validated. For example, in Figure 7.1, the attribute xsi:noNamespaceSchemaLocation [171] associates a schema with the element Comment. (In XML Schema and in DTD, elements and attributes are the only kind of XML content that may be typed.)

Serializing a mark association as an element requires that a mark be associated with a DOM node *only if the node can contain an element, or if an element can contain the node*. (In this limited context, we treat an attribute node as being contained by an element node.) We call this requirement the *element-containment requirement*.

A serialized mark association must also meet the requirement that a serialized XML document be well-formed. A *well-formed document* [43] begins with the *XML declaration*, followed by (but not necessarily immediately) exactly one *document element* (which is the element that contains all other elements in the document). For example, the first line in Figure 7.1 is the XML declaration. Comment is the document element. A document that is not well-formed is an *ill-formed document*.

7.3.1.2. DOM Node Types Permitted for Mark Association

We allow marks to be associated with the following *six types* of DOM nodes: element, attribute, text, CData section, comment, and PI. However, we *disallow* mark associations for a comment, or a PI, if it is not contained by an element, because serializing such nodes results in an ill-formed document.

A note about comment nodes: An XML comment is quite different from its programming-language counterpart. A comment in a program typically has no run-time representation, but an XML comment does. Also, XML comments may be selected and constructed using queries. We see several situations where an XML comment can benefit from mark associations. For example, a comment in the XML version of an API's documentation might reference the API's source, and possibly even obtain comment text from the source. (Both C# [23] and Java [72] promote API documentation in XML format.)

We allow any number of marks with nodes of the aforementioned six types. A developer may use a schema to constrain the number of mark associations for a particular node.

We disallow marks to be added to entities, documents, and document fragments, because nodes of these types are just containers for other nodes. (That is, serializing a node of any of these types simply serializes its contents.)

We disallow mark associations for an entity reference because it cannot satisfy the element-containment requirement. We also disallow marks with document type and notation nodes because their serialization would cause the document to be ill-formed.

7.3.2. Mark-Association Element Types

We now provide an overview of the element types to associate marks with the six DOM node types with which marks may be associated. Appendix A shows the

XML-Schema instance document containing the complete definition of the element types. That instance document is also available online [119].

Section 4.5 introduced the element types EMark, AMark, and TMark to associate marks with elements, attributes, and text content, respectively. To recap, an EMark element is added as a sub-element of the *target element* (that is, the element with which the mark is associated). An AMark element is included as a sub-element of the element that owns the *target attribute*. A TMark is made a sub-element of the element that owns the *target text content*, and the target text content is *wrapped* inside the TMark. Figure 7.1 illustrates the use of these three element types.

We refer to the element types EMark, AMark, and TMark as *uni-mark types* because an instance of any of these types associates only one mark with its target. In contrast, a *multi-mark type* associates multiple marks with a node. We now introduce some new uni-mark and multi-mark types.

The uni-mark types CDataMark, CMark and PIMark respectively help associate a mark with CData section, comment, and PI. As with TMark, an instance of any of these types wraps its target.

The multi-mark element types TMarks, CDataMarks, CMarks, and PIMarks respectively associate multiple marks with text, CData section, comment, and PI. An instance of any of these types also wraps its target, and it contains one uni-mark instance for each mark associated with the target. In this case, a contained uni-mark element does not wrap its target because the outer multi-mark element would have already done so. The following XML segment shows two marks associated with the text content shown in Figure 7.1.

```
<sixml:TMarks sixml:type="sixml:TMarks">
 Contradicts prior work
  <sixml:TMark sixml:type="sixml:TMark" sixml:markID="45"/>
  <sixml:TMark sixml:type="sixml:TMark" sixml:markID="78"/>
</sixml:TMarks>
```

No multi-mark element types are needed to associate multiple marks with elements and attributes. Instead, multiple marks are associated with an element (attribute) simply by using one EMark (AMark) element for each mark to be associated. (A multi-mark type is needed for content other than elements and attributes, so that content is not repeated. For example, using TMarks to associate many marks with the same text content avoids repeating the text for each associated mark.)

7.4. Design of Sixml DOM

In this section, we describe the design of Sixml DOM. We discuss how mark associations are associated with DOM nodes at run time, how a serialized Sixml document is read for manipulation, and how a Sixml document is serialized when writing. Figure 7.2 shows a UML class diagram for Sixml DOM. Appendix B shows the complete Sixml DOM interface definition.

7.4.1. Overview

We first introduce the classes, methods, and properties Sixml DOM defines to support mark associations.

7.4.1.1. Sixml Nodes

In Sixml DOM, a node with which marks may be associated is called a *Sixml node*, and is represented by the class SixmlNode. A Sixml node that can contain a value is a *Sixml value node*, and is represented by SixmlValueNode, an extension of SixmlNode. See the column "Has Value?" in Table 7.1 for a list of node types that may contain a value.

Although we allow marks to be associated with six types of DOM nodes, for simplicity, we limit this discussion to elements, attributes, and text nodes. The classes SixmlElement, SixmlAttribute, and SixmlText represent these types of nodes, respectively. These classes respectively extend the DOM classes Element, Attr, and Text. In addition, the class SixmlElement extends the class SixmlNode (because an element cannot have a value). The classes SixmlAttribute and SixmlText extend the class SixmlValueNode because nodes of these types may have a value.

The class SixmlDocument extends the DOM class Document. It overrides the DOM factory methods in order to create Sixml nodes instead of creating DOM nodes. For example, it overrides the method createElement to create an instance of the class SixmlElement instead of an instance of the DOM class Element. SixmlDocument does not override the factory methods for the types of nodes with which marks cannot be associated. Consequently, a Sixml document can contain a mixture of regular DOM nodes and Sixml nodes.

7.4.1.2. Mark-Association Nodes

A mark-association node pairs a Sixml node, called the *target node*, with a mark and assigns a name to the pairing. A Sixml node may be associated with different marks using the same name, but a name may be used only once for a node-and-mark pairing. A node may be associated with any number of marks, unless the node's schema (if any) limits the maximum number of marks that may be associated with the node.

A mark-association node has no children. It is attached to a target node, but it is not a child of its target. (This relationship between a mark association and its target is similar to the relationship between an attribute and its owner element.) Marks may not be associated with a mark-association node.

The class MarkAssociation together with its relationships with SixmlNode and Mark represents a mark association. MarkAssociation extends the DOM class Element because we represent a mark association as an element.

A mark-association node is created using the factory method createMarkAssociation in the class SixmlDocument. The mark-association node thus created is added to a target node using the method appendMarkAssociation defined in SixmlNode. Methods to add a mark association at a particular location in the list of mark associations, to replace a mark association, and to delete a mark association are also defined.

The mark-association nodes added to a Sixml node may be accessed via the collection induced by the relationship markAssociations. Mark associations with a specific name may be retrieved using the method getMarkAssociationsByName.

7.4.2. Reading a Sixml document

We now describe how a Sixml DOM tree is created at run time from a serialized Sixml document. In this description, we use the term *mark-association element* to denote an element that represents a mark association in the serialized form. We use the term *mark-association node* to denote a Sixml DOM node that is created from a mark-association element.

7.4.2.1. Creating a Sixml DOM Tree

Conceptually, a Sixml DOM tree for a Sixml document is created in three steps. First, the document is represented as a tree in DOM. This step represents mark associations as DOM elements. Second, a mark-association node is created from each mark-association element and is attached to the appropriate target node. Finally, the nodes for the source mark-association elements are deleted from the tree.

The flow chart in Figure 7.4 outlines the procedure to create a mark-association node from a mark-association element of uni-mark type. Figure 7.5 shows the Sixml DOM tree generated from the DOM tree in Figure 7.3. A dashed edge connects a mark-association node with its target node (to clarify that a mark-association node is not a child of its target node). Following, the procedure in Figure 7.4, the element EMark is replaced by a mark-association node attached to the element Comment. The mark-association node generated from AMark is attached to the attribute excerpt. The text node that was wrapped inside TMark is now a child of Comment and the mark-association node generated from TMark is attached to the text node.

The partial value shown in Figure 7.5 for the attribute excerpt is the text excerpt obtained from the associated mark. Section 7.4.3.4 describes how the mark's excerpt is assigned to the attribute. (Figure 1.4 shows the base region corresponding to the mark associated with the attribute. Figure 5.2 shows the descriptor and context information for the mark.)

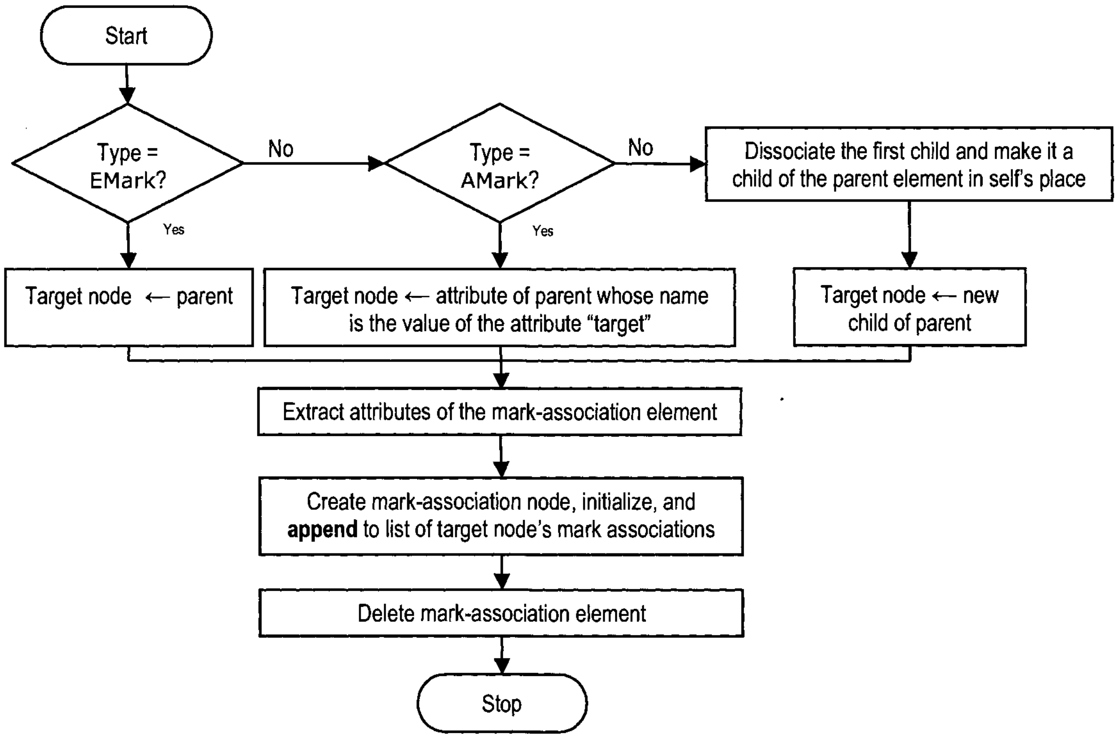

Start
Type = EMark?
No
Type = AMark?
No
Dissociate the first child and make it a child of the parent element in self's place
Yes
Yes
Target node ← parent
Target node ← attribute of parent whose name is the value of the attribute "target"
Target node ← new child of parent
Extract attributes of the mark-association element
Create mark-association node, initialize, and **append** to list of target node's mark associations
Delete mark-association element
Stop

Figure 7.4: Procedure to create a mark-association node from a uni-mark type of mark-association element

We now briefly discuss transforming a mark-association element of multi-mark type. Such a mark-association element (for example, TMarks) contains the target node (for example, text content) and a sequence of uni-mark elements (for example, TMark). This case is handled in the same mannner a uni-mark type that wraps its target node is handled: The target node is first made a child of the parent element, a mark-association node is created from each contained uni-mark element, and the generated mark-association nodes are attached to the target node. Finally, the multi-mark element is deleted from the tree.

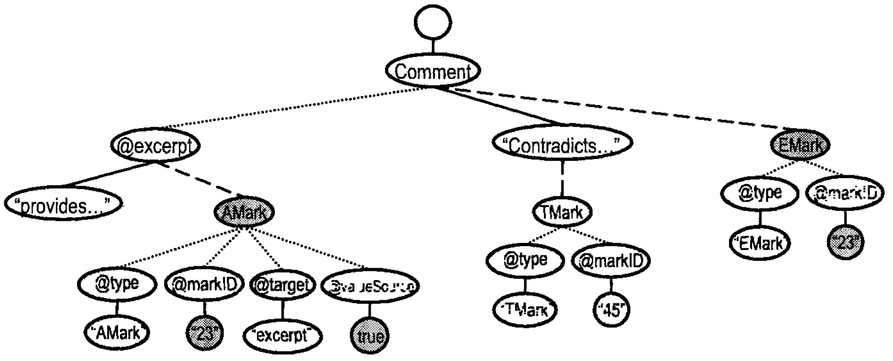


Figure 7.5: A simplified Sixml DOM tree for a Sixml document. The tree corresponds to the Sixml document in Figure 7.1. A dashed line connects a mark-association node with its target node. Other conventions used and the simplifications made are the same as in Figure 7.3

7.4.2.2. Detecting Mark-Association Elements

As seen in Section 7.4.2.1, determining which elements represent mark associations and determining the type of mark association an element represents are key parts of the procedure to transform mark-association elements to mark-association nodes.

If a schema is associated with the input XML document, the type of an element can be examined to determine if the element is a mark-association element and to determine the type of node with which it associates marks. For example, an element associated with the type `AMark` from the namespace whose URI is `http://schema.sixml.org` is an element that associates a mark with an attribute.

There are two impediments to relying on type information to detect mark-association elements and determine their types. First, many XML documents (especially those produced by ad-hoc queries) are not associated with a schema. Second, type information is supported only in DOM Level 3, but the DOM implementation an SA developer (or a user) chooses might not conform to DOM Level 3.

We propose the following *rules* to determine if an element represents a mark association. The prefix sixml is associated with the URI http://schema.sixml.org:

1. If the DOM implementation conforms to Level 3 and the element has type information, the element's type determines whether the element represents a mark association.
2. If no schema is associated with the element, or if the DOM implementation conforms only to Level 2, the element represents a mark association if its qualified name is the same as the qualified name of a mark-association type. For example, an element with the name "sixml:AMark" associates a mark with an attribute.
3. If neither Rule 1 nor 2 holds, and the DOM implementation conforms only to Level 2, the element represents a mark association if the value of its attribute sixml:type is the same as the qualified name of a mark-association type.
4. If Rules 1–3 do not hold, or if the DOM implementation conforms only to Level 1, the element's name (that is, the unqualified name) indicates the type of mark association. For example, an element with the name "AMark" associates a mark with an attribute.

5. If Rules 1–4 do not hold, or if the element does not conform to the mark-association type inferred according to Rules 2–4, the element does not represent a mark association.

We recommend associating a schema with each mark-association element. We also recommend the use of the attribute sixml:type for mark associations with custom names (even if the serialization is produced by a DOM Level 3 implementation and a schema is associated with the mark association) so that mark associations can be interpreted correctly by an implementation that conforms only to DOM Level 2. Following either of these recommendations allows the use of mark-association elements with custom names.

The Sixml document in Figure 7.1 complies with our recommendations: It references a schema, and it includes the attribute sixml:type with each mark-association element. Strictly speaking, even without the schema, the attribute sixml:type is not needed in this document, because the mark-association elements do not use custom names. (The mark associations would be interpreted correctly according to Rule 2.)

The Sixml document in Figure 4.26 does not fully comply with our recommendations though it references a schema, because it does not use the attribute sixml:type. A DOM implementation conforming to Level 3 would correctly interpret the mark associations in this document (according to Rule 1). However, the custom names would prevent an implementation conforming only to Level 2 from correctly interpreting the mark associations. (A Level 2 implementation would incorrectly recognize the mark-association elements as regular XML elements.)

7.4.3. Managing and Using Marks

We now provide an overview of managing marks, and accessing mark descriptors and context information.

7.4.3.1. Mark Repositories

In Section 7.4.1.2, we mentioned the use of the factory method createMarkAssociation in the class SixmlDocument to create a mark association. This method consults a *mark repository*, which is a collection of marks, to create marks. A mark repository corresponds to the notion of a *descriptors repository* introduced in Section 3.2.2.

The class MarkRepository represents a mark repository. The method getMark of this class accepts a mark ID and a descriptor, and returns a matching mark from the repository, creating a new mark if necessary. At least one of the two inputs must be provided.

The class MarkRepository is *abstract*. That is, this class is not directly instantiated. Implementations of this class may differ in their strategies to store, look up, and create marks, but the method getMark in any implementation should exhibit the following behavior:

- If a descriptor, but no ID is provided, the method should return a mark with a matching descriptor, creating a new mark if no existing mark matches the descriptor.

- If only an ID is provided, the method should return the mark with the specified ID. The method should cause an error if no matching mark is found.
- Whenever a descriptor is provided, the descriptor of the mark returned must match the input descriptor. If an ID is also provided, an implementation may choose to return a mark with a different ID. (Multiple marks in a repository might have the same descriptor.)

Figure 7.2 includes two example implementations of the class MarkRepository. The class TransientRepository implements a main-memory-based collection of marks. The marks in this repository last only as long as the instance of the repository does. Also, the descriptor for each mark must be present in the input document.

The class PersistentRepository models a repository that stores IDs and descriptors of marks in a persistent storage such as a disk file or a database. A persistent repository backed by an efficient look up facility for marks (for example, with the help of a database management system) can be useful when working with a large number of marks.

An instance of the class SixmlDocument is generally associated with one mark repository, but the instance might perform the repository tasks on its own, instead of employing a concrete implementation of MarkRepository.

7.4.3.2. Marks

The abstract class Mark models a reference to base information. A concrete implementation of this class must exist for each kind of mark descriptor. (Section 4.6.3 discusses descriptor kinds.) Because the exact instantiation requirements of a mark implementation cannot be known at design time, marks are created using a *mark factory* class specific to a descriptor kind.

The abstract class MarkFactory models a mark factory. A concrete implementation of this class must exist for each descriptor kind. The property descriptorType returns the kind of descriptor from which a factory can create a mark.

The mark repository in use by a Sixml document may be associated with one or more mark factories. The repository uses a mark factory to create a mark from a mark descriptor. It chooses a mark factory by matching the attribute xsi:type (described in Section 4.6.3) of the mark descriptor with the property descriptorType of each mark factory.

In Figure 7.2, the class SPARCEMark denotes a mark created from a SPARCE descriptor. SPARCEMarkFactory is the corresponding mark factory. Similarly, XPointerMark and XPointerMarkFactory support XPointer pointers [168]. These four classes are not part of Sixml DOM, but they are shown for illustration.

7.4.3.3. Mark Descriptors and Context

The class MarkDescriptor represents a mark descriptor. A mark descriptor is either included in the input document (as the element Descriptor), or it is obtained from a repository. In either case, navigating the relationship descriptor provides access to a mark's descriptor.

A mark descriptor does not have a parent even though the descriptor might have been included in the input document. This constraint allows the same descriptor element to be reused when the same mark is used more than once in a document.

MarkContext represents the top-level element (that is, the element `sixml:Context` in Figure 5.2) in the context information retrieved from a mark's context. This top-level element also does not have a parent, so that it can be reused with different mark associations that use the same mark.

The context information corresponding to a mark is retrieved using the context-agent implementation that the mark's descriptor indicates. As described in Section 3.3, a context agent represents context information as a hierarchical property set. MarkContext transforms the hierarchical property set a context agent returns to the XML model so that context information can be navigated using the DOM API.

The property text in the class Mark provides access to the text excerpt retrieved from the context of a mark. The method activate displays the referenced base region in its original context, as described in Section 3.3.3.

Our design allows an implementation to retrieve a mark's descriptor (from a mark repository) and context information (from the base layer) *on demand* (that is, only if the user navigates to these parts of a mark). The design also allows the implementation to *cache* context information so that the context information for the same mark is not repeatedly retrieved from the base layer. Our Sixml DOM implementations described in Section 7.6.1 implement both these features.

A bi-level query processor that uses Sixml DOM to (internally) represent a Sixml document can benefit from on-demand information retrieval and context caching. Chapter 9 illustrates such a query processor.

7.4.3.4. Deriving a Node's Value from Mark Context

A Sixml node may return a value derived from its associated marks, instead of returning an explicitly-stored value, as in DOM. The derived value of a node is the *concatenation* of the string values obtained from each of its *contributing marks*. Not every mark associated with the node is required to contribute to the node's value.

The class MarkAssociation defines the properties valueSource, value, and valueExpression to determine the value that a mark contributes to the target node's value (that is, supplies a part of the node's value): The property valueSource determines whether the mark contributes a value. The property value returns the contributed value if the property valueSource is `true`; otherwise value returns an empty string.

The property valueExpression determines the value a mark contributes. If this property is empty, the text excerpt retrieved from the mark (using the property text) is contributed. If this property is not empty, it should be an XPath expression that selects the context elements that contribute the value. The expression is executed with the top-level element (that is, the element sixml:Context) as the current node. For example, the expression `Container/Section/Heading` over the context information included in Figure 5.2 contributes the heading of the section that contains the marked region. (Section 4.8 also illustrates retrieving context information using path expressions.)

The properties valueSource and valueExpression of a mark-association node correspond to attributes of the same name in a mark-association element. Section 4.5.2 describes these attributes.

The class SixmlValueNode defines the property isValueFromMarks to denote whether a node's value is derived from its associated marks. This property is `true` only if the property valueSource is `true` for any of the mark associations added to the node. Setting the property isValueFromMarks of a value node to `false` (`true`) automatically sets the property valueSource of each of the node's mark associations to `false` (`true`).

The class SixmlValueNode overrides the property value defined in the base DOM class Node to account for the property isValueFromMarks. If isValueFromMarks is `false`, the data explicitly stored in the node is returned. Otherwise, a concatenation of the string values obtained from each contributing mark is returned.

7.4.4. Serializing a Sixml Document

We now discuss how mark associations in a Sixml document are serialized. (Mark descriptors and context information can also be serialized, but we omit those aspects. Section 7.3.1.1 discussed the need for serialization and the serialization considerations.)

A Sixml document is serialized using the class SixmlSerializer, because the DOM serializer (LSSerializer) would serialize only the SI portion of the document. (In Sixml DOM, a mark-association node is not a child of its target node.)

Figure 7.6 shows a pseudo-code procedure to serialize a Sixml element and its child nodes and mark associations. Comments placed at the end of selected lines in the procedure show how the different parts of the document element Comment in the tree of Figure 7.5 are serialized.

```
procedure WriteElement(SixmlElement e)
write start of element; //<Comment
write attributes and namespaces; //excerpt="" ...>

//write out child nodes and their mark associations
for each child node
  if (child node is a SixmlElement)
     WriteElement(child node); //None in the example
  else if (child node does not have mark associations)
     write child node as in DOM;
  else if (child node has more than one mark association)
     //No multiple mark associations in the current example
     write start of multi-mark-association element (for example TMarks);
     write child node as in DOM;
     for each mark associated with child node
       write mark-association element (for example TMark);
     write end of multi-mark-association element;
  else
     write start of mark-association element; //<sixml:TMark
     write attributes and namespaces; //sixml:type="sixml:TMark" sixml:markID="45">
     write the child node as in DOM; //Contradicts...
     write end of mark-association element; //</sixml:TMark>

//write mark associations for attributes
for each mark associated with each attribute
  write mark-association element; //<sixml:AMark sixml:type="sixml:AMark" .../>

//write mark associations for self
for each mark associated with self
  write mark-association element; //<sixml:EMark sixml:type="sixml:EMark" ... />
write end of element; //</Comment>
```

Figure 7.6: Pseudo code to serialize a Sixml element, its contents, and mark associations. End-of-line comments show how the document element Comment in Figure 7.5 is serialized

The serialization procedure writes the mark associations for a node in tree order. This order is important for a node that derives its value from marks because the value of that node is a concatenation of the string values obtained from each contributing mark (and string concatenation is not commutative). Also, the procedure first serializes child nodes (including their mark associations) of the input element, followed by the mark associations of the attributes of the element. Finally, it serializes the mark associations for the input element. The ordering of the mark associations for attributes and the containing element is not necessary, but it provides determinism.

7.5. Integration with DOM

We now briefly discuss the integration of Sixml DOM interfaces with DOM interfaces.

Sixml DOM can be integrated with DOM by *extending* the DOM interfaces through inheritance, or by *revising* the DOM interfaces to include Sixml functionality from the ground up. In this section, we present four alternative means of integration: three using the extension strategy, one using the revision strategy.

Alternative 1 is to introduce a new level, *Level 4*, to DOM. The new level would contain two modules. The module *Level 4 Core* would extend Level 3 Core, and the module *Level 4 Load and Save* would extend the module of the same name in Level 3. This approach provides a clean separation between DOM and Sixml DOM, but it requires an implementation to comply with Level 3 functionality, even though its developer might wish to support only un-typed mark associations. (In general, conformance to a DOM Level *n* requires conformance to Level *n*–1.)

Alternative 2 is to add new "Sixml" modules to existing DOM levels. That is, add the module *Level 1 Sixml* to support mark associations with default unqualified names, *Level 2 Sixml* to support mark associations with custom and default qualified names, and *Level 3 Sixml Load and Save* to support reading and writing of Sixml documents. (A *Level 3 Sixml* module would not be needed because typed mark associations are handled using existing interfaces defined in Level 3 Core.) This approach does not affect existing DOM applications, but it contradicts the apparent DOM convention that no module extends an interface defined by another module at the same level.

Alternative 3 is to add the Sixml interfaces to existing DOM modules. This approach allows creation of both DOM nodes and Sixml DOM nodes using the same DOM implementation because both DOM and Sixml DOM interfaces would be available simultaneously. An application navigating a Sixml DOM tree using the DOM interfaces would be able to access only SI, but it would be able to access mark associations in the same tree using the Sixml interfaces.

This approach, too, does not affect existing DOM applications, but the simultaneous availability of the two sets of interfaces can be occasionally confusing (to developers). However, the simultaneous availability of DOM and Sixml DOM interfaces can be handy at times, as Section 7.6.3.2 illustrates.

Alternative 4 is to revise DOM interfaces such that the Sixml functionality is added to DOM from the ground up. This approach has the same effect as Alternative 2, but without using extensions and without adding new modules. This approach alters some of the interfaces in existing DOM modules, and it requires changes to existing DOM implementations. Existing applications need not be changed, but they might need to be recompiled.

In Alternatives 1, 2, and 4, Sixml DOM functionality would be available through DOM interfaces. For example, the method appendMarkAssociation to add a mark association to a target node would be available in the class Node, and the class SixmlNode would cease to exist.

Our description of Sixml DOM in Section 7.4 corresponds to Alternative 3. Appendix B lists the complete IDL definition for this alternative. The IDL definitions for all four integration alternatives are available online [117]. We chose Alternative 3 because of the ability to use both DOM and Sixml DOM interfaces. We have also implemented Alternative 4 to see if it performs better than Alternative 3.

7.6. Evaluation

We have evaluated Sixml DOM by implementing the design presented in Section 7.4 and by running experiments. We have evaluated the Sixml mark-association types and Sixml DOM by employing them in different applications. We first describe the Sixml DOM implementation and some applications, followed by experimental results.

7.6.1. Implementation

We have *three* implementations of Sixml DOM in C#: two implementations in the extension strategy (Alternative 3 outlined in Section 7.5) and one in the revision strategy (Alternative 4 in Section 7.5). The first implementation in the extension strategy is based on the DOM implementation in Microsoft's distribution of the .NET Framework (.NET) [129]. The other two implementations are based on Mono's distribution (Version 1.2.5.1) [106] of .NET.

We refer to our three implementations as *Microsoft Extension* (MSX), *Mono Extension* (MNX), and *Mono Revision* (MNR), respectively. We refer to the base

DOM implementation for MSX as *Microsoft Base* (MS), and refer to the base of MNX and MNR as *Mono Base* (MN). We have the source code for MN, but not for MS. We used the same source code to build MSX and MNX, and adapted much of that source code in MNR.

We had initially implemented only MSX. Its performance overhead (compared to its base, MS) was more than what we anticipated. We then implemented MNX and MNR to test if the overhead in providing Sixml DOM functionality can be reduced. Section 7.6.3 compares the performance of the three implementations.

All three Sixml DOM implementations conform only to DOM Level 2 Core, because the base DOM implementations in .NET conform only to Level 2 Core [172]. That is, .NET does not implement the classes LSParser and LSSerializer in Figure 7.2, but implements all the other shaded classes. We have implemented all the Sixml-specific classes (that is, the classes not shaded), except SixmlParser and SixmlSerializer (because their respective base classes do not exist).

Although .NET does not implement the classes LSParser and LSSerializer, it does provide routines to parse and serialize XML data. We have implemented the parsing and serialization routines for Sixml data on top of these .NET routines.

7.6.1.1. Overview

Each of the three Sixml DOM implementations has the following capabilities:

- Associate any number of marks with any of the six types of nodes (element, attribute, text, CData section, comment, and PI) identified in Section 7.3.1.2, using the mark-association element types introduced in Section 7.3.2.
- Detect mark associations according to Rules 2 through 5 listed in Section 7.4.2.2. Rule 1 is not implemented because the base DOM implementation does not support typing. (The base .NET implementation conforms only to Level 2 Core.)
- Derive a node's value from context information as described in Section 7.4.3.4.
- Serialize a Sixml document using the deterministic procedure outlined in Section 7.4.4. Also, a developer may choose the scope of serialization: only SI; SI and mark associations; or SI, mark associations, and mark descriptors.
- Use any mark repository implementation that conforms to the specification in Section 7.4.3.1. Implementations of a transient and a persistent repository are included (in the form of the classes TransientRepository and PersistentRepository shown in Figure 7.2). The persistent repository implementation manages marks stored in any data source (such as a database created using MS SQL Server 2005 [99]) that complies with the OLE DB specification [98]. *OLE DB* is an object-oriented API that presents a row-set interface to data that may or may not be stored in a relational database.
- Manipulate marks using any concrete implementation of the classes Mark and MarkFactory described in Section 7.4.3.2. Implementations for SPARCE descriptors and XPointer descriptors are included. (See SPARCEMark, SPARCEMarkFactory,

XPointerMark, XPointerMarkFactory in Figure 7.2.) Multiple mark implementations may be used with the same Sixml document. For example, in Sixml document in Figure 7.1, the element TMark might use an XPointer mark descriptor, but AMark might use a SPARCE descriptor. (Figure 4.20 shows such descriptors.) Manipulating this document using Sixml DOM would then result in the simultaneous use of both the SPARCE and XPointer mark implementations at run time.

- Retrieve both mark descriptors (from a mark repository) and mark context (from the base layer) *on demand*, without special effort on the part of implementers of mark repositories and context agents. Also, mark context is retrieved using the *bulk accessor* component described in Chapter 6. The bulk accessor may be configured (for example, the pooling policy may be altered) independently of any concrete mark implementation.
- Share mark descriptors and context information when a mark is used more than once in the same document. For example, the elements AMark and EMark in Figure 7.1 would share both the mark descriptor and context information because the two mark associations involve the same mark.

The following list presents some high-level implementation statistics (as of this writing) to create the MSX and MNX implementations:

- Number of interfaces: 7
- Number of classes: 23

- Number of source files: 17
- Number of lines of code: 5,771
- Estimated time spent on implementation: 145 hours

The following list presents some high-level implementation statistics for the MSR implementation:

- Number of new interfaces: 7
- Number of new classes: 14 (Sixml code is added to existing DOM classes)
- Number of new source files: 2
- Number of source files shared with MSX and MNX: 10
- Number of lines of code shared with MSX and MNX: 3,838
- Number of new lines of code: 469
- Number of changes to base DOM implementation to add Sixml capability: 57
- Estimated time spent on implementation: 75 hours

7.6.1.2. Experience

We now share our experience dealing with some design and implementation issues related to creating mark associations. These issues are due to constraints on creation of nodes in DOM.

A DOM implementation may include several classes that implement a node type (such as element), but a factory method (for example, `createElement`) to create a node can use only the node's name (that is, the local name in DOM Level 1; the combination of the local name and the namespace URI in other levels) to choose the instantiated class. For example, the factory method createElement in Sixml DOM instantiates the class MarkDescriptor when an element's local name is "Descriptor" and the namespace URI denotes Sixml, but instantiates the class SixmlElement in other cases. This choice allows strong typing of mark descriptors.

The constraint that only a node's name be used to determine the node's class prevents us from making a mark association an instance of the class MarkAssociation, because we allow custom names for mark associations, as reflected in the rules listed in Section 7.4.2.2. For example, Rule 3 permits a mark-association element of any name, and allows the element's attribute sixml:type to convey the type of the association. However, an element's attribute is created *after* the element is created. Thus, the method createElement is forced to make a mark association an instance of the SixmlElement, causing loss of type-checking benefits for mark associations.

We remedy this situation with a combination of type casts and run-time checks. We define an interface IMarkAssociation that defines the functionality specific to mark associations, and implement this interface explicitly in the class SixmlElement. In C#, an *explicitly implemented interface* allows a class to implement many methods with the same signature, but identify each implementation with a different interface [23]. The actual method invoked depends on the *compile-time* type of the calling instance (which is different from *polymorphism*, where the method invoked depends on the *run-time* type of the calling instance; with polymorphism, a class cannot define two methods with the same signature). For example, SixmlElement has two implementations of the property childNodes. One implementation is identified with SixmlElement and returns a (possibly empty) list of child nodes. The other implementation is identified with IMarkAssociation and always returns an empty list, because a mark association has no child nodes.

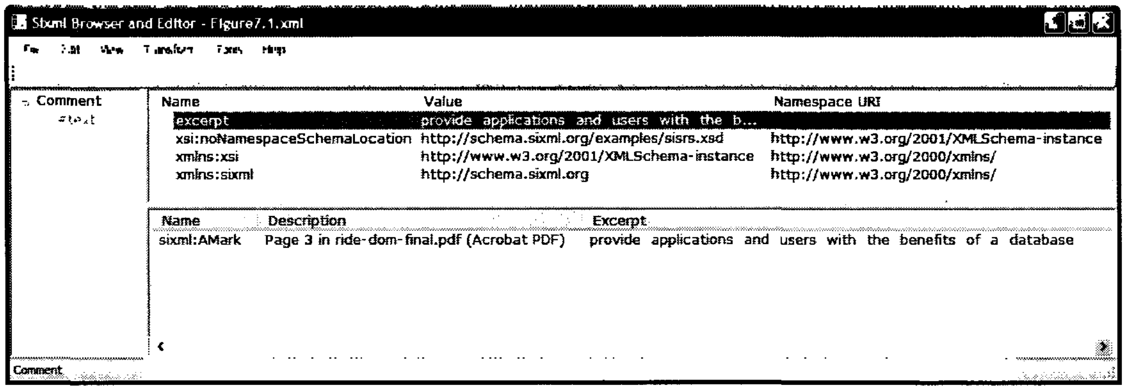
A disadvantage of using explicit interface implementation to work around the aforementioned problem is that *any* instance of SixmlElement can be cast as IMarkAssociation. So, to prevent misuse of IMarkAssociation, we check at run time if a method invoked is appropriate for the instance's role. For example, we disallow appending a child node to an element that represents a mark association, and bar retrieval of context information from an element that is not a mark association.

The explicit use of the interface IMarkAssociation provides strong type checking for mark associations, and the run-time checks provide operational consistency. However, depending on the access pattern, the run-time checks can introduce non-trivial run-time overhead.

The node-creation constraint also affects the attributes and child nodes of mark associations and mark descriptors because a DOM node is created independently of its use context (and then added to another node). For example, it should not be possible to associate marks with an attribute of a mark association, but the method createAttribute is forced to always instantiate the class SixmlAttribute, allowing mark associations to be added to an attribute of a mark association or a descriptor. We work around this problem using run-time checks, and by *lazily* building the data structures that hold mark associations, but certain memory and processing-time overheads are unavoidable.

7.6.2. Applications

We have created a general-purpose tool that can use any of our three Sixml DOM implementations to browse and edit arbitrary Sixml documents. Figure 7.7 shows the Sixml document of Figure 7.1 being viewed using the tool. The tree on the left shows the name of the document element Comment and its text child. (DOM [35] fixes the string #text as the name of any text node.) The top pane on the right lists the attributes and the namespaces of Comment. The attribute excerpt is selected and its lone mark association is listed in the bottom pane. The partial data shown for the value of the attribute and the excerpt retrieved from the associated mark are the same because the attribute's value is set to be the mark's text excerpt.



Name	Value	Namespace URI
excerpt	provide applications and users with the b...	
xsi:noNamespaceSchemaLocation	http://schema.sixml.org/examples/sisrs.xsd	http://www.w3.org/2001/XMLSchema-instance
xmlns:xsi	http://www.w3.org/2001/XMLSchema-instance	http://www.w3.org/2000/xmlns/
xmlns:sixml	http://schema.sixml.org	http://www.w3.org/2000/xmlns/

Name	Description	Excerpt
sixml:AMark	Page 3 in ride-dom-final.pdf (Acrobat PDF)	provide applications and users with the benefits of a database

Figure 7.7: The Sixml Browser and Editor. The Sixml document of Figure 7.1 is shown

We have used Sixml in *four* SAs: Sidepad (introduced in Section 1.2.1), SuperMix (Section 1.2.2), the Superimposed Scholarly Review System (SISRS, Section 4.9.3), and the Superimposed System Information Browser (SSIB, Section 4.2). Sidepad represents its data in a proprietary format, but also exposes its data in Sixml format to support transformation and other activities. SuperMix, SISRS, and SSIB represent their data as Sixml documents. Each of these four applications is able to use any of our Sixml DOM implementations (because all three implementations present the same interface).

We have used Sixml to specify data mash-ups. A *data mash-up* is a document that contains information drawn from different sources [120]. (A data mash-up is different from a *mash-up application* that retrieves information from different sources. A mash-up application might produce a data mash-up.) For example, a document that describes comments over different documents, with each comment modeled as the Comment structure in Figure 7.1, would be a data mash-up, because the value of the attribute excerpt could be drawn from different documents for different comments. The Sixml document in Figure 4.26 and the output of the bi-level query in Figure 4.27 are also data mash-ups. (Section 11.2.1 discusses data mash-ups in detail.)

We have used Sixml DOM to manipulate and automatically *reconstitute* (that is, extract constituent parts from different sources) data mash-ups specified using Sixml. For example, Figure 7.7 (and Figure 7.5) shows the value of the attribute excerpt reconstituted according to the specification in Figure 7.1. A tool called *Mash-o-matic* [115] uses Sixml, Sixml DOM, and our bi-level query processor to respectively specify, reconstitute, and format map-based mash-ups.

Finally, we have used Sixml DOM to provide a run-time representation of Sixml documents to support bi-level querying. Chapter 9 illustrates this use.

7.6.3. Experiments

We now present the results of experiments on the three Sixml DOM implementations. For these experiments, all C# code was compiled using Microsoft Visual Studio 2005 [102]. The experiments were run in Microsoft's distribution of the .NET Common Language Runtime (Version 2.0) [128] on an Intel Core Duo 1.66 GHz processor [65] with 1 GB of main memory. The operating system was Microsoft Windows XP (Service Pack 2) [104].

We present the results of experiments that demonstrate the scalability of the Sixml DOM implementations and the savings possible by using Sixml DOM (compared to DOM) when retrieving mark associations and SI. We ran each experiment three times and report here the average results.

7.6.3.1. Overview of the Datasets

Table 7.2 lists the Sixml documents used in the evaluation. The documents are generated by the applications SISRS and SSIB, and are based on the schemas presented in Section 4.9. The number at the end of each document's name (in the first column) is the *size scale factor*. For example, the document SISRS-2 has twice the number of mark associations as SISRS-1; SSIB-8 has eight times the number of mark associations as SSIB-1. The documents SISRS-8 and SSIB-8 correspond to the datasets used to evaluate the bulk accessor (as described in Sections 6.4.2.3 and 6.4.2.4).

Table 7.2: Sixml documents used in the experiments to measure performance when retrieving mark associations and SI. The columns EMark, AMark, and TMark show a breakdown of the number of mark associations by mark-association type

			Number of mark associations			
Document	File size (KB)	Number of base documents	EMark	AMark	TMark	Total
SISRS-1	206	53	1,908	53	0	1,961
SISRS-2	414	106	3,816	106	0	3,922
SISRS-4	833	213	7,668	213	0	7,881
SISRS-8	1593	426	15,336	426	0	15,762
SSIB-1	3,243	18	0	25,922	12,961	38,883
SSIB-2	6,486	18	0	51,850	25,925	77,775
SSIB-4	12,987	18	0	103,710	51,855	155,565
SSIB-8	26,107	18	0	207,426	103,713	311,139

The third column in Table 7.2 lists the number of base documents each Sixml document references. The SISRS documents reference PDF fragments (as described in Section 6.4.2.3), whereas the SSIB documents reference cells in MS Excel spreadsheets (as outlined in Section 6.4.2.4). For the SSIB dataset, we used only event information because we did not have error reports and update history for all computers. Using only event information ensured that the number of mark associations and SI scale up uniformly, and that the performance comparisons would be fair.

Table 7.2 also shows the breakdown of the number of mark associations by mark-association type, for each document. The SISRS documents use only mark associations of type EMark and AMark. That is, these documents do not contain TMark elements of the type shown in Figure 7.1. The SSIB documents use AMark and TMark elements, but no EMark elements. This variety allows us to test how the different types of mark associations affect performance.

7.6.3.2. Ease of Accessing Mark Associations and SI

We first compare the effort to access mark associations and SI using DOM to the effort to access the same information using Sixml DOM.

Figure 7.8 shows pseudo-code procedures, based on node type, to retrieve mark associations for a target node in a Sixml document using DOM. Each procedure retrieves a list of nodes from an appropriate containing element and tests if each node in the list represents a mark association. For example, the procedure to discover the mark associations for an element tests the element's child nodes. The procedure to find the mark associations for an attribute tests the child nodes of the attribute's owner element.

A C# implementation (available online [117]) of the procedures to retrieve mark associations using DOM contains about *382 lines*. In contrast, with Sixml DOM, mark associations are retrieved simply by using the property `markAssociations` on the target node. The procedure `SixmlDOMGetMarkAssociations` in Figure 7.9 illustrates the use of this property.

We do not show the procedures to retrieve SI using DOM because they are too long. For example, using DOM, about *355 lines* of C# code (available online) are needed to access only the following five types of SI nodes (related to a target node): parent node, first child, last child, next sibling, and previous sibling. (A node may have up to 19 related SI nodes.)

```
const string sixmlNSURI = "http://schema.sixml.org";
procedure DOMGetMarkAssociations_AnyNode(Node target)
switch(target.nodeType)
  case ELEMENT_NODE: DOMGetMarkAssociations_Element((Element)target);
  case ATTRIBUTE_NODE: DOMGetMarkAssociations_Attribute((Attr)target);
  case TEXT_NODE: DOMGetMarkAssociations_Other(target, "TMark");
  case CDATA_SECTION_NODE: DOMGetMarkAssociations_Other(target, "CDataMark");
  //similarly handle nodes of type COMMENT_NODE and PROCESSING_INSTRUCTION_NODE
procedure DOMGetMarkAssociations_Element(Element e)
for each node c in e.childNodes //mark associations are sub-elements
  if(c.nodeType == ELEMENT_NODE)
     if(IsMarkAssociation((Element)c, "EMark"))
        print(c.nodeName); //c is a mark association for e
procedure DOMGetMarkAssociations_Attribute(Attr a)
for each node c in a.ownerElement.childNodes //owner element has mark associations
  if(c.nodeType == ELEMENT_NODE)
    Element m = (Element)c;
     if(IsMarkAssociation(m, "AMark")) &&
        m.getAttributeNS(sixmlNSURI, "target") == a.nodeName)
        print(m.nodeName); //m is a mark association for a
procedure DOMGetMarkAssociations_Other(Node n, string typeName)
if(n.parentNode != null)
   if(n.parentNode.nodeType == ELEMENT_NODE)//parent is or has mark associations
     Element p = (Element)n.parentNode;
     if(IsMarkAssociation(p, typeName)) //typeName is "TMark", "CDataMark", etc.
       print(p.nodeName); //p is a uni-mark association for n
     else if(IsMarkAssociation(p, typeName+"s")) //"TMarks", "CDataMarks", etc.
       for each node c in p.childNodes //p is a multi-mark association
         if(c.nodeType == ELEMENT_NODE)
           if(IsMarkAssociation((Element)c, typeName))
             print(c.nodeName); //c is a mark association for n
//helper function to determine if an element represents a mark association
procedure IsMarkAssociation(Element e, string sExpectedLName)
if (e.namespaceURI == sixmlNSURI && e.localName == sExpectedLName)
  //the element's QName denotes a mark association: Rule 2, Section 7.4.2.2
  return true;
else //test if the attribute sixml:type indicates a mark association: Rule 3
  string qName = e.getAttributeNS(sixmlNSURI, "type");
  //skipped: parse qName and place constituent parts in variables prefix and lName
  return (e.lookupNamespaceURI(prefix) == sixmlNSURI && lName == sExpectedLName);
```

Figure 7.8: Procedures to get mark associations of a target node using DOM. Some code is omitted for brevity

```
procedure SixmlDOMGetMarkAssociations(SixmlNode target)
for each mark association m in target.markAssociations
  print(m.nodeName); //m is a mark association for target
procedure SixmlDOMGetSI(Node target) //use only DOM to access SI
if (target.parentNode != null) print(target.parentNode.nodeName);
for each node c in target.childNodes print(c.nodeName);
if (target.nodeType == ELEMENT_NODE) //print attributes
  for each attr a in target.attributes print(a.nodeName);
```

Figure 7.9: Procedures to get mark associations and SI using Sixml DOM

In contrast, Sixml DOM allows SI to be retrieved using just the DOM interfaces. For example, the DOM-defined properties `parentNode` and `childNodes` return the parent node and list of child nodes, respectively. The procedure `SixmlDOMGetSI` in Figure 7.9 illustrates the use of the DOM interface to retrieve SI.

A drawback when using DOM to manipulate a Sixml document is that the mark-association elements are *repeatedly* distinguished from other elements. In contrast, Sixml DOM distinguishes each mark association *only once*. Also, it performs the tests needed to distinguish mark associations *lazily* so that any performance penalty is incurred only when the mark associations contained in an element need to be distinguished from other elements.

7.6.3.3. Scalability

We now show how the run-time performance of the three Sixml DOM implementations scales up with the number of mark associations and SI. In this experiment, we traversed each Sixml document depth-first and retrieved the mark associations of each SI node in the document using the property `markAssociations` (as in Figure 7.9). We then computed a *speed scale factor* for each document in a dataset as the ratio of the time to traverse mark associations in the document to the time to traverse mark associations in the *first document* in its set (that is, in the documents SISRS-1 and SSIB-1). Similarly, we also computed the speed scale factor to retrieve only the SI portion of each document using the property `childNodes` as in Figure 7.9.

Table 7.3 shows the time (in milliseconds) to complete 20 depth-first traversals of each Sixml document to retrieve all mark associations and SI (separately), in each of the three Sixml DOM implementations. The speed scale factor for each document is shown in parentheses. For example, using MSX, accessing all mark associations in the document SISRS-2 takes 2.3 times the time it takes for SISRS-1, but the same activity takes 2.1 times the time using MNR. (SISRS-2 has twice the number of mark associations SISRS-1 has.) By definition, the speed scale factor for the first document in each dataset is 1.

Table 7.3: Time (in milliseconds) to retrieve mark associations and SI (separately) over 20 iterations using the Sixml DOM implementations. The dashed line separates the documents in the SISRS dataset from documents in the SSIB dataset. A number in parentheses shows the speed scale factor. The speed scale factor for the first document in each dataset is 1.

	Time to access mark associations (ms)			Time to access SI (ms)		
Document	**MSX**	**MNX**	**MNR**	**MSX**	**MNX**	**MNR**
SISRS-1	62.5	93.8	78.1	57.3	78.1	72.9
SISRS-2	145.8 (2.3)	218.8 (2.3)	166.7 (2.1)	125.0 (2.2)	171.9 (2.2)	156.2 (2.1)
SISRS-4	338.5 (5.4)	442.7 (4.7)	364.6 (4.7)	286.5 (5.0)	416.7 (5.3)	359.4 (4.9)
SISRS-8	625.0 (10.0)	953.1 (10.2)	765.6 (9.8)	572.9 (10.0)	875.0 (11.2)	718.8 (9.9)
SSIB-1	1,463.5	1,807.3	1,708.3	1,442.7	1,911.5	1,739.6
SSIB-2	2,901.0 (2.0)	3,890.6 (2.2)	3,479.1 (2.0)	2,932.3 (2.0)	3,906.3 (2.0)	3,500.0 (2.0)
SSIB-4	6,057.3 (4.1)	7,828.1 (4.3)	6,906.3 (4.0)	5,963.5 (4.1)	8,041.7 (4.2)	7,203.1 (4.1)
SSIB-8	11,828.1 (8.1)	15,526.0 (8.6)	13,890.6 (8.1)	11,963.5 (8.4)	16,656.3 (8.7)	14,828.1 (8.4)

Table 7.3 shows that, for all documents, MSX provides the fastest response, and MNX has the slowest response. MSX is faster because its base, MS, is faster than MN, the base of MNX and MNR [74]. MNR is faster than MNX because it does not have the

inheritance overheads of MNX, and Sixml DOM capability is added at the most optimal location within the base implementation.

The speed scale factor for MNR is always lower than or equal to that of MSX, though MSX has the better absolute speed. That is, the performance of MNR scales better than that of MSX.

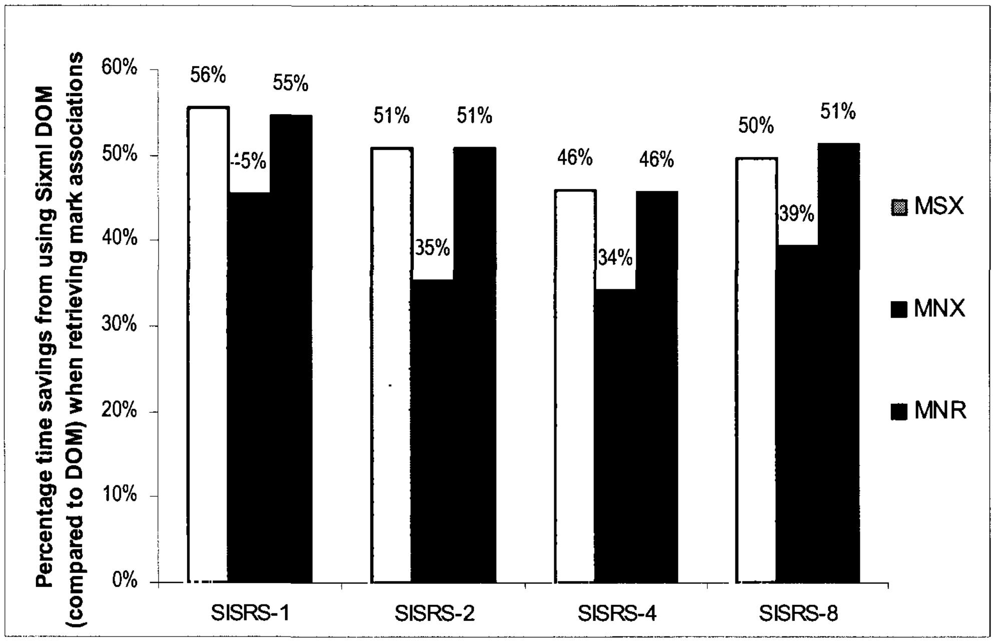
7.6.3.4. Savings when Traversing Mark Associations

We now compare the time to retrieve mark associations using DOM (as in Figure 7.8) to the time to retrieve the associations using Sixml DOM (as in Figure 7.9).

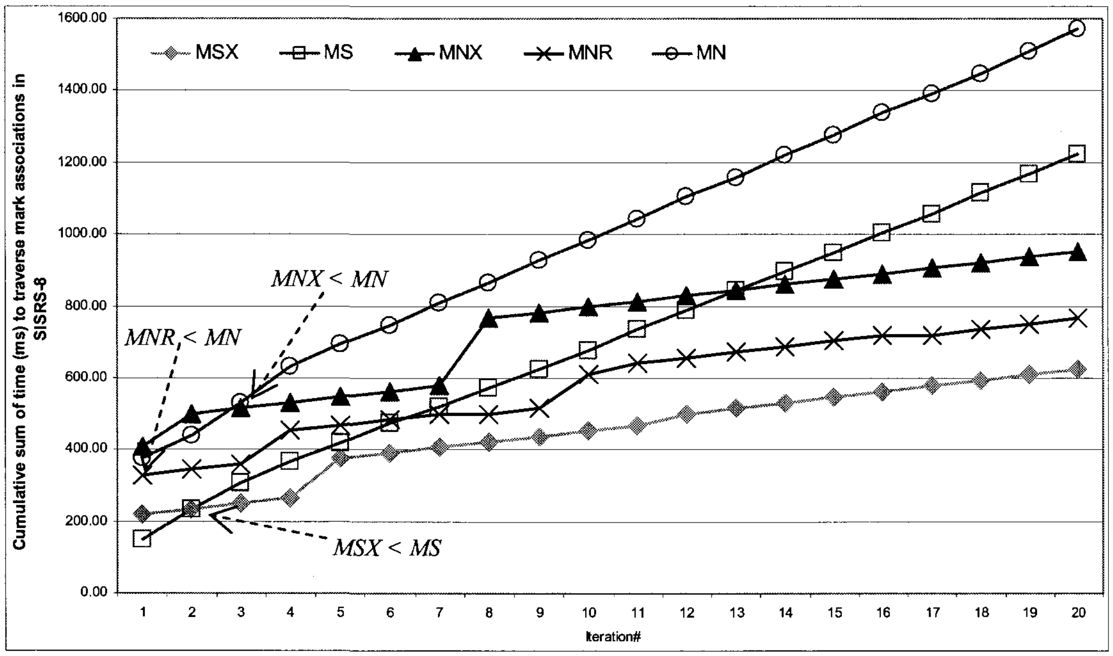
In this experiment, we measured the time to retrieve all mark associations in each Sixml document using each Sixml DOM implementation and computed the percentage time saved in comparison to the corresponding base DOM implementation. That is, we compare the performance of MSX to that of MS, and the performance of MNX and MNR to that of MN.

Unless explicitly specified, the savings (overhead) we discuss in the rest of this chapter correspond to the savings (overhead) obtained by using a Sixml DOM implementation in comparison to its base DOM implementation.

Figure 7.10(a) shows the percentage time savings when traversing mark associations in the SISRS documents. The figure shows that MNX saves the least, and the savings from MNR are comparable to that from MSX.

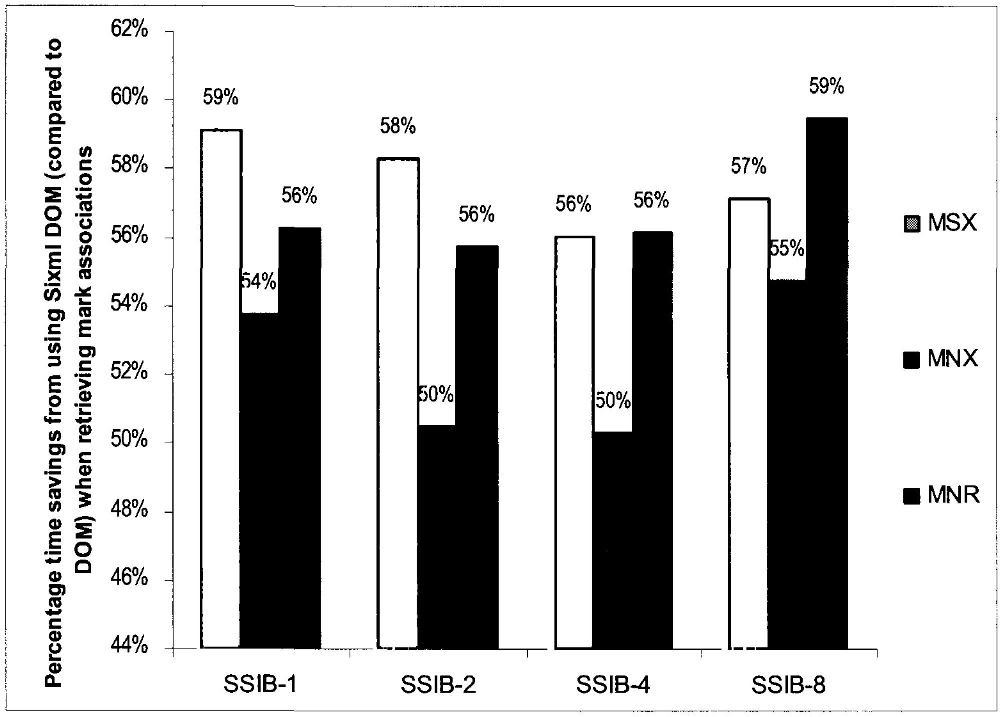


(a)

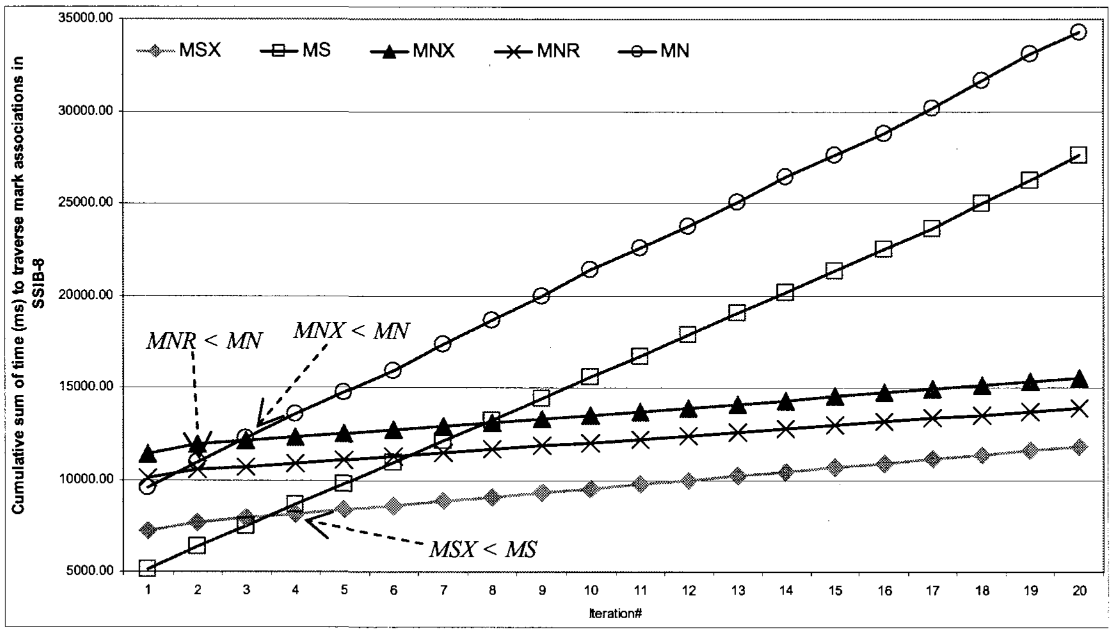


(b)

Figure 7.10: A comparison of the Sixml DOM implementations when traversing mark associations in the SISRS dataset. (a) Percentage savings due to Sixml DOM, compared to DOM; (b) Cumulative sum of time to access all mark associations in the document SISRS-8. The annotations call out the iteration at which a Sixml DOM implementation outperforms its base DOM implementation



(a)



(b)

Figure 7.11: A comparison of the Sixml DOM implementations when traversing mark associations in the SSIB dataset. (a) Percentage savings due to Sixml DOM, compared to DOM; (b) Cumulative sum of time to access all mark associations in the document SSIB-8.

Figure 7.10(b) shows the cumulative sum of the access time for the document SISRS-8 as the 20 iterations progress. MNR outperforms MN from the first iteration. MSX and MNX initially consume more time than their respective base implementations, but MSX outperforms MS after the first iteration, and MNX bests MN after two iterations. In general, a Sixml DOM implementation consumes more time initially because it extracts mark-association elements from their original location and inserts them under appropriate target nodes (as depicted in Figure 7.4). When using DOM, no changes are made to the tree, and an element's mark-association type is tested each time the element is visited (as illustrated in Figure 7.8).

Figure 7.11 compares the performance of the Sixml DOM implementations with DOM for the SSIB dataset. The performance for this dataset is similar to that for SISRS, but each Sixml DOM implementation needs more iterations (than needed for SISRS documents) to outperform its respective base DOM implementation. For example, Figure 7.11(b) shows that MNR outperforms MN only in the second iteration. This behavior is largely due to the presence of mark associations of type TMark, because the target text node that is initially represented as a child of a TMark is made a child of the parent of the mark-association element as described in Figure 7.4. (Compare the positions of the target text node in Figures 7.3 and 7.5.) Making this change consumes a non-trivial amount of time.

7.6.3.5. Savings when Traversing SI

We now discuss the percentage time savings and overhead (as applicable) when traversing SI using Sixml DOM. For simplicity, we depict overhead as negative savings.

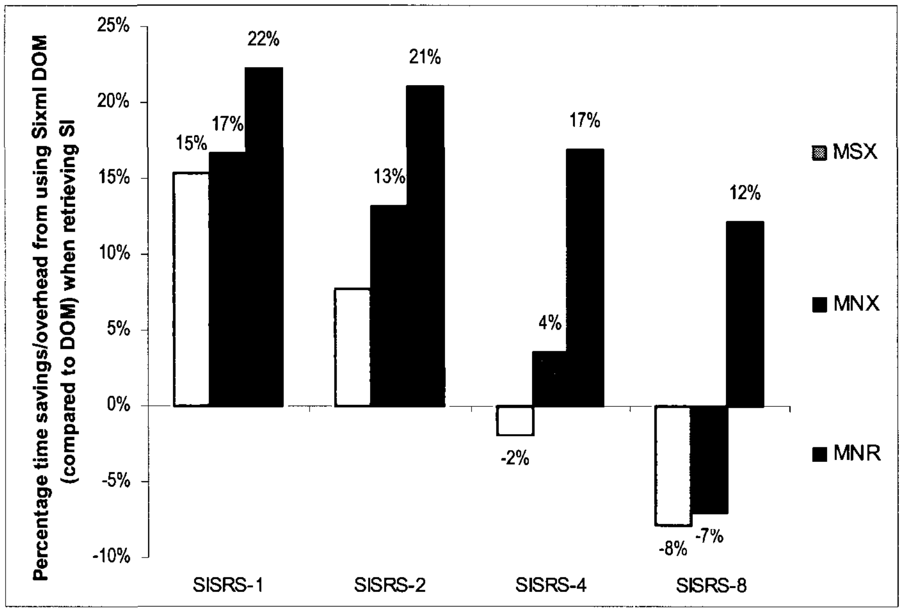
Figure 7.12(a) shows the percentage time savings for the SISRS dataset. In all cases, the savings decline as the amount of SI increases. MSX has overhead for the documents SISRS-4 and SISRS-8. MNX has overhead for only SISRS-8, but MNR provides savings in all cases. The reduction in savings from the first document to the fourth document is 23–24 percentage points for both MSX and MNX, but the reduction is only 10 percentage points for MNR. That is, as in the case of mark associations, MNR scales better.

Figure 7.12(b) shows the cumulative sum of the access times for the document SISRS-8 as the iterations progress. It shows that MNR outperforms MN after the sixth iteration. It also shows that MSX and MNX are on a converging course with their respective base implementations. However, the relatively large number of iterations needed for convergence might make MSX and MNX unsuitable for traversing SI in some applications.

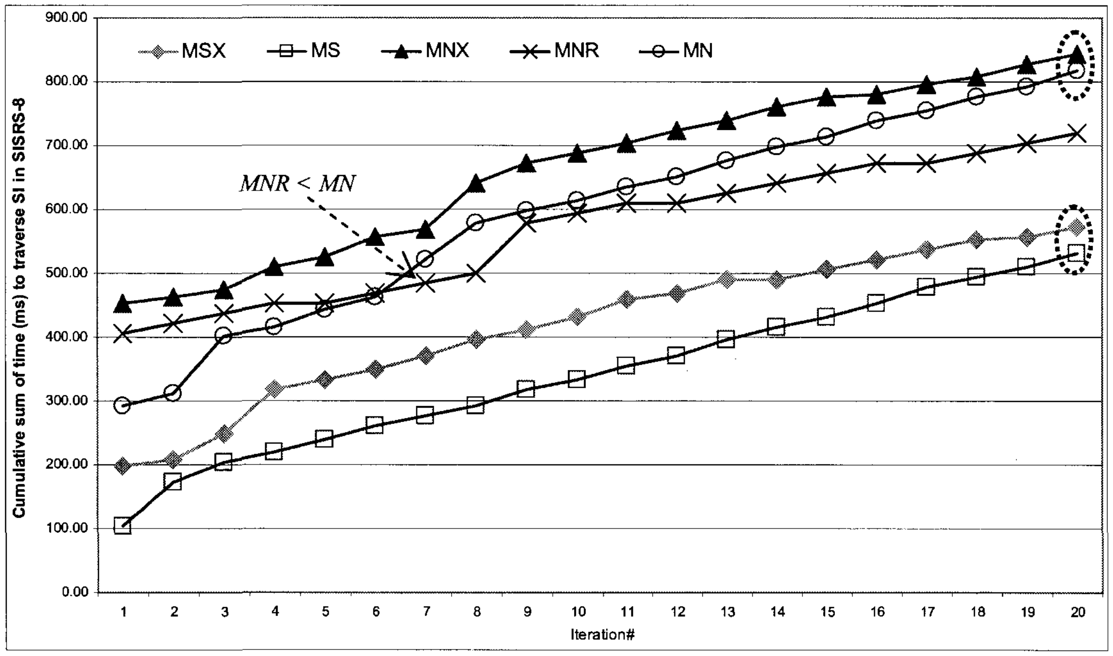
Figure 7.13(a) compares the performance of the Sixml DOM implementations with DOM for the SSIB dataset. As with the SISRS dataset, MNR provides the best performance, but the overall performance is contrary to that seen for SISRS. This change is again due to the presence of mark associations of type TMark: Whereas a TMark element hurts the performance of Sixml DOM when traversing mark associations, it hurts DOM when traversing SI because the target text node wrapped inside the TMark element is repeatedly unwrapped when using DOM (as outlined in Section 7.6.3.2). Thus, Sixml DOM performs better than DOM as the number of TMark elements increases.

Figure 7.13(b) shows the cumulative sum of the access time for the document SSIB-8 as the iterations progress. It shows that MNR outperforms MN after 12 iterations. It also shows that MSX and MNX are on a converging course with their respective base implementations.

Figures 7.12 and 7.13 show that the savings when accessing SI using Sixml DOM is less than that obtained for mark associations. This difference is partly due to the overheads we called out in Section 7.6.1.2, and in the case of MSX and MNX, it is also due to inheritance overheads. For example, when the property `childNodes` is invoked to retrieve SI children as shown in Figure 7.9, the base DOM implementation of this property is also invoked (after performing a few checks). This overhead is not incurred when accessing mark associations. The inheritance overhead is not incurred in MNR when retrieving SI, because the base DOM implementation is directly altered.

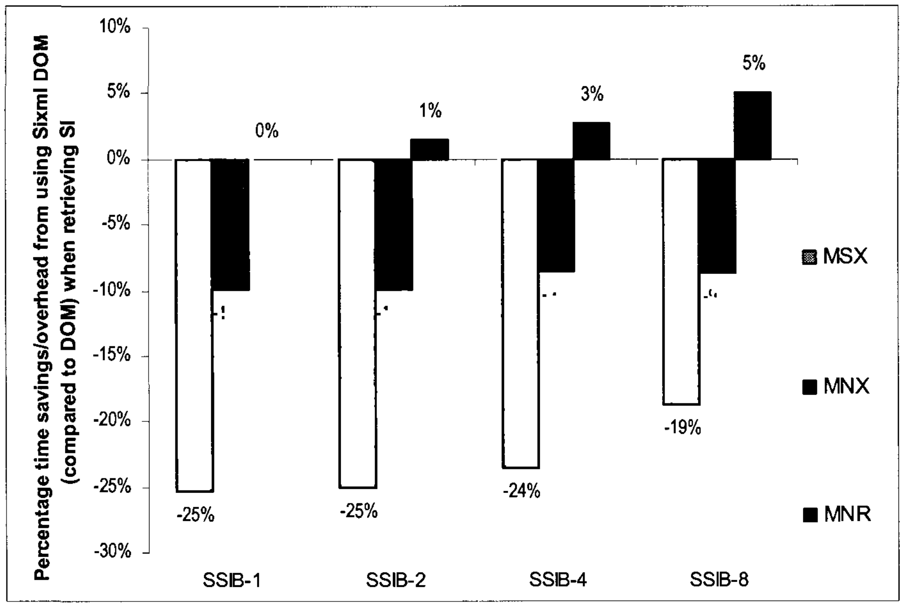


(a)

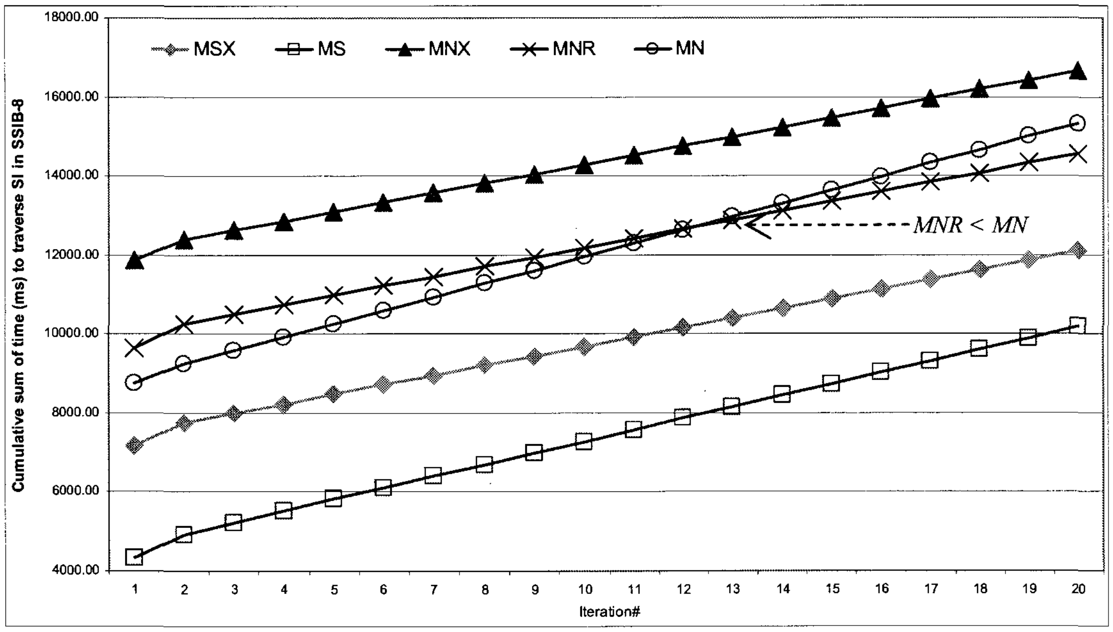


(b)

Figure 7.12: A comparison of the Sixml DOM implementations when traversing SI in the SISRS dataset. (a) Percentage savings (positive values) and overhead (negative values) due to Sixml DOM, compared to DOM; (b) Cumulative sum of time to access all SI in the document SISRS-8. The ovals highlight the converging course of MSX and MNX with their respective base DOM implementations



(a)



(b)

Figure 7.13: A comparison of the Sixml DOM implementations when traversing SI in the SSIB dataset. (a) Percentage savings (positive values) and overhead (negative values) due to Sixml DOM, compared to DOM; (b) Cumulative sum of time to access all SI in the document SSIB-8

7.6.3.6. Overhead to Traverse Non-Sixml Data

We also measured the performance of the Sixml DOM implementations when traversing *non-Sixml documents* (that is, XML documents with no mark associations). We conducted this experiment to see if Sixml DOM can be used to work with traditional XML documents as well. Also, a non-Sixml document is a good proxy for a Sixml document with few mark associations.

We report results for three non-Sixml documents: SIGMOD Record 1999, the XML index of issues of ACM SIGMOD Record [5] for the year 1999; XMark, a document from the XMark benchmark [143]; and MBench, a document from the Michigan benchmark [142]. The salient features of these documents are, respectively: size 484 KB and tree depth 4; 113.7 MB, depth 8; and 14.7 MB, depth 16.

Figure 7.14 shows the percentage overhead to traverse the three non-Sixml documents. The figure is oriented such that it can be easily compared with Figures 7.12(a) and 7.13(a). For each document, MNR has the least overhead and MSX has the most overhead. In general, the performance of a Sixml DOM implementation when traversing a non-Sixml document is similar to that of accessing SI in a Sixml document. (In fact, we use the same code in both cases.) For example, the trends seen in Figure 7.14 are similar to the trends seen in Figure 7.12(a) for the SISRS dataset. The trends in Figure 7.14 are dissimilar from the trends shown in Figure 7.13(a) for SSIB because of the absence of elements such as TMark that need to be unwrapped.

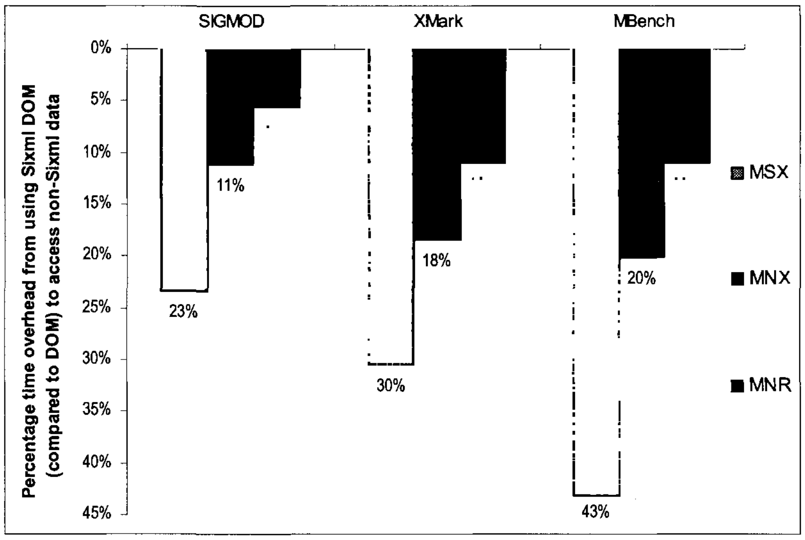


Figure 7.14: Overhead to traverse non-Sixml data using Sixml DOM, compared to DOM

7.6.3.7. Evaluation Summary

Using Sixml DOM to access mark associations and SI requires less development effort than using DOM to access the same information. Using Sixml DOM saves time when accessing mark associations, even for a small number of traversals over the document. However, using Sixml DOM to access only SI can have some overhead. Mark associations that wrap their targets (that is, mark associations that are not of type EMark or AMark) slow down retrieval of mark associations, but they speed up retrieval of SI.

The performance characteristics of a Sixml DOM implementation when accessing a non-Sixml document are similar to the characteristics seen when accessing SI in a Sixml document that does not contain mark associations that wrap their targets.

It might be better to use DOM to navigate some non-Sixml documents instead of using Sixml DOM implemented using the extension strategy. However, a developer is not required to exclusively choose DOM or Sixml DOM to work with all documents. Instead, he can switch between the two at run time by simply switching the document constructor used: `doc = new XmlDocument()` or `doc = new SixmlDocument()`.

Among the Sixml DOM implementations, MSX has the best absolute performance when traversing mark associations, SI, and non-Sixml data because its base, MS, is faster than MN. However, MNR scales best and gives the most savings (or has the least overhead) relative to its base DOM implementation. MNX generally underperforms MNR due to inheritance overheads.

Both the extension and revision strategies of implementing Sixml DOM have merits. A Sixml DOM implementation can be fast (as in MSX) and have low overheads (as in MNX and MNR) if the base DOM implementation is fast and the source code for the base is available. That is, the speed of MNX and MNR can be improved by improving MN. The overheads in MSX can be reduced with compile-time access to the source code for MS. Overheads could be further reduced by adding Sixml DOM functionality to a DOM implementation from the ground up.

7.7. Related Work

In this section, we provide an overview of two systems of embedding links in XML documents, and briefly touch upon DOM extensions specially defined for two XML-based markup languages.

7.7.1. Embedding Links

We first review XLink and Active XML, two systems of embedding links in arbitrary XML documents.

7.7.1.1. XLink

Like Sixml, the XML Linking Language (*XLink*) [164] also allows embedding of links in arbitrary XML documents. A link is to a resource (for example, a document) that can be addressed using a URI or an XPointer pointer. A resource may be *remote* (that is, it may reside outside the document in which the link is embedded) or it may be *local* (that is, it can be a part of the linking document).

An XLink link is expressed using a *link element*, which is any XML element that employs specific XLink-defined attributes such as xlink:type. (The namespace prefix xlink is associated with the URI http://www.w3.org/1999/xlink.) A link may be simple or extended. A *simple link* connects the link element (a local resource) to a remote resource, and is indicated by the value "simple" for xlink:type. For example, the following XML fragment links the element Comment with a PDF document.

```
<Comment excerpt="" xlink:type="simple" xlink:href="file://c:/ride-dom-final.pdf"/>
```

The value of the attribute xlink:href is always the URI of a remote resource. The optional fragment-specifier portion of the URI (that is, the part after the # character in the URI) may use an XPointer pointer to identify a part of a resource.

An *extended link* is indicated by the value "extended" for the attribute xlink:type. It links two or more resources. A link element that expresses an extended link uses a sub-element called a *locator* to identify a remote resource, and a sub-element called a *resource* to identify a local resource. A local resource can be the link element itself, or it can be another element within the current document. A local resource is indicated by the value "resource" for xlink:type.

In addition to specifying the participating resources, an extended link can also specify a role for each linked resource and indicate how the resource should be displayed when the link is activated. For example, a link can specify that a resource should open in a new window. Sixml does not inherently support role and activation specifications, but an SA developer is free to introduce attributes and elements in the SI schema to support these features. (We do not constrain an SI schema in any way.)

An XLink locator element (used to identify a remote resource) is comparable to a mark association. The attribute xlink:type is comparable to our attribute sixml:type because both attributes determine the owner element's function. We also allow a mark association's type to be conveyed via a schema, but XLink does not have typed locators. Additionally, the value of the attribute xlink:href (an XLink locator uses to identify a remote resource) is restricted to being a URI or an XPointer. In Sixml, a mark descriptor may have arbitrary structure and it may conform to any linking technology. See Section 4.6.3.

Unlike XLink, Sixml does not directly support links from a Sixml document to another part of the same document. However, a mark descriptor is free to identify any part of any document, including the current Sixml document.

In XLink, a link does not always imply a connection with the link element, whereas in Sixml, a mark association is always paired with some part of the linking document. However, an SA is free to ignore this pairing and support XLink-like semantics.

XLink allows a link only with an element in the linking document, but Sixml supports links to non-element content as well. Finally, XLink does not support deriving of content (for example, attribute values) from linked resources.

7.7.1.2. Active XML

Active XML (AXML) [3] provides a means to describe parts of an XML document intensionally using *service-call elements* that encode calls to *web services* [161] (which provide a means of executing code located on a remote computer). The following is a hypothetical AXML representation of a part of the information in the element Comment in Figure 7.1. (This representation is based on examples in an unpublished report [156] on AXML.)

```
<Comment xmlns:axml="http://activexml.net">
  <axml:sc>sixml.org/getExcerpt(<mark ID="23">)</axml:sc>
</Comment>
```

The element axml:sc denotes a service call, and its children elements denote service parameters. The URI sixml.org/getExcerpt identifies a hypothetical web service to obtain the text excerpt of a mark. At run time, the service-call element is *replaced* by the XML element that the web service returns. For example, the following AXML fragment shows a possible result of executing the example service call. Here, the result element Excerpt has replaced the service-call element.

```
<Comment xmlns:axml="http://activexml.net">
  <Excerpt>provides...</Excerpt>
</Comment>
```

No special DOM is defined to manipulate an AXML document, but a special query processor executes service calls, and replaces each service-call element with a result element. In contrast, we provide both a DOM and a query processor so that an SA developer can use the tool that is most appropriate to the task at hand. (Section 9.4 compares the AXML query processor to our bi-level query processor.)

An AXML document references programs (in the form of web services), but a Sixml document references data. External data (that is, the result of service calls) brought into an AXML document is not necessarily related to any part of the document specified extensionally, and it is not possible to distinguish external data from extensional data after replacement occurs. In contrast, Sixml makes the division between SI and the external data apparent.

AXML uses a *schema-language extension* to express the type of the result of a service call, because at the schema level, the element axml:sc represents both a service call and its run-time result. That is, the schema for axml:sc needs to describe two types, but neither XML Schema nor DTD support assigning two types to a single element. In contrast, the schema of a Sixml document can be expressed using only the standard XML Schema constructs.

An AXML service-call element can supply the content of an XML element, but unlike Sixml, it cannot supply values of parts such as attributes. Thus, the excerpt retrieved from a commented region is represented as an element in the example AXML fragment, not as an attribute as in the Sixml document in Figure 7.1.

In this chapter, we have not described a means to supply the content of an element in a Sixml document, but we do have the designs for a facility to achieve this goal. The facility uses the attributes valueSource and valueExpression in a mark association of type EMark, similar to the use of these attributes in other mark-association types (as described in Section 7.4.3.4). At run time, the part of the context information that the path expression in valueExpression selects would be added as the content of the target of EMark, in place of EMark. The EMark itself would be moved to the list of mark associations of the target element, as described in Section 7.4.2.1.

7.7.2. DOM Extensions

DOM extensions have been defined for *Mathematical Markup Language* (*MathML*) [39] and *Scalable Vector Graphics* (*SVG*) [151], which are markup languages for mathematical and graphics information, respectively. Like Sixml, these extensions define specialized classes for elements and attributes, but unlike Sixml, their factory methods choose a class to instantiate based only on the node's name. For example, in the DOM extension for MathML, the factory method createElement instantiates the class MathMLMathElement if the element's local name is math and the namespace URI is http://www.w3.org/1998/Math/MathML. (The element math is the top-level element in each MathML document or segment.)

In Sixml DOM, a mark-association element can be detected either by its name or by its type, and a mark association's type can be assigned without using a schema (but using the attribute `sixml:type`). Because the DOM extensions for MathML and SVG rely only on a node's name to determine the node class to instantiate, they are not impeded by the implementation and performance issues discussed in Section 7.6.1.2.

7.8. Summary and Conclusions

In this chapter, we have completed the discussion of *Sixml* (first introduced in Section 4.5.2), a representation of SI as XML using only standard XML constructs. Sixml provides a means to incorporate marks (that is, links) to heterogeneous information fragments in arbitrary XML documents. A mark may be associated with the following kinds of XML content: element, attribute, text content, CData section, comment, and processing instruction. We have arrived at this list of content kinds, and have chosen to represent a mark association as an element, after considering issues such as serialization and validation of mark associations. We have defined different element types for each kind of content with which a mark may be associated, using only the constructs available in XML Schema.

In this chapter, we have also described Sixml DOM, an extension of DOM to easily and efficiently manipulate Sixml data at run time. Using Sixml DOM, an SA developer can easily manipulate marks independently of the linking technology the marks employ. He can also access mark associations without regard for the schema used to represent them.

We have defined rules to detect mark associations when a Sixml document is parsed, and have provided a deterministic procedure to serialize a Sixml document using only the W3C recommended syntax. We have also defined an interface for mark repositories and outlined the expected behavior from a mark repository when a lookup operation is performed.

We have presented some thoughts on integrating Sixml DOM into DOM, outlined two strategies to implement Sixml DOM, and presented three implementations of Sixml DOM. We have also presented experimental results showing the savings achieved (or overhead incurred) from using Sixml DOM, in comparison to DOM.

The schema for mark-association elements, the alternative interface definitions for Sixml DOM, and the source code for the three Sixml DOM implementations are all available online from http://www.sixml.org.

Both Sixml and Sixml DOM support the *normalized* and *nested* representation schemes we identified for SI in Section 5.2. Both Sixml and Sixml DOM help us meet the goals G1 (SI-Schema independence), G2 (diversity and multiplicity of mark associations), and G3 (execution efficiency) identified in Section 5.3. In addition, Sixml also helps us meet the goal G5 (ease of query expression). Sixml DOM aids G3 by *lazily* retrieving mark descriptors (from a mark repository) and context information (from the base layer). For brevity, we omit summarizing how the other goals are aided.

Sixml and Sixml DOM are useful in a wide range of superimposed applications. We have illustrated the use of Sixml (see Section 4.9) and Sixml DOM (see Section 7.6.2) in three SAs: Sidepad, SISRS, and SSIB. Apart from their use in SAs, Sixml and Sixml DOM are also useful in declaratively producing *data mash-ups*, which are documents that contain information obtained from different sources.

This chapter concludes our discussion on representing and accessing bi-level information in the XML model. Chapter 9 describes the use of this chapter's developments in bi-level query processing.

8. A Model for Improving Query Expression and Execution

This chapter introduces the notion of *cloaking* (that is, temporarily hiding) parts of data from a query processor so that certain classes of queries can be expressed easily and executed efficiently. It provides a means to achieve our goals of ease of expression (Goal G5 in Section 5.3.1) and efficient execution (G3) for bi-level queries; SI-only-query preservation (G6); and compatibility with existing query languages (G7).

In this chapter, we define a formal model and an architectural reference model for a *cloaking query processor* (that is, a query processor that supports cloaking). The two models are independent of applications and data models (such as the relational and XML models). We also illustrate the benefits of cloaking in both bi-level-query and non-bi-level-query settings.

Chapter 9 describes a cloaking query processor that applies the models presented in this chapter to improve the expression and execution of XML bi-level queries.

8.1. Introduction

In this section, we give an informal introduction to cloaking using a *tree data model.* We then illustrate use and benefits of cloaking in both bi-level-query and non-bi-level-query settings. For simplicity, we limit this discussion to the XML model.

8.1.1. A Tree Model for Cloaking

Cloaking can be explained using a simple tree model in which tree nodes *and* operations have colors. A color is chosen from a *color set*, and a *cloaking scheme* assigns colors to tree nodes. A tree in which each node is colored using a cloaking scheme is a *cloaked tree*. Several cloaking schemes are possible.

An operation on a cloaked tree is performed not on the tree, but on a sub-tree called the *scope* of the operation. (We use the term *sub-tree* in the same sense the term *sub-graph* is used in relation to graphs [28].) This sub-tree is obtained by retaining only the input nodes (and the corresponding edges) that satisfy a given *test function*, which relates the operation's color with a node's color. That is, only the nodes that satisfy the test function are revealed to the operation, whereas the other nodes are cloaked.

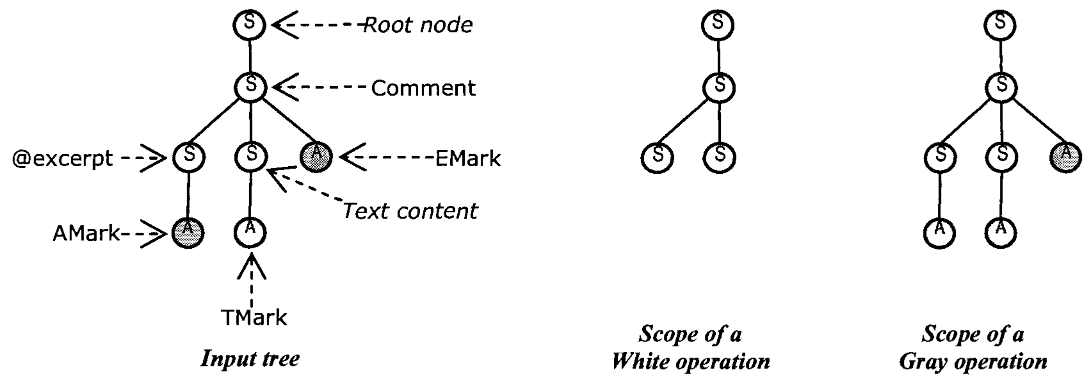


Figure 8.1: A cloaked tree and the scope of two operations over the tree. Colors are assigned from the totally ordered set {*White*, *Gray*}. Annotations map the nodes to the content of the Sixml document in Figure 7.1

A color set, a cloaking scheme, and a test function, all taken together, are called a *cloaking configuration*. Here is an example cloaking configuration:

- the totally ordered set {*White*, *Gray*}, where *Gray* > *White*;
- the cloaking scheme in which, for each node *n*, *Color(n)* ≥ *Color(Parent(n))*; and

- the test function *Color(operation)* ≥ *Color(node)*.

In this configuration, only *White* nodes are revealed to a *White* operation, but all nodes are revealed to a *Gray* operation. Figure 8.1 illustrates this example. Section 8.1.2 describes the annotations and the node labels used in this figure.

8.1.2. Application to Bi-level Querying

In a bi-level-query setting, cloaking can preserve SI-only queries because a Sixml document can be represented as a tree, and a query can be seen as an operation on the tree. The tree representation is based on the XPath data model [166], which produces a tree similar to a Sixml DOM tree. Section 9.1.1.1 introduces the XPath data model. Section 9.2.2 discusses a bi-level query processor's representation of a Sixml document.

We first discuss cloaking applied to a Sixml document in the *normalized schema* (which includes SI and mark associations, but not mark descriptors or context information), and then discuss the *nested schema* (which includes mark descriptors and context information). Section 5.2 introduced these schemas.

The *input tree* in Figure 8.1 sketches the tree representation of the Sixml document in Figure 7.1. An SI node is labeled S, and a mark-association node is labeled A. The annotations map the nodes to the content of the source Sixml document. The attributes of mark-association nodes are excluded for simplicity.

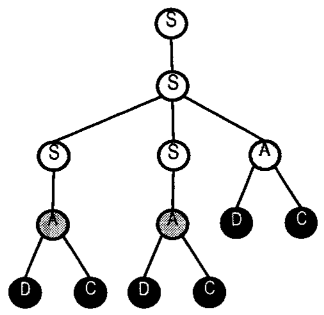
The example cloaking configuration in Section 8.1.1 can preserve SI-only queries over a Sixml document in the normalized schema if the SI nodes are colored *White* and the mark-association nodes are colored *Gray*. The nodes in the input tree in Figure 8.1 (and in the Sixml DOM tree of Figure 7.5) are colored in this manner. With this coloring, a *White* query would be an SI-only query, but a *Gray* query would operate over the entire document.

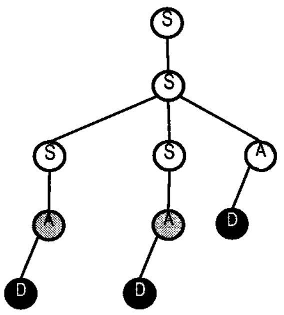
Extending the example color set to {*White*, *Gray*, *Slate*, *Black*} can distinguish SI-only queries even over a Sixml document in the *nested schema*. In addition, the extended color set can distinguish *three* other classes of queries: queries that involve mark associations (for example, count the number of mark associations employed); queries that involve mark descriptors (list the base documents referenced); and queries that involve context information (get the page number of a commented region).

With the extended color set, *White* queries continue to be SI-only queries; *Gray* queries operate only on SI and mark associations; *Slate* queries operate on SI, mark associations, and descriptors; and *Black* queries operate on the entire document.

The input tree in Figure 8.2 shows the nested schema version of the Sixml document in Figure 7.1 as a cloaked tree. The nodes are colored from the totally ordered set {*White*, *Gray*, *Slate*, *Black*}. As in Figure 8.1, an SI node is colored *White* and a mark-association node is colored *Gray*. In addition, a mark descriptor is colored *Slate*, and context information is colored *Black*. Also, a mark descriptor is labeled D, and context information is labeled C. For simplicity, the details of mark descriptors and context information are omitted.

Figure 8.2 includes the scope of *Slate* and *Black* queries. The scopes of *White* and *Gray* queries are exactly as in Figure 8.1.





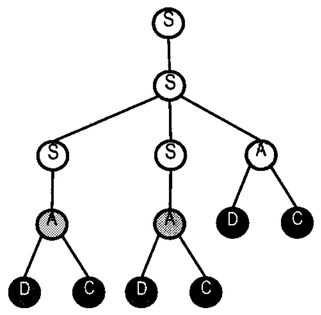


Figure 8.2: A cloaked tree for a Sixml document in the nested schema and the scope of two classes of queries. (Figure 8.1 shows the scope of two other classes of queries.) Colors are assigned from the totally ordered set {*White*, *Gray*, *Slate*, *Black*}

8.1.3. Non-Bi-level-Query Applications

Cloaking can be useful in non-bi-level-query settings as well. For example, if different versions of a document (such as source code) are represented as an XML document, cloaking can limit the version of the document that is exposed to a query. Cloaking can also be useful in data privacy and security applications [86]. In the rest of this section, we introduce the use of cloaking in an application involving spreadsheet data.

Microsoft Excel [96] (Excel) allows a spreadsheet, or a range of cells in a spreadsheet, to be saved as an XML document, but much of the XML document generated relates to the presentation of the spreadsheet (for example, the height and color of a cell). Figure 8.3 shows a part of the XML data generated for a spreadsheet with just one cell. The ellipses indicate content edited for brevity. The portions with gray background indicate presentation markup; the other portions indicate spreadsheet data. For example, the element Styles defines different display styles (using elements named Style) and the attribute ss:StyleID associates a style with parts of the data. The elements Row and Cell define spreadsheet data.

```
<Workbook xmlns="..." xmlns:o="..." xmlns:x="..."  xmlns:ss="..." xmlns:html="...">
 <Styles>
  <Style ss:ID="Default" ss:Name="Normal">...</Style>
  <Style ss:ID="s23">...</Style>
  <Style ss:ID="s30">
   <Alignment ss:Vertical="Top"/>
   <Borders>
    <Border ss:Position="Left" ss:LineStyle="Continuous" ss:Weight="2"/>
    <Border ss:Position="Right" ss:LineStyle="Continuous" ss:Weight="2"/>
   </Borders>
   <Font ss:FontName="Times New Roman" x:Family="Roman" ss:Size="11" ss:Color="#000000"/>
  </Style>
 </Styles>
 <Worksheet ss:Name="Sheet1">
  <Table ss:ExpandedColumnCount="1" ss:ExpandedRowCount="1" ss:StyleID="s23">
   <Column ss:StyleID="s23" ss:AutoFitWidth="0" ss:Width="91.5"/>
   <Row ss:Height="15">
    <Cell ss:StyleID="s30">
     <Data ss:Type="String">Arnold Ice Cave</Data>
    </Cell>
   </Row>
  </Table>
 </Worksheet>
</Workbook>
```

Figure 8.3: Partial XML data generated for a single cell in a Microsoft Excel spreadsheet. Fragments with clear background represent spreadsheet data. Fragments with gray background indicate presentation markup

In this setting, cloaking presentation markup (such as the element Styles and the attribute ss:StyleID) can improve the expression and execution of *data-only queries*, which are queries that read and return only the spreadsheet data.

The example cloaking configuration in Section 8.1 can cloak presentation markup from data-only queries, if the root node and the nodes representing spreadsheet data are colored *White*, and the presentation nodes are colored *Gray*. A data-only query would be colored *White*, whereas the other queries would be colored *Gray*.

8.1.4. Benefits from Cloaking

For certain classes of queries, cloaking can ease query expression and it can improve query-execution performance. We first discuss ease of expression and then discuss execution performance.

Ease of query expression: Cloaking allows the use of lightweight languages such as XPath in place of languages such as XQuery and XSLT [176, 177]. For example, consider the task of retrieving comments, minus the embedded mark associations, from a Sixml document that contains comments using the Comment structure in Figure 7.1. Without cloaking, the following XQuery query would be needed to complete this task. This query explicitly copies the text content of each Comment element, and explicitly leaves out the embedded mark associations.

```
<result> {
   for $c in fn:doc("comments.xml")//Comment
   return <Comment>{$c/text()}</Comment>
} </result>
```

Without cloaking, the simple XPath expression `//Comment` cannot accomplish this task, because XPath cannot remove the child nodes of a node it returns. Specifically, in this case, XPath cannot remove the mark associations contained in each Comment element. However, when mark associations are cloaked, the expression `//Comment` would accomplish the task, because only SI would be revealed to the query processor.

Similarly, with the XML data generated from Excel, when the presentation markup is cloaked, the XPath expression `//Cell` returns only spreadsheet cell data, automatically excluding the presentation attribute ss:StyleID. Achieving the same result without cloaking would require the following XQuery query:

```
<result> {
   for $c in fn:doc("workbook.xml")//Cell
   return <Cell>{$c/Data}</Cell>
} </result>
```

Query-execution performance: Cloaking can speed up query execution in two ways. First, it eliminates the need for languages such as XQuery and XSLT, which always construct new result nodes. Using XPath saves execution time because XPath returns existing nodes. Second, cloaking can reduce the number of nodes the query processor visits, further reducing execution time. For example, without cloaking, when executing the expression //Comment over the input tree in Figure 8.2, the query processor examines *all* elements in the document, including the mark-descriptor and context information. (The descriptor and context information for a mark can have arbitrary structure, and retrieving context information from the base layer can consume a large amount of time.) In comparison, with cloaking, the query processor examines only Comment and the embedded mark-association element. (The mark-association element is examined, but is not output because it is not SI.) Also, context information would not be retrieved from the base layer.

Similarly, with the Excel-generated XML data, executing the data-only query //Cell requires the query processor to examine 26 elements without cloaking, but with the presentation markup cloaked, the processor needs to examine only eight elements. (For brevity, we omit calling out the elements examined in each case.)

The savings in the example queries (`//Comment` and `//Cell`) are due to the cloaking scheme in use, which allows the child nodes of a node to be skipped if the node is cloaked. Our formal model (described in Section 8.2.1) for a cloaking query processor does not require this behavior from a cloaking scheme, but we expect this behavior to be fairly common in practice.

Cloaking also has the potential to save memory during query execution because a cloaking query processor might be able to avoid allocating memory for cloaked nodes. The memory savings can be substantial in a bi-level query setting if the processor obtains mark descriptors and context information *on demand.* Our bi-level query processor implementation exploits this capability. Section 9.3 describes the implementation and the savings obtained from using the implementation.

The aforementioned improvements in query-expression and execution due to cloaking make *ad hoc* querying and data exploration easy, because a developer can use *unfocused path expressions* without incurring the performance penalties normally associated with such expressions. A *focused path expression* is an expression that guides the query processor strictly along the path of interest. An unfocused expression does not guide the processor in this manner.

For example, the expression `/Comments/Comment/text()` is focused, whereas the expression `//text()` is unfocused. The latter expression asks for text nodes anywhere in the document, but when mark associations are cloaked, the query processor examines only SI, and returns only the nodes that represent comment text. Section

9.3.3.5 shows experimental results that highlight the benefit of using unfocused path expressions with cloaking.

8.1.5. Discussion

We call queries that reveal cloaked information *tachyon queries*, after beams of hypothetical particles called *tachyons* [44]. Works of science fiction (for example, Star Trek [91, 149]) often employ tachyon beams to reveal cloaked objects. In the example cloaking configuration of Section 8.1, a *Gray* query is a tachyon query.

We have thus far described a means of coloring both data and queries to selectively cloak and reveal data, but it is possible to achieve the same result by coloring only data. In this alternative, a query only sees *White* nodes. A node is colored *Gray* to cloak it; *White* to reveal it. In science-fiction parlance, this alternative is similar to a universe with no tachyon beams.

Coloring only data is simpler than coloring both data and queries, and it needs only two colors, but it requires updates to data depending on query needs. Updates can be time consuming, because they may need to examine many nodes (which we wish to avoid through cloaking). Also, updates can hinder the execution of multiple simultaneous queries over the same data.

We pursue the approach of coloring both data and queries, where multiple simultaneous queries with different visibilities can be executed over the same data without changing the data. The alternative of coloring only data can still be emulated by limiting a query's color to the first color in the set of colors used.

8.2. Modeling a Cloaking Query Processor

In this section, we present a formal model and an architectural reference model for a cloaking query processor. These models help us analyze cloaking independent of data models (such as the relational and XML models) and applications.

We also relate the architectural model to the formal model and show how the formal model applies to the relational and XML data models.

8.2.1. A Formal Model

We model the data input to a query processor as a forest of trees. We cloak tree nodes from queries by coloring nodes and queries. In Section 8.1, we illustrated cloaking by assigning one color to each node. In the formal model, we generalize this aspect and allow multiple colors per node. However, we limit a query's color to one.

A node is assigned colors from a *color set* (which is a non-empty set of colors) according to a *cloaking scheme.* A *test function* determines the nodes revealed to a query based on the query's color.

Our approach to cloaking does not really require colors, but we use them because they make it easy to visualize cloaking. Section 8.2.3 discusses this topic further.

We assume the following domains:

$\mathcal{B}$: The domain of truth values. $\mathcal{B} = \{\texttt{true}, \texttt{false}\}$.

$\mathcal{D}$: The domain of colors.

$\mathcal{C}$: The domain of color sets. $\mathcal{C} = \{C \mid C \subseteq \mathcal{D}\}$.

$\mathcal{K}$: The domain of cloaking schemes.

$\mathcal{N}$: The domain of nodes.

$\mathcal{F}$: The domain of forests.

$\mathcal{F}_k$: The domain of colored forests. $\mathcal{F}_k \subseteq \mathcal{F}$. See Definition 8.3.

$\mathcal{Q}$: The domain of user queries.

$\mathcal{T}$: The domain of test functions. See Definition 8.6.

Definition 8.1: A *tree* T is a tuple (N, E), where $N \subseteq \mathcal{N}$ is the set of tree nodes, and E is the set of edges between the tree nodes. Each node has a structured label. The label may include the set of colors associated with the node.

Definition 8.2: *Colors*: $\mathcal{N} \rightarrow C$ is a function that returns the colors assigned to a node. The function returns the empty set $\varnothing$ if no color is associated with the node. A cloaking scheme assigns a node's colors from a color set. See Definition 8.4.

Definition 8.3: A *forest* F is a tuple (N, E), where $N = \bigcup_{t=1}^{|F|} N_t$ and $E = \bigcup_{t=1}^{|F|} E_t$, where $|F|$ is the number of trees in the forest F. N_t and E_t denote the node set and edge set, respectively, of the tth tree in the forest. A forest is *colored* if $Colors(n) \neq \varnothing \ \forall \ n \in N$. Additionally, the forest is colored *from* the color set C if $Colors(n) \subseteq C \ \ \forall \ n \in N$.

Definition 8.4: A *cloaking scheme* is a function k: $\mathcal{N} \times \mathcal{C} \times \mathcal{F} \rightarrow \mathcal{N}$ that assigns colors from a color set C to a node n in a forest F. $Colors(k(n, C, F)) \subseteq C$. The nodes n and $k(n, C, F)$ can differ only by their colors.

A cloaking scheme colors each node individually, but within the context of a forest so it can examine other nodes and edges in the forest. For example, the example cloaking scheme of Section 8.1 colors a node based on the colors of the node's parent. Another scheme might assign colors based on the tree to which the node belongs.

A cloaking scheme might impose certain constraints on the color set and the input forest. For example, the example scheme of Section 8.1 requires the color set to be totally ordered. Another scheme might require the forest to contain a single tree.

Definition 8.5: The functional *Cloak*: $\mathcal{F} \times \mathcal{K} \times \mathcal{C} \rightarrow \mathcal{F}_k$ colors a forest $F = (N, E)$ from a color set C according to a cloaking scheme k to produce a colored forest F_k.

$Cloak(F, k, C) = F_k = (N_k, E_k)$, where:

$N_k = \{k(n, C, F) \mid n \in N\}$ and $E_k = \{(k(n_1, C, F), k(n_2, C, F)) \mid (n_1, n_2) \in E\}$

Definition 8.6: A *test function* t: $\mathcal{D} \times \mathcal{C} \rightarrow \mathcal{B}$ "tests" a color c against a color set C. For example, a test function might test if a query's color is one of the colors assigned to a node. Though the second input's domain is $\mathcal{C}$, in our use, its value will be a subset of the color set used to cloak the input forest. See the following definition.

Definition 8.7: The functional *Reveal*: $\mathcal{F}_k \times \mathcal{T} \times \mathcal{D} \rightarrow \mathcal{F}_k$ produces the scope of a query, based on query color, from a colored forest. Given a colored forest

$F_k = (N_k, E_k)$ (likely produced by the function *Cloak*), a test function *t*, and a query color *c*, the following holds:

$Reveal(F_k, t, c) = F_s = (N_s, E_s)$, where:

$N_s = \{n \mid n \in N_k \wedge t(c, Colors(n))\}$ and

$E_s = \{(n_1, n_2) \mid (n_1, n_2) \in E_k \wedge n_1 \in N_s \wedge n_2 \in N_s\}$

Because the set of edges E_s is equal to E_k restricted to the set of nodes in N_s, we express E_s as $E_{k \mid Ns}$ (read "E_k restricted to N_s").

Note that the revealed scope F_s might have more trees than the input colored forest F_k.

Definition 8.8: A *user query* q: $\mathcal{F} \times \mathcal{N} \rightarrow \mathcal{B}$ is a function that determines if a node *n* in the input forest *F* is passed to the output of the query processor.

This function models the actual query the user intends to execute. The function operates on one node at a time, but within the context of a forest so it can examine other nodes and edges in the forest. For example, a query might relate nodes in different trees.

A user query might create new nodes in addition to filtering input nodes, but such additions may be performed after the filtering.

Definition 8.9: The functional *Query*: $\mathcal{F}_k \times \mathcal{Q} \rightarrow \mathcal{F}_k$ computes the result of a user query over a colored forest (likely produced by the functional *Reveal*). Given a user query *q*, and the scope $F_s = (N_s, E_s)$, we have:

$Query(F_s, q) = F_r = (N_r, E_r)$, where $N_r = \{n \mid n \in N_S \wedge q(F_s, n)\}$ and $E_r = E_{s \mid Nr}$

The next section explores a possible means of effectively evaluating *Query* given a user query and a colored forest.

8.2.2. Architectural Reference Model

In this section, we present an architectural reference model for a cloaking query processor and relate it to the formal model presented in Section 8.2.1.

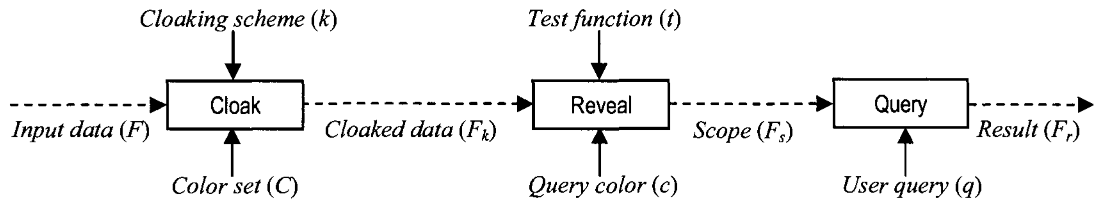


Figure 8.4: An architectural reference model for a cloaking query processor. Dashed arrows indicate data flow. Solid arrows denote parameters of the query-execution process

Figure 8.4 shows a reference model for a cloaking query processor. The modules Cloak, Reveal, and Query correspond respectively to the functionals *Cloak*, *Reveal*, and *Query* in the formal model. The symbols in parentheses in Figure 8.4 correspond to the symbols used in Section 8.2.1.

Given an input forest $F = (N, E)$, a cloaking scheme k, a color set C, a test function t, a query color c, and a user query q, the reference query processor produces a forest F_r.

$F_r = (N_r, E_r) = Query(Reveal(Cloak(F, k, C), t, c), q)$, where:

$N_r = \{k(n, C, F) \mid n \in N \wedge t(Colors(c, k(n, C, F))) \wedge q(F, k(n, C, F))\}$

$E_r = \{(k(n_1, C, F), k(n_2, C, F)) \mid (n_1, n_2) \in E \wedge k(n_1, C, F) \in N_r \wedge k(n_2, C, F) \in N_r\}$

The equation for N_r is obtained by expanding the functionals *Query*, *Reveal*, and *Cloak* using Definitions 8.9, 8.7, and 8.5, respectively. The equation for E_r is crafted such that the query processor's output includes all edges in the input forest, provided the corresponding nodes are also output.

The equations for N_r and E_r show that a cloaking query processor can execute a query without altering input nodes and without explicitly constructing the scope F_s of a user query q. (Note that the formula for N_r operates directly on the input entities.)

The equation for N_r shows two optimization opportunities for a cloaking query processor. First, because the test function and the user query are conjunctive terms, the processor is free to choose the order in which the terms are evaluated. This choice could even be based on cost estimates. Second, the processor might be able to combine the test function with the user query, so that the query is executed more efficiently.

8.2.3. Discussion

We now briefly discuss the use of color sets in our model, and the applicability of the tree model to the relational and XML data models.

We have thus far used color sets to model cloaking, but our model does not need colors. In reality, a "color set" can be any set of values, but the properties of the values influence the domain of test functions. For example, if the values are *nominal* (that is, the values can be tested only for equality), a test function would be limited to equality tests on individual values. However, the function can apply inequality tests as well if the values are *ordinal* (for example, the totally ordered color set in the example cloaking configuration of Section 8.1). An application can control the color sets used by choosing the domains $\mathcal{D}$ and C appropriately.

Our tree model works well in both the relational and XML data models. In the relational model, a relation instance is a tree (of height 2): The relation is the root node, a tuple in the relation is a child of the root node, and a field (that is, a column) in a tuple is a child of the tuple's node. In the XML model, an XML document is a tree in the data models of XPath, XSLT, and XQuery.

The tree model also works well with relational and XML query languages. In the relational model, an SQL query [92] operates on a set of trees and outputs a single tree. In XML, the query languages XPath, XSLT, XQuery, all operate on and produce trees.

8.3. Representing and Assigning Colors

Section 8.2.2 has shown that a cloaking query processor can execute a query without explicitly assigning colors to input nodes, but, for performance efficiency, it might be better to assign colors beforehand.

In this section, we briefly discuss some alternative means of representing and assigning colors to input nodes. Our cloaking model allows multiple colors per node, but, for ease of this discussion, we assume a node is assigned a single color.

An input node's color can be represented at the schema level or at instance level. It can be represented extensionally or intensionally. Also, the assignment can be implicit or explicit. (Section 8.4.2 reviews data provenance and annotation management systems that use some of these means to represent and assign data similar to colors.)

Schema-level and instance-level assignments: Assigning color at the schema-level makes color a part of a node's type, and all instances of a node type have the same color. (This approach obviously requires a schema.) If represented at the instance level, different instances of a node type can have different colors.

Representing colors at the instance level can pose problems in the relational model because an attribute's visibility to a query can vary between rows, and the relational model requires the same number of attributes for each row in a (result) relation. This difference in the visibility of the attribute between rows would need to be somehow reconciled. For example, a query processor can output a `NULL` value for the attribute in a row where the attribute is cloaked (but that value has to be distinguished from a `NULL` value in another row where the attribute is not cloaked).

Explicit, extensional assignment: Explicitly representing colors at the schema-level requires appropriate features in the schema language (and possibly in the data model). For example, new constructs need to be added to XML Schema [170] for the XML model. Similarly, in the relational model, the syntax of the SQL statements `CREATE TABLE` and `ALTER TABLE` need to be extended to associate colors with attributes. (Alternatively, parallel *color metadata* might be employed.)

Extensionally representing colors at the instance-level requires maintaining a "color" attribute for each node. The color attribute may be added as regular data or as *metadata* (that is, as secondary data). If color attributes are added as metadata, query languages need to provide a means to access metadata, but popular data models (including the

XML and relational models) and their query languages typically do not natively support the notion of metadata.

Explicit, intensional assignment: A node's color can be defined explicitly as a function of schema and data. For example, in the XML model, an XPath expression paired with a color might be used to assign colors to nodes. The color assignment can be at the schema level because XPath allows examination of schema information (for example, the name of an element).

In the relational model, an SQL query can be paired with a color to assign a color to nodes, but such assignments are possible where the schema is also stored in relations, if those relations are revealed. (Most current relational systems store schemas in relations. A query in a language such as SchemaSQL [85] can examine schemas regardless of how the schema is stored.)

Implicit assignment: Node colors can be implicitly assigned, instead of users explicitly assigning them. The assignments can be at the schema level, the instance level, or both. For example, a query processor can assign a node's color based on the node's name.

Implicitly assigning colors has the advantage that no additional data is needed to represent colors. The disadvantage is that implicit assignment affects all applications and queries.

For XML bi-level query processing, we assign node colors implicitly at the schema-level, using the 4-color cloaking scheme introduced in Section 8.1 (and illustrated in Figure 8.2). Section 9.2.6 provides the details.

8.4. Related Work

In this section, we review three systems: a tree model to ease navigation, a data provenance system, and an annotation propagation system. None of these systems is designed to cloak information from a query processor, but each system has some similarity to our approach to cloaking.

8.4.1. The Multi-colored Tree Model

The *multi-colored tree model* (MCT) [70] attempts to avoid update anomalies caused by data replication. It also attempts to simplify query expression over shallow trees that result from normalization of nested data. Thus, at a high level, MCT addresses some of the problems discussed in Section 5.2 in relation to the nested and normalized schemas for Sixml data. MCT achieves its goals by extending XQuery's data model (XDM) [175] and query language.

In MCT, each XML document tree has a color. An element can be used in multiple document trees, and is implicitly assigned the set of colors formed by collecting the color of each tree in which the element is used. Attributes, namespaces, and the non-element child nodes (such as text nodes) of an element are assigned the same set of colors as the element.

An MCT *database* is a sequence of colored trees that share the same root node. Figure 8.5 shows a database with two trees, each tree modeling a Sixml document. The shared root node is not shown. Elements P, Q, and R denote SI. The first tree is colored white, the second is colored gray. The trees share the mark-association element EMark. This element (and its attributes and descendants) is colored both white and gray. It has two candidate parents: P in the white tree; Q in the gray tree. It has no preceding sibling in the white tree, but it has one preceding sibling (R) in the gray tree.

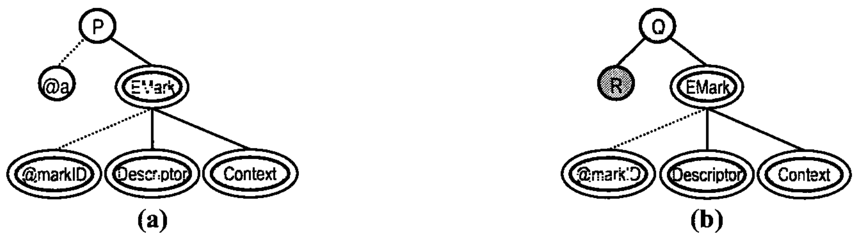


Figure 8.5: An example MCT database. (a) A white tree using a mark-association element; (b) A gray tree using the same mark-association element used in the white tree shown in Part (a)

In MCT, a node can appear only once in a given tree. For example, the element EMark associated with the element Q in the tree of Figure 8.5(b) cannot also be associated with R in the same tree. A *copy* of EMark can be associated with R, but doing so also creates copies of the child elements Descriptor and Context. This level of copying causes redundancy and can lead to update anomalies, defeating one of MCT's goals.

An MCT database is queried using *MCXQuery*, an extension of XQuery. MCXQuery allows each step in a path expression to choose the tree in which the navigation is executed. The tree in which navigation is performed is indicated by including the tree color at each step. For example, the expression `doc("SI")/{white}child::*` selects P; the expression `doc("SI")/{gray}child::*` selects Q.

The expression `doc("SI")/{white}descendant::EMark` selects the shared element EMark in the white tree. Changing the color in this expression to gray selects the same element, but in the gray tree. In the context of EMark, `{white}parent::*` selects P, but `{gray}parent::*` selects Q.

Our approach uses existing query languages as they are. A node's color is used to determine if a node is visible to a query; not to determine navigation paths. Also, in our approach, the entire query has the same color. Thus, all parts of the query operate over the same scope.

As discussed in Section 8.1.4, an XQuery constructor always returns a copy of a node. This action gives the copy a new identity, hindering MCT's goal of reusing nodes. To address this problem, MCXQuery redefines XQuery constructors to return a node *as is*. It introduces new constructors to create a copy of a node when the original XQuery semantics are desired. It also introduces a color constructor to designate a color to a result tree.

In our approach, nodes can be copied freely because the approach does not depend on node identity.

MCT cannot cloak data. For example, consider retrieving the SI element P from the MCT database in Figure 8.5. The expression `doc("SI")/{white}descendant::P` correctly selects P, but that element will include as its child the mark-association element EMark. The same is true for the expression `doc("SI")/{gray}descendant::Q`. In both cases, eliminating EMark requires a more complex XQuery query (as illustrated in Section 8.1.4).

8.4.2. Data Provenance

We now review a representative system that supports *data provenance* (which is a record of the derivation of data items [21]).

The system we review is due to Buneman and others [20]. They use colors to represent the provenance of a data item. They consider data in the *nested relational model* [2] (which allows complex values for attributes). They represent a relation instance as a tree similar to our approach described in Section 8.2.3. (The relation is at the root, a tuple is a child of the root, and a field in a tuple is a child of the tuple's node.) They add a primitive data type called *color* to the data model and allow *one* color to be *explicitly* associated with an object. An *object* is a generic term that means a relation, tuple, field, or any part of a complex field. An object associated with a color is a *colored object.* All sub-objects of a colored object (for example, tuples in a relation, and fields in a tuple) are also colored, but not necessarily in the same color as the parent object. This coloring method is similar to ours.

Figure 8.6(a) shows an instance of the relation `R(A, B)` modeled as a tree. The relation has one tuple `(3, 5)`. The nodes are colored for illustration.

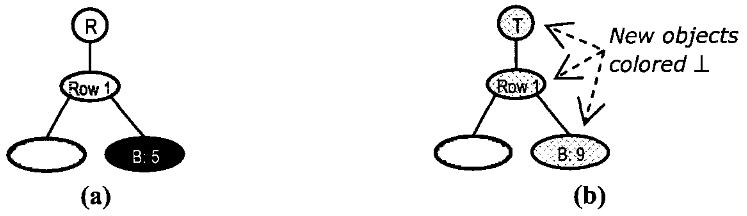


Figure 8.6: An illustration of data provenance. (a) A tree model of a relation R(A, B) with one tuple in the relation instance; (b) A model of the result of the query SELECT A, 9 AS B FROM R. The name T for the result table is chosen arbitrarily

A query is expressed in *nested relational algebra* [19] extended with "provenance aware" semantics. The extended algebra operators define *color-preserving functions* to propagate colors. A query may create new objects. A new object has the special color $\perp$.

Consider the SQL query `SELECT A, 9 AS B FROM R` (note the name of the second output attribute) over the relation modeled in Figure 8.6(a). This query creates a new relation instance with the tuple `(3, 9)`. The color of the output attribute `A` is propagated because that attribute is a copy of the input attribute `A`. The output attribute `B`, the output tuple, and the output relation have the color $\perp$ because they are all new. Figure 8.6(b) illustrates the query result.

Although our work is not about propagating colors, a node's color can be propagated in our approach as well (because a cloaking scheme can leave the node's color unchanged if the node is already colored). Also, as seen in the equation for N_r in Section 8.2.2, our formal model always colors result nodes.

Our formal model for cloaking does not assign colors to new nodes, because we only consider input nodes. (Again, see the equation for N_r in Section 8.2.2.) However, an additional cloaking scheme may be used to color output nodes, without affecting our model.

8.4.3. Annotation Propagation

We now review MONDRIAN [50], another system to represent and propagate annotations. This system includes an extension of the relational model to represent annotations, and an extension of relational algebra to propagate annotations.

MONDRIAN introduces the notion of a *block*, which is a non-empty subset of fields in a tuple. An *annotation* is a label attached to a block. An annotation is represented by a *color*, and a block with an attached annotation is a *color block.* A color block may have one or more colors (whereas Buneman and others allow only one color per object). A field (in a tuple) may be in zero or more color blocks. The set of fields in a color block can vary between tuples. That is, the definition of blocks and association of colors (to blocks) is at the instance level, not at the schema level.

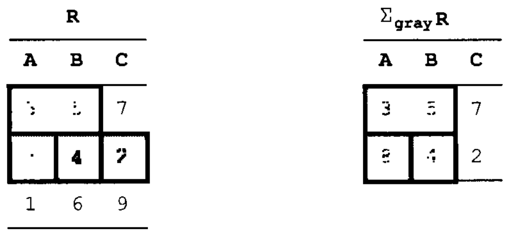


Figure 8.7: An instance of a MONDRIAN relation and the result of a block selection operation

Figure 8.7 shows an instance of the relation `R(A, B, C)`. Thick borders around cells indicate blocks. The gray block in the first tuple contains the fields `A` and `B`, but the gray block in the second tuple contains only `A`. Field `C` in the second tuple is in the slate block. Field `B` in the second tuple is in a block that is colored both gray and slate.

MONDRIAN uses a language called *color algebra* for querying. This algebra includes special operators to project, select, and merge blocks. Each operator in the algebra defines a *fixed* coloring function that decides how colors are propagated. Only the block-selection operator Σ accepts an *explicit* color; the other operators implicitly choose a color. We illustrate the use of coloring functions in MONDRIAN using the block-selection operator.

The block-selection operator does three things: It filters out tuples that do not contain any block of the input color, it assigns only the input color to blocks that contain the input color, and it removes all colors from blocks that do not contain the input color. For example, Figure 8.7 shows the result of the query Σ_{gray} R. The result excludes the last input tuple because that tuple does not contain any block colored gray. The input blocks colored gray are output as they are; the input block colored both gray and slate is colored only gray; and the slate block is not colored anymore.

In our approach, cloaking schemes are not fixed and they are independent of operators and queries. (In fact, the same cloaking scheme may be used for different queries.) Also, in our approach, a color is assigned to an entire query, not to individual operators in a query.

8.5. Summary and Conclusions

In this chapter, we have presented both an informal and a formal model for a cloaking query processor. We have also provided an architectural reference model for a cloaking query processor, related it to the formal model, and showed how a query processor can implement cloaking without altering its input data. The formal model and the architectural model are independent of applications and data models.

We have illustrated how cloaking improves query expression and execution in both bi-level and non-bi-level query settings. We have also reviewed representative systems from three different related areas and compared these systems to our own.

The models for cloaking presented in this chapter helps achieve our goals of efficient query execution (Goal G3 in Section 5.3.1), ease of query expression (G5), SI-only-query preservation (G6), and compatibility with existing query languages (G7). Chapter 9 describes an XML bi-level query processor that employs cloaking to achieve these goals.

9. Querying XML Bi-level Information

In this chapter, we describe how bi-level queries over Sixml documents can be processed using existing query processors and query languages. Specifically, we introduce the *bi-level navigator,* an alternative XML path navigator designed to support bi-level querying [120].

This chapter brings together the various components and concepts described in Chapters 4 through 8 to realize our strategy to meet the seven goals we set for transforming bi-level information. (Section 5.3 outlines the goals and the strategy.)

We begin the chapter with an overview of XML querying. We then present a data model for Sixml documents to support bi-level querying and discuss the details of the custom bi-level navigator. We also give an overview of our implementation of the bi-level navigator and share the results of an experimental evaluation.

9.1. Overview of XML Querying

In this section, we provide an overview of the popular XML query languages XPath [166] and XSLT [177], including an overview of data navigation.

9.1.1. Overview of XPath

We begin with an overview of the XPath data model, the different parts of an XPath expression, and the process of evaluating an XPath expression. This discussion focuses on XPath 1.0 [166], but much of it also applies to XPath 2.0 [165].

9.1.1.1. The XPath Data Model

XPath represents an XML document as an ordered tree similar to the Document Object Model (DOM) [34], but unlike DOM, XPath does not include an application-programming interface (API). Consequently, XPath evaluators typically do not represent an XML document in the XPath data model. Instead, they internally represent the document as a DOM tree and evaluate path expressions over the DOM tree.

XPath defines seven kinds of nodes. Four of these kinds—*element*, *attribute*, *comment*, *processing instruction*—are functionally equivalent to DOM node types with the same name. (Section 7.2 gives an overview of DOM.) The fifth kind—*text*—includes both text nodes and CData section nodes of DOM. The sixth kind, called *root*, corresponds to the node-type *document* in DOM. The seventh kind, called *namespace*, is not available in DOM. (Namespace information is available as node properties in DOM Level 2 Core [36].)

The following types of DOM nodes have no equivalent in XPath: document fragment, document type, entity, entity reference, and notation.

In XPath, only the root node and element nodes may have child nodes. An attribute node is not a child of its owner element, but, for querying purpose, an element may be treated as the parent of its attributes. The same is true for namespace nodes. All nodes except the root node may have siblings. An attribute node may have only an attribute as a sibling; a namespace may have only a namespace as a sibling; and a text node may *not* have another text node as an immediate sibling.

Table 9.1: Kinds of XPath nodes, and kinds of their children, siblings, and parent. The acronym PI denotes a processing instruction

Node type	Kinds of children	Kinds of siblings	Kinds of parent
Element	Element, Comment, PI, Text	Element, Comment, PI, Text	Element, Root
Attribute	None	Attribute	Element (only for querying)
Comment	None	Element, Comment, PI, Text	Element, Root
Processing instruction (PI)	None	Element, Comment, PI, Text	Element, Root
Text	None	Element, Comment, PI	Element
Root	Element, Comment, PI	None	None
Namespace	None	Namespace	Element (only for querying)

Table 9.1 shows the different kinds of XPath nodes. For each kind of node, the table also shows the kinds of children and siblings a node of that kind may have. It also shows the kinds of nodes that could be the parent.

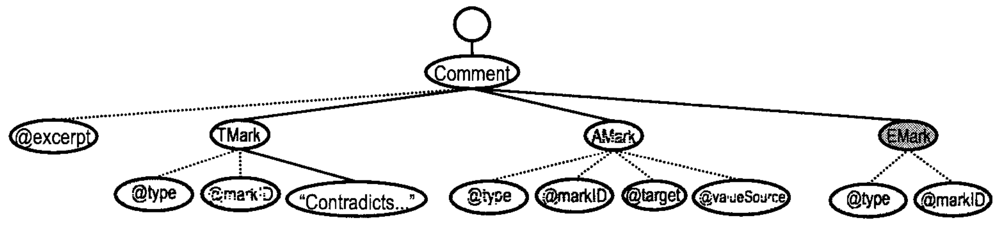


Figure 9.1: XPath representation of an XML document. The representation for the Sixml document of Figure 7.1 is shown. The unlabeled node is the root node

Figure 9.1 shows the Sixml document of Figure 7.1 represented as a tree in the XPath data model. This tree is similar to the DOM tree of Figure 7.3, except that attribute nodes do not use separate text nodes to represent values. The symbol @ denotes an attribute node; text enclosed in quotes indicates text nodes. A solid edge indicates a parent-child relationship; a dotted edge shows an attribute's relationship to its element.

9.1.1.2. XPath Expressions

An XPath *expression* (also called a *location path*) selects parts of a tree represented in the XPath data model. It is always evaluated in the context of a node. A path expression consists of one or more steps separated by the delimiter `/`. A *step* defines the criteria to choose nodes that are reachable from a given node. The first step indicates a sequence of nodes reachable from the expression's context node; the second step indicates the combined sequence of nodes reachable from the result nodes of the first step, and so on. For example, the expression `TMark/*` has two steps. When evaluated in the context of the element Comment in Figure 9.1, the first step (`TMark`) selects all elements named TMark. The next step (`*`) selects all child elements of each node selected in the first step. (XPath 1.0 uses the term *node set* to describe the collection of nodes a step selects, but the collection is actually a node sequence. We, and XPath 2.0, use the term *node sequence* for accuracy.)

The character / placed at the beginning of an expression indicates a step by itself. This step selects the root node. An expression beginning with the character / is an *absolute expression* (or an *absolute path*); all other expressions are *relative expression* (or *relative paths*).

A step in an expression consists of three parts: an axis, a node test, and an optional sequence of predicates. An *axis* indicates a sequence of nodes reachable in a particular "direction" from a node. XPath defines 13 axes such as child, attribute, and descendant. The *child* axis of a node includes all its child nodes. The *attribute* axis of a node (applicable only to an element node) includes all the attributes of the node. The *descendant* axis of a node includes all its child nodes, child nodes of child nodes, and so on. The child axis and the descendant axis do not include attributes. For example, in Figure 9.1, the attribute excerpt is not on the child axis (from element Comment), and markID is not on the descendant axis.

An axis in a step is indicated by writing the name of the axis followed by a pair of colons. For example, `child::`, `attribute::`, and `descendant::` indicate the child, attribute, and descendant axes, respectively. Names of some axes have abbreviations. The delimiting colons are omitted when an abbreviated name is used. The abbreviation for the child axis is an empty string (thus the child axis is the default axis); the character `@` is the abbreviation for the attribute axis. The character / is the abbreviation for the descendant axis. Within an expression, the string `//` indicates the descendant axis: The first `/` is the step separator, the second `/` is the axis abbreviation. The string `//` at the beginning of an expression selects all descendant nodes of the root node.

The second part of a step, the *node test*, restricts the nodes in the selected axis based on node kinds and node names. For example, one can choose only element nodes or only text nodes. One can also choose nodes with a specific name or choose all nodes by specifying the wildcard character `*`.

Here are some XPath expressions and their results when the element Comment in the tree in Figure 9.1 is the context node:

- `TMark` selects the element TMark.

- `@excerpt` selects the attribute excerpt.
- `*` selects all contained elements.
- . (a period by itself) selects the context node.
- `.//text()` selects all descendant text.
- `.//*` selects all elements that descend from Comment, but not Comment itself.
- `.//@*` selects all attributes of Comment and the attributes of its descendant elements.

Here are some XPath expressions and their results when the element TMark is the context node: `@excerpt` selects no nodes because TMark has no attribute named excerpt; `/Comment/@excerpt` selects the attribute excerpt of Comment (because this expression is absolute); `//text()` selects all text nodes that descend from the root node; and `//@*` selects all (nine) attributes in the tree.

The third part of a step, the optional sequence of predicates, further restricts the nodes that pass the node test. A *predicate* is a Boolean expression enclosed in brackets. For example, the following expressions over the tree in Figure 9.1 use predicates: `/Comment/*[position()=1]` selects the element TMark contained in the element Comment (because it is the first child in the sequence of children of Comment); `/Comment/*[@target]` selects all child elements of Comment that possess an attribute named `target`. In this case, this expression selects the lone AMark element in the document. (The function `position` is built-in. It returns the position of a node in a sequence.)

The XPath syntax allows the use of an axis with a node even when that axis is known to be empty for that (kind of) node. For example, the expression `//@excerpt/*` to return all elements in the child axis of attribute excerpt is valid, even though an attribute does not have child nodes. (We take advantage of such expressions to facilitate navigation from an attribute to its mark associations. See Section 9.2.)

9.1.1.3. Evaluating XPath Expressions

Evaluating an XPath expression involves sequentially evaluating the steps in the expression. The pseudo-code in Figure 9.2 provides an overview of the conceptual procedure to evaluate a step. In this procedure, the input node sequence for a step is the node sequence resulting from the previous step; for the first step, the input node sequence consists of only the expression's context node. The result node sequence from the final step is the result of evaluating the expression.

Figure 9.2 illustrates three key parts of the procedure to evaluate an XPath step: Determining the nodes that lie on an axis; performing node tests and evaluating predicates; and navigating the tree nodes. The code for navigating tree nodes is shown in large text.

Figure 9.3 shows a high-level class diagram (drawn using the syntax defined in UML, the Unified Modeling Language [159]) depicting the association between an XPath evaluator and a context node. (This specific depiction is ours, but the XML query-processor development community widely uses the approach depicted, in one form or another.) The class XPathEvaluator is responsible for evaluating XPath expressions. It uses XPathNavigator to navigate the nodes related to a node. Section 9.1.2 describes XSLTProcessor and its relationship with XPathEvaluator.

```
for each node in the input node sequence
  if (axis is "child")
  {
    child = first child of the input node
    while (child)
    {
      if child passes node test and predicates, add it to the result sequence
      child = next child of the input node
    }
  }
  else if (axis is "attribute") {
    attribute = first attribute of the input node
    while (attribute)
    {
      if attribute passes node test and predicates, add it to the result sequence
      attribute = next attribute of the input node
    }
  }
  … //handle other axes
```

Figure 9.2: Pseudo-code outlining the procedure to evaluate a step in an XPath expression. Lines in the larger font size contain code to navigate among tree nodes

Separating navigation from evaluation as shown in Figure 9.3 makes it possible to employ a custom *navigator* (that is, an implementation of the navigation routines) that can report nodes not in the tree and skip nodes in the tree. We use the ability to report non-existing nodes to support navigation in the nested schema even though the input document is in the normalized schema. We use the ability to skip existing nodes to support cloaking. Thus, custom navigation is a key part of our strategy to meet our goals for bi-level querying. (Section 5.2 describes the normalized and nested schemas for Sixml data. Section 5.3 describes our goals and strategy for bi-level querying.)

Because navigation is an important part of our approach to bi-level querying, we provide a brief overview of navigation among nodes in an XPath tree.

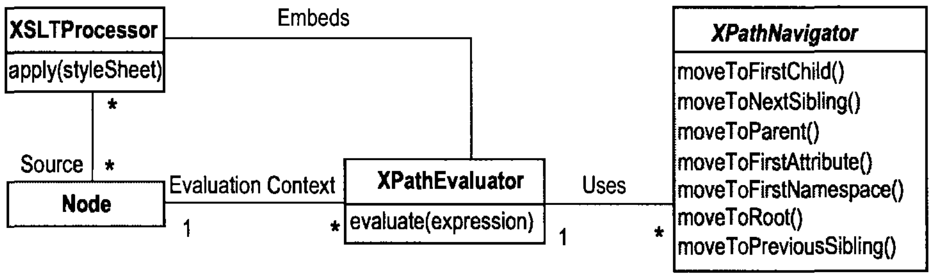


Figure 9.3: Overview of XPath and XSLT processing

Table 9.2 summarizes the different kinds of movements possible among XPath nodes. Conceptually, an XPath navigator (or, just *navigator*) needs to support *five basic movements* from any node: first child, next sibling, parent, first attribute, and first namespace. Any other movement can be implemented using a combination of these five movements.

Table 9.2: Possible movements among XPath nodes

Current node type	Move to root?	Move to child?	Move to sibling?	Move to parent?	Move to attribute?	Move to namespace?
Element	Yes	Yes	Yes	Yes	Yes	Yes
Attribute	Yes	No	Yes	Yes	No	No
Comment	Yes	No	Yes	Yes	No	No
Processing Instruction	Yes	No	Yes	Yes	No	No
Text	Yes	No	Yes	Yes	No	No
Root	Yes	Yes	No	No	No	No
Namespace	Yes	No	Yes	Yes	No	No

Some navigators also explicitly support two other movements—to the root and to the previous sibling—though these movements are not really needed. The seven methods in the class XPathNavigator of Figure 9.3 indicate the five basic movements and movement to root and previous sibling.

A navigator might support other kinds of movements (beyond the aforementioned seven movements). For example, a navigator might support direct movement to an attribute of a particular name. This movement can be provided by moving to the first attribute and traversing the attribute sequence until the required attribute is seen, but the movement can be more efficient if the attributes are organized as a hash table with the attribute name as the key.

9.1.2. Overview of XSLT

Both XSLT and XQuery [176] provide ways to manipulate parts of an XML document already selected using embedded XPath expressions. In this section, we provide an overview of XSLT and show how XPath expressions are employed in XSLT queries. For simplicity, we limit this discussion to XSLT 1.0, but the discussion, in general, applies to XSLT 2.0 as well.

XSLT is a rule-based language to transform an XML document to other forms (including non-XML forms, but we focus on XML). In the process, parts of the input document can be filtered out and new data can be added. XSLT represents the XML document to be transformed as a tree in the XPath data model.

An XSLT *transformation* (that is, a query) is expressed in a *style sheet* (which is itself an XML document) consisting of a series of templates. A *template* defines a rule that is triggered based on the node being processed. A template uses a pattern expression to describe the nodes to which it applies. A *pattern expression* tests features such as type and name of a node.

An XSLT processor operates on a sequence of nodes. It traverses the tree induced by each input node in *document order* (that is, the order in which the nodes are serialized), and matches each unprocessed node in the current tree against template patterns. (The processor uses a complex priority scheme to search templates.) With the current node as the context node, the processor triggers a matching template if found, or a default template if no matching template is found. In most cases, the default template simply outputs the value of its context node.

The foregoing description of the process of triggering templates is overly simplified, but it suffices for our purpose. Kay [80, 81] describes the process in detail.

Figure 9.4(a) shows an XSLT style sheet with five templates to output mark-association elements and to filter out other elements from a Sixml document. The first template matches only the root node. The second through fourth templates match any element with the name AMark, EMark, and TMark, respectively. The last template matches any element that is not matched by another template. The comments in the style sheet (shown in gray background) provide helpful information about the templates. Figure 9.4(b) shows the result of transforming the document represented in Figure 9.1.

The first template begins an output element named Marks. It then asks the XSLT processor, using the apply-templates instruction, to trigger a template for each element that descends from the root node (using the XPath expression //* as the value of the attribute select). The XSLT processor uses an XPath evaluator to evaluate the expression //*. In response, the XPath evaluator returns all elements in the document. Then, for each element returned, the XSLT processor finds a matching template, makes the element the context node for the template, and triggers the template.

For the document tree in Figure 9.1, the first template selects and processes the following elements (in the order shown) to produce the result shown in Figure 9.4(b): Comment, TMark, AMark, and EMark. We now discuss how each of these elements is processed.

The element Comment matches the last template, which outputs nothing. That is, this element is filtered out.

The element TMark matches the fourth template. This template outputs an element named TextMark and writes out the text with which the TMark element is associated. The template uses the instruction value-of to output text content. This instruction accepts an XPath expression, evaluates the expression using an XPath evaluator, and outputs the string version of the result.

The element EMark matches the third template. This template outputs an element named ElementMark and writes the name of the element that is the parent of the EMark element as the text of the output element. That is, the output will identify the element with which the input EMark element associates a mark.

The element AMark matches the second template. This template outputs an element named AttributeMark and writes the value of the attribute target (of the context node) as the text of the output element. That is, the output will identify the attribute with which the input AMark element associates a mark.

```
<?xml version="1.0"?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
 <xsl:template match="/"> <!--1. Template for the root node -->
  <Marks>
    <xsl:apply-templates select="//*"/> <!--Trigger a template for each element in the tree-->
  </Marks>
 </xsl:template>
<xsl:template match="AMark"> <!--2. Template for an AMark element -->
  <AttributeMark>
    <xsl:value-of select="@target"/> <!--Output name of attribute with which mark is associated-->
  </AttributeMark>
 </xsl:template>
 <xsl:template match="EMark"> <!--3. Template for an EMark element -->
  <ElementMark>
    <xsl:value-of select="name(..)"/><!--Output name of element with which mark is associated-->
  </ElementMark>
 </xsl:template>
<xsl:template match="TMark"> <!--4. Template for a TMark element. Assume uni-mark -->
  <TextMark>
    <xsl:value-of select="."/> <!--Output the text with which mark is associated-->
  </TextMark>
 </xsl:template>
 <xsl:template match="*"/> <!--5. Template for other elements: does nothing -->
</xsl:stylesheet> <!--End of style sheet-->
```

(a)

```
<?xml version="1.0"?>
<Marks>
  <TextMark>Contradicts...</TextMark>
  <AttributeMark>excerpt</AttributeMark>
  <ElementMark>Comment</ElementMark>
</Marks>
```

(b)

Figure 9.4: An example of transforming an XML document using XSLT. (a) A style sheet with templates to transform mark-association elements and filter out other elements. The start of each template is shown in bold for ease of reading; (b) The result of transforming the document tree in Figure 9.1 using the style sheet shown in Part (a) of this figure

As illustrated in the example style sheet, an XSLT processor uses an XPath evaluator to select parts of the input document. Thus, the navigator used with the XPath evaluator also determines the nodes exposed to the XSLT processor.

9.2. Representing Sixml Data

We now describe how a Sixml document is represented using an extension to the XPath data model. We first enumerate some alternative approaches and then describe our approach. Also, we discuss representing a Sixml document in the normalized schema before discussing representation in the nested schema.

One approach, *Alternative 1*, to representing a Sixml document is to use the XPath data model *as is*. For example, the Sixml document of Figure 7.1 would be represented as the tree in Figure 9.1. In this approach, the XPath expression `./sixml:EMark` navigates from an element to its mark associations, but navigation from an attribute to mark associations requires the expression `./sixml:AMark[@target=$name]`, where `$name` is a variable bound to the name of the target attribute. However, creating variables and variable bindings requires the use of languages such as XSLT and XQuery, making query expression harder. (The expression `./AMark[@target=name()]` cannot return the mark associations for the attribute because, in this expression, the function `name` returns `'AMark'`, not the attribute's name.)

Alternative 1 requires knowledge of the mark-association schema (for example, one would need to know mark association names), but it does not require any changes to the XPath data model or the query language.

Alternative 2 is to emulate Sixml DOM—remove the mark associations from the child axis and attach them to their respective target nodes—and introduce a new axis called *marks* to navigate from a node to its marks. (Section 7.4.2.1 describes how Sixml DOM represents mark associations.) The new axis would be needed because the mark associations would not be visible after removing them from the child axis. In this approach, the expression `marks::*` returns the marks associated with the current node, regardless of the node type.

Alternative 2 does not require knowledge of the mark-association schema, but it requires extensions to both the XPath data model and the query language.

Alternative 3, our approach, is to emulate Sixml DOM, but make a mark association a *child* of its target node, and access mark associations using the existing child axis. Figure 9.5 represents the tree of Figure 9.1 in this alternative. This tree is similar to the Sixml DOM tree in Figure 7.5, except that a mark-association element is represented as a child of its target node (as indicated by a solid edge).

Alternative 3 extends the XPath data model (because nodes such as attributes would now need to accommodate children), but not the query language. This approach exploits an allowance in the XPath expression syntax that allows following an axis from a node, even if that axis does not apply to that node. For example, following the child axis using the expression `child::*` (or just `*`) in the context of attribute excerpt ordinarily returns an empty node sequence (because an attribute does not have children), but a custom navigator can return mark associations if the XPath evaluator can handle such results.

Alternative 3 allows the navigator to reuse Sixml DOM because the extended data model is similar to the Sixml DOM representation. As will be illustrated in Section 9.3, this reuse simplifies the design and implementation of the navigator.

Selecting the mark associations of an element using the child axis requires a predicate or a name test because the element may have other child elements as well. For example, the expressions `/Comment/*[@sixml:type]` and `/Comment/sixml:*` return only mark associations; the expression `/Comment/*[not(@sixml:type)]` returns child elements that are not mark associations. (Section 7.4.2.2 discusses the use of the attribute sixml:type and the namespace sixml with mark-association elements.)

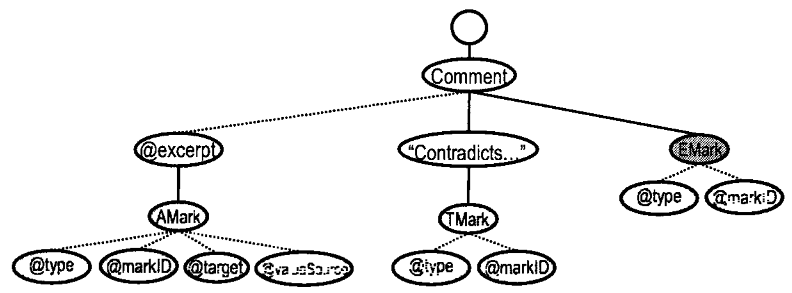


Figure 9.5: Representation of a Sixml document in the normalized schema using the extended XPath data model. A mark association is represented as a child of its target node. SI is colored white. Mark association information is colored gray

We now discuss representing a Sixml document in the nested schema. We represent a Sixml document in the nested schema by adding two child elements to each mark association: sixml:Descriptor representing the mark descriptor, and sixml:Context representing the context information retrieved from the mark using the descriptor.

Figure 9.6 shows the tree of Figure 9.5 represented in the nested schema. Details of namespaces, mark descriptors, and context information are omitted for simplicity.

Representing mark descriptor and context information as children of a mark association differs from Sixml DOM (in which a mark association has no children), but it eases query expression because the child axis can be used to select the details of a mark association. For example, the following expression selects the descriptor for each mark associated with the attribute excerpt:

```
/Comment/@excerpt/*/sixml:Descriptor
```

Here, the wildcard * selects all children of excerpt, which, in the extended model, selects its mark associations. The last step selects the mark descriptor for each mark association.

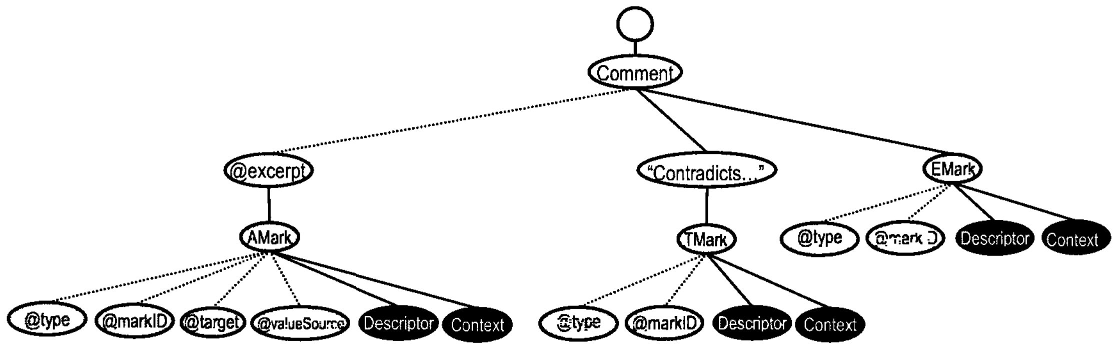


Figure 9.6: Representation of a Sixml document in the nested schema using the extended XPath data model. This representation is obtained from the tree in Figure 9.5 by adding child elements for descriptor and context information to each mark-association element. Details of mark descriptor and context information are omitted for brevity

Representing the descriptor and context information as child elements does not imply that these information parts are *eagerly* added to the tree. Instead, a custom navigator can just report the presence of these elements even if they are not present in the input document, and construct the elements *on demand* if the XPath evaluator accesses them. Our bi-level navigator implementation described in Section 9.4.1 follows this strategy.

9.3. Processing Bi-level Queries

In this section, we give an overview of a bi-level query processor, and discuss in detail the design of the *Sixml Navigator*, a custom XPath navigator for Sixml data. For ease of presentation, we call this navigator the *bi-level navigator.*

9.3.1. Overview of a Bi-level Query Processor

The bi-level navigator realizes the following strategies outlined in Section 5.3.2 for bi-level querying:

- Presenting a Sixml document in the nested schema for querying when the document is in the normalized schema.
- Retrieving mark descriptors and context information on demand.
- Cloaking mark associations to preserve SI-only queries and three other classes of queries identified in Section 8.1.2.

Figure 9.7 shows the architecture of a bi-level query processor as a UML class diagram. The classes XPathEvaluator and XSLTProcessor are *traditional query processors* (that is, existing processors used to query XML data). These processors, *unchanged,* become bi-level query processors by virtue of using the bi-level navigator implemented in SixmlNavigator.

The class SixmlNode is similar to the Sixml DOM class SixmlNode (in Figure 7.2), except here, SixmlNode represents an XPath node. Alternatively, the Sixml DOM class SixmlNode may be used directly to simplify the bi-level navigator. (Our bi-level navigator implementation takes this approach.)

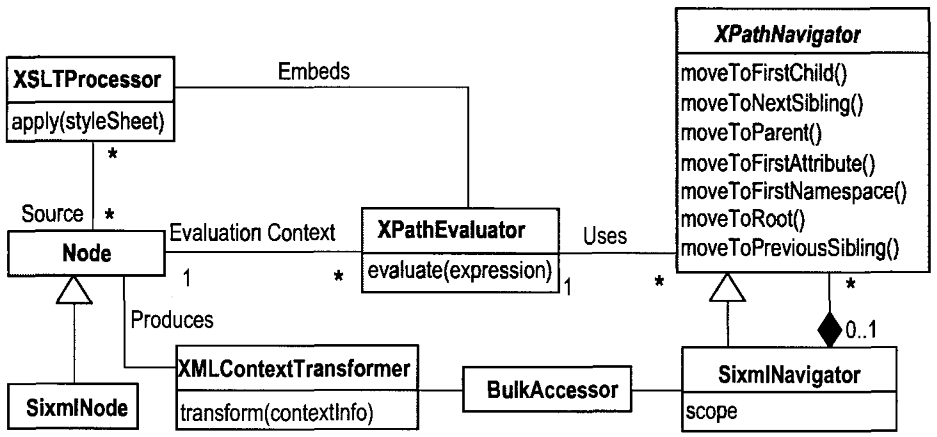


Figure 9.7: The architecture of a bi-level query processor. XPathEvaluator and XSLTProcessor are traditional query processors using the bi-level navigator implemented in SixmlNavigator

The class BulkAccessor represents the bulk accessor component described in Section 6.3. XMLContextTransformer transforms context information retrieved from a context agent into XML. These classes are not needed here if the navigator uses Sixml DOM to represent the input document (because Sixml DOM already uses BulkAccessor and XMLContextTransformer).

The class SixmlNavigator extends XPathNavigator (henceforth called the *traditional navigator*) to support bi-level navigation. This bi-level navigator functions as the traditional navigator in the context of a non-Sixml node and in the context of a Sixml node that is not associated with marks.

In the rest of this section, we describe the design of the bi-level navigator and show the mechanics of bi-level navigation. We also discuss how the navigator cloaks selected information. Chapter 8 introduced cloaking.

9.3.2. Navigator State and Scope

The bi-level navigator is a state machine with four possible states: *SI*, *Association*, *Descriptor*, and *Context*. A state indicates the kind of information to which the navigator currently points.

The navigator's state transitions are governed by its movements. Table 9.3 shows the possible state transitions due to navigator movements. In this table, movement to a child means movement to first child; movement to sibling means movement either to the next sibling or to the previous sibling. The entries with asterisk indicate movements permitted only in the bi-level navigator, and are due to the use of the child axis to navigate to mark associations.

The salient movements and state transitions are: Movement from any node to the root node causes a transition to the state SI. Movement from an element to its first child, a sibling, or the parent can change the state. Movement to a sibling from an element, text node, comment, or PI can cause the state to change from SI to Association.

The state transitions of the navigator are influenced by the *scope* of the navigator which determines the kind of information the navigator reveals to the query processor. The scope is a value from the ordered set {*SI*, *Association*, *Descriptor*, *Context*}. When the scope is SI, the navigator reveals only SI to the query processor; when the scope is Association, the navigator reveals SI and mark association information, but cloaks descriptor and context information; and so on.

Table 9.3: Possible state transitions of the bi-level navigator due to movements among XPath nodes. The letters S, A, D and C correspond to the states SI, Association, Descriptor, and Context, respectively. X means any state. N/A means not applicable. A dash indicates no transition. Entries with asterisk denote deviation from the XPath model

Current node type	Move to Root	Move to Child	Move to Sibling	Move to Parent	Move to Attribute	Move to Namespace
Element	X→S	S→A A→D	S→A D→C	A→S D→A C→A	-	-
Attribute	X→S	S→A*	-	-	-	N/A
Comment	X→S	S→A*	S→A	-	N/A	N/A
Processing Instruction	X→S	S→A*	S→A	-	N/A	N/A
Text	X→S	S→A*	S→A	-	N/A	N/A
Root	X→S	-	N/A	N/A	N/A	N/A
Namespace	X→S	N/A	-	-	N/A	-

The query processor sets the navigator scope implicitly as follows, based on the kind of information the path expression references:

1. The scope is Association if the expression references an element in the namespace sixml or if the query examines the attribute sixml:type. For example, the scope of the expression `/Comment/*[@sixml:type]` is Association, whereas the scope of the expression `/Comment/*` is SI.

2. The scope is Descriptor if the expression references the child element Descriptor of a mark association. For example, the scope of the expression `/Comment/sixml:EMark/sixml:Descriptor` is Descriptor, but the scope of the expression `/Comment/*/sixml:Descriptor` is SI.

3. The scope is Context if the expression references the child element Context of a mark association. The expression `/Comment/sixml:EMark/sixml:Context` has Context scope, but the scope of the expression `/Comment/*/Context` is SI.
4. The scope is SI if the expression does not satisfy any of the three aforementioned conditions.

If the expression satisfies more than one condition, the navigator scope is set to the most permissive value. For example, if the first three conditions are met, the scope is set to Context.

The user may explicitly set the navigator scope to any of the four possible values, regardless of the kind of information the query expression references.

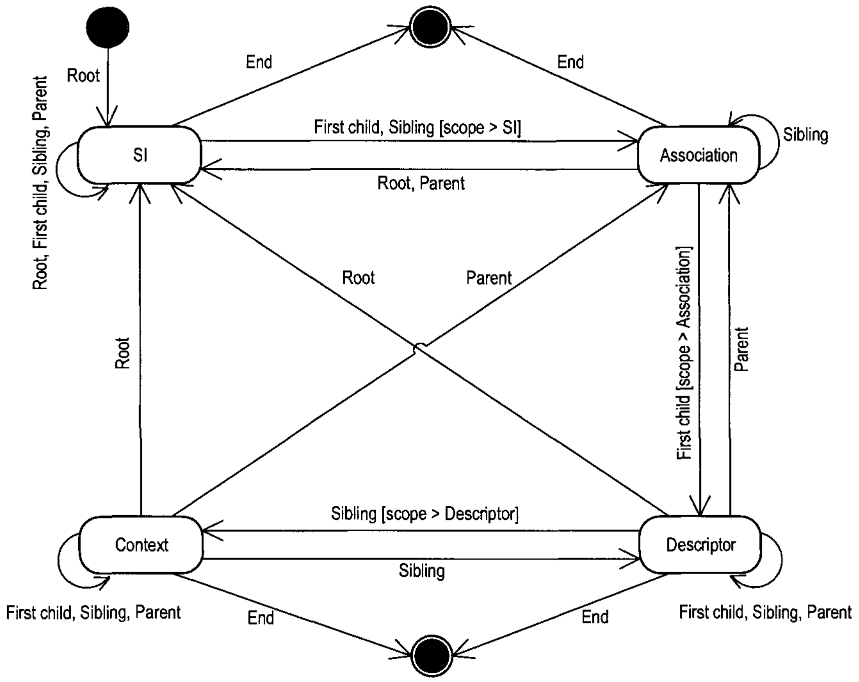

Root
End
End
Root, First child, Sibling, Parent
SI
First child, Sibling [scope > SI]
Association
Sibling
Root, Parent
Root
Parent
Root
First child [scope > Association]
Parent
Sibling [scope > Descriptor]
Context
Descriptor
Sibling
First child, Sibling, Parent
End
End
First child, Sibling, Parent

Figure 9.8: The bi-level navigator state diagram. Labels attached to arrows show movements that cause state transitions. Text in brackets denotes the condition for a transition

Figure 9.8 shows a UML state diagram that describes the navigator states, the movements that trigger state transitions, and the conditions for transitions, taking into account the navigator scope. The ovals represent states and the arrows represent state transitions. The single dark circle denotes the start state. A dark circle with a surrounding unfilled circle denotes a final state. The label associated with a transition denotes the kind of navigator movement that triggers the transition. The text in brackets denotes the condition under which the trigger (that is, the movement) causes the transition. The trigger labeled 'End' indicates the end of the use of the navigator. The navigator has no movement such as "move to end' that corresponds to this trigger, but the trigger is used in the figure to comply with the UML syntax: In UML, a state transition is immediate if no trigger or condition is attached to the transition.

9.3.3. Navigating Bi-level Information

We now describe how the bi-level navigator navigates a Sixml document.

9.3.3.1. Overview

Conceptually, a bi-level navigator uses *four* traditional navigators, one per state. The navigators are called *S-navigator*, *A-navigator*, *D-navigator*, and *C-navigator*. The bi-level navigator receives navigation requests from the query processor and employs an internal traditional navigator based on the current state and the movement requested.

Figures 9.9 and 9.10 show the procedures (in pseudo-code) to move the bi-level navigator from a node to its first child and to its next sibling, respectively. Tables 9.4 and 9.5 list the movements made when evaluating example expressions. (The caption for each table includes the example expression evaluated.) In the procedures, text with gray background indicates comments. End-of-line comments in bold provide example results of the internal movements. A number in parentheses (such as `9.4-1`) indicates the table number and the row number that correspond to the example movement. For brevity, not all movements are identified in the figure.

We illustrate the procedures using the expression `sixml:EMark/sixml:Context` to retrieve the context information for the marks associated with the element `Comment`. We assume that the bi-level navigator is currently positioned on `Comment`, so the navigator's current state is SI. We also assume that the navigator's scope is set to Context. Table 9.4 shows the four movements needed to reach context information. The movements can be traced on the tree in Figure 9.6.

First, the query processor asks the bi-level navigator to move to the first child of the current node (that is, of the element `Comment`). The bi-level navigator in turn asks the S-navigator to move to the first child. The S-navigator moves successfully to the text node. The bi-level navigator remains in the state SI.

The second movement asks the bi-level navigator to move to the next sibling of the text node. The bi-level navigator in turn asks the S-navigator to move to the next sibling. The S-navigator fails because the text node has no SI siblings. As a result, the bi-level navigator initializes the A-navigator with the mark associations for `Comment` (if the A-navigator is not yet initialized with that information) and then asks the A-navigator to move to the first mark-association element. It then changes its state from SI to Association, and presents the first mark-association element as a sibling of the text node to the query processor.

In the third movement, the query processor asks the bi-level navigator to move to the first child of the current mark-association element. In response, the bi-level navigator retrieves the descriptor for the current mark association (if the descriptor has not been retrieved before), initializes the D-navigator, asks the D-navigator to move to the element Descriptor. The bi-level navigator then changes its state from Association to Descriptor.

In the fourth movement, the query processor asks the bi-level navigator to move to the sibling of the element Descriptor. In response, the bi-level navigator retrieves the context information (if it has not been retrieved before) using the descriptor for the current mark association and initializes the C-navigator with the retrieved context information. It then asks the C-navigator to move to the element Context, and changes state from Descriptor to Context.

9.3.3.2. Selecting Multiple Nodes in a Step

The navigation overview thus far assumes only one node passes the node test at each step, but, in reality, several nodes may pass the test at each step. For example, several EMark elements may be associated with Comment.

```
boolean moveToFirstChild()
{
  if (state == SI)
  {
    //SI before mark associations
    if (sNavigator.moveToFirstChild())return true; //Comment→text (9.4-1)
    else if (scope > SI)
    {
      //no SI child, first mark association of current SI node is the first child
      aNavigator.load(currentNode.MarkAssociations);
      aNavigator.moveToRoot();
      aNavigator.moveToFirstChild(); //@excerpt→AMark (9.5-2)
      state = Association; return true;
    }
    else return false;
  }
  else if (state == Association and scope > Association)
  {
      //first child of a mark association is always the mark descriptor
      dNavigator.load(currentMarkAssociation.Descriptor);
      dNavigator.moveToRoot();
      dNavigator.moveToFirstChild();  //EMark→Descriptor (9.4-4)
      state = Descriptor; return true;
  }
  else if (state == Descriptor)
      //move to the descriptor's first child
      return dNavigator.moveToFirstChild();
  else if (state == Context)
      //move to the context's first child
      return cNavigator.moveToFirstChild();
}
```

Figure 9.9: Pseudo-code outlining movement to the first child of the current XPath node

Table 9.4: Movements to retrieve context information. Context information from marks associated with the element Comment is retrieved using the expression `sixml:EMark/sixml:Context`. Comment is the "current node" at the beginning

Movement to	Result node	Result state	Remarks
1. First child	Text node	SI	Invoke S-navigator
2. Next sibling	sixml:EMark	Association	S-navigator fails; load and invoke A-navigator.
3. First child	sixml:Descriptor	Descriptor	Load and invoke D-navigator.
4. Next sibling	sixml:Context	Context	D-navigator fails; load and invoke C-navigator.

```
boolean moveToNextSibling()
{
  if (state == SI)
  {
    //SI is before mark associations
    if (sNavigator.moveToNextSibling())return true;
    //next sibling of an SI attribute or a namespace is also SI.
    else if (sNavigator.CurrentNode is an attribute or a namespace)
      return false;
    else if (scope > SI)
    {
      //no more SI, the first mark association of current node is the next sibling
      aNavigator.load(currentNode.MarkAssociations);
      aNavigator.moveToRoot(); aNavigator.moveToFirstChild(); //text ›EMark (9.4-2)
      state = Associations; return true;
    }
    else return false;
  }
  else if (state == Association)
  {
    return aNavigator.moveToNextSibling(); //returns "false" for running example
  }
  else if (state == Descriptor)
      if (dNavigator.CurrentNode is not Descriptor) //somewhere inside descriptor
        return dNavigator.moveToNextSibling();
      else if (scope > Descriptor)
     {
        //context information follows descriptor
        cNavigator.load(currentMarkAssociation.Context); cNavigator.moveToRoot();
        cNavigator.moveToFirstChild(); //Descriptor ›Context (9.4-4)
        state = Context; return true;
      }
      else return false;
  else if (state == Context)
      if (cNavigator.CurrentNode is Context)
        //a mark association has only two children
        return false;
      else //somewhere inside context
        return cNavigator.moveToNextSibling();
}
```

Figure 9.10: Pseudo-code outlining movement to the next sibling of the current XPath node

Table 9.5: Movements to retrieve a mark descriptor. The descriptor for each mark associated with the attribute excerpt is retrieved using the expression `@excerpt/*/sixml:Descriptor`. Comment is the "current node" at the beginning

Movement to	Result node	Result state	Remarks
1. First attribute	@excerpt	SI	Invoke S-navigator. Movement code is omitted.
2. First child	sixml:AMark	Association	S-navigator fails, load and invoke A-navigator.
3. First child	sixml:Descriptor	Descriptor	Load and invoke D-navigator.

To select multiple nodes at a step, when a node passes a test, the query processor saves the navigator state so it can test other nodes later. For example, in Table 9.4, the second movement finds a node that passes the test for an element node with the name EMark. At this stage, the query processor saves the navigator state, and then proceeds with the other movements shown. After performing movements 3 and 4 to retrieve context information for the EMark element just found, the query processor restores the saved navigator state, and moves the navigator to the next node to see if another node passes the original test. It then repeats movements 3 and 4 for each EMark child element it finds for Comment.

The query processor decides when to save and restore the navigator state. It also decides which movements to perform and when. The navigator is responsible for saving and restoring the state when the processor instructs it. It is also responsible for carrying out the movements the processor directs it to perform. Due to this separation of concerns, the query processor does not need to directly examine the navigator state, and the navigator can be independent of the overall query-processing strategy. Consequently, the design of the query processor and the navigator is simple, and different navigators can be used with the same query processor.

9.3.3.3. Retrieving Information on Demand

Our representation of Sixml data and the design of the bi-level navigator together realize our strategy of retrieving mark descriptors and context information *on demand* to improve query-execution efficiency. (Section 5.3.2 outlines our strategy for bi-level querying.)

The procedures in Figures 9.9 and 9.10 illustrate the opportunity to retrieve information on demand: The code in Figure 9.9 loads descriptor information only when the query processor needs to examine the descriptor for a mark association. Similarly, the code in Figure 9.10 loads context information on demand. Movements 3 and 4 in Table 9.4 exemplify on demand retrieval.

Loading descriptor and context information on demand can significantly improve the performance of queries, especially for queries that do not need context information, because obtaining context information requires interacting with the base layer. An implementation of the bi-level navigator can further improve query-execution efficiency by caching the descriptor and context information for a mark, and reusing the cached information if the same mark is associated with another node.

Our implementation of the bi-level navigator retrieves mark descriptors and context information on demand, and caches both kinds of information. Section 9.4.1 gives an overview of the implementation. Section 9.4.3 provides experimental verification of the savings due to our strategy.

9.3.4. Cloaking Information

We now discuss how the bi-level navigator cloaks (that is, hides) information from the query processor so that certain classes of queries, such as SI-only queries, can be expressed more easily and be executed more efficiently. Section 8.1 introduced cloaking and its application to bi-level querying.

In terms of the formal model for cloaking described in Section 8.2.1, the bi-level navigator employs the following cloaking configuration (which is also the configuration used in Figure 8.2):

- a totally ordered set {*White*, *Gray*, *Slate*, *Black*}, where *Black* > *Slate* > *Gray* > *White*;
- a cloaking scheme in which, for each node *n*, *Color(n)* ≥ *Color(Parent(n))*; and
- a test function *Color(query)* ≥ *Color(node)*.

The navigator uses the following *implicit* cloaking scheme. The tree in Figure 9.6 is colored using this scheme:

- A mark-association element is colored *Gray*.
- Ancestors of a mark-association element are colored *White*.
- The child element `Descriptor` of a mark-association element and the descendants of `Descriptor` are colored *Slate*.
- The child element `Context` of a mark-association element and the descendants of `Context` are colored *Black*.
- The color of an attribute is the same as its owner element. The same is true for a namespace node.

In terms of the design of the bi-level navigator, the state of the navigator corresponds to the color of the current node, and the scope of the navigator corresponds to the query color. Further, the set {*SI*, *Association*, *Descriptor*, *Context*} used to indicate the navigator scope is order isomorphic with the set {*White*, *Gray*, *Slate*, *Black*} used in the cloaking scheme. For example, the color *White* translates to the scope SI; the color *Black* translates to the scope Context. Thus, when the scope is SI, the navigator reveals only SI to the query processor; when the scope is Context, the navigator reveals all information to the query processor.

The bi-level navigator realizes the vision (stated in Section 8.2.2) that a query processor can cloak information without explicitly coloring the nodes and without explicitly constructing query scope. Figures 9.8, 9.9, and 9.10 illustrate how the bi-level navigator realizes this vision.

We now provide an example of how cloaking can improve the efficiency of executing certain queries. Consider the path expression `//text()` to select all text nodes in the tree. Without cloaking (or, with the navigator scope set to Context), the navigator visits at least *12 nodes* when this expression is evaluated over the tree in Figure 9.6: Root, Comment, the text child of Comment, the three mark-association elements, and the child elements Descriptor and Context for each mark-association element. The navigator also visits all descendants of Descriptor and Context, but more importantly, it retrieves the context information from the base layer for each mark employed.

If the intent is to retrieve only comment text, the navigator scope may be set to SI. In this case, the navigator visits only *three nodes*: Root, Comment, and the text child of Comment. Also, it does not retrieve any context information from the base layer.

Section 9.4.3 provides experimental verification of the savings due to cloaking.

9.4. Evaluation

We have evaluated the extended XPath data model and the design of the bi-level navigator by implementation, by using the implementation in a variety of applications, and by running experiments. We first outline the implementation and some applications, and then describe the experiments.

9.4.1. Implementation

We have fully implemented the extended XPath data model presented in Section 9.2 and the design presented in Section 9.3. We have used the Sixml DOM implementation described in Section 7.6.1 to internally represent the data to be queried. (The navigator can use any of our three Sixml DOM implementations.)

The bi-level navigator (that is, the class SixmlNavigator) is implemented in C# by extending the class XPathNavigator included in the .NET Framework (.NET) [129]. Our navigator can work with both the XPath evaluator and the XSLT processor included in .NET (including those in Microsoft's distribution of .NET, for which we do not have the source code).

The .NET class XPathNavigator combines the functionality of an XPath evaluator and navigator. In our navigator, we have overridden the method evaluate to implicitly assign scope based on the expression to be evaluated (as discussed in Section 9.3.2). We have also overridden the navigation methods. The navigation methods generally follow the procedures in Figures 9.9 and 9.10, but some changes are made to suit the .NET implementation of the class XPathNavigator. In the design we presented, the methods MoveToNextSibling and MoveToPreviousSibling handle any kind of XPath node, but .NET handles movement among siblings differently: It uses MoveToNextAttribute to move from an attribute to the next attribute. Similarly, it uses MoveToNextNamespace to move among namespace nodes. It uses MoveToNext and MoveToPrevious for movements among other node kinds. (In .NET, XPathNavigator does not provide a means to move backward along the attribute and namespace axes.)

Due to reusing Sixml DOM, our bi-level navigator implementation caches the context information retrieved from the base layer for a mark in a hash table keyed on mark ID. If context information for the same mark is needed again, it is retrieved from the cache (if available) instead of the base layer. The context cache is global so context information can be reused even when a mark is reused in a different (possibly simultaneously executing) query.

The size of the context cache is configurable. By default, the size is bound only by available memory, but the size can be limited. When the cache size is limited, entries are evicted using a first-in, first-out policy, when needed.

The following list provides some high-level implementation statistics (as of this writing).

- Number of interfaces: 1
- Number of classes: 1
- Number of source files: 1
- Number of lines of code: 1,265
- Estimated time spent on implementation: 90 hours

9.4.2. Applications

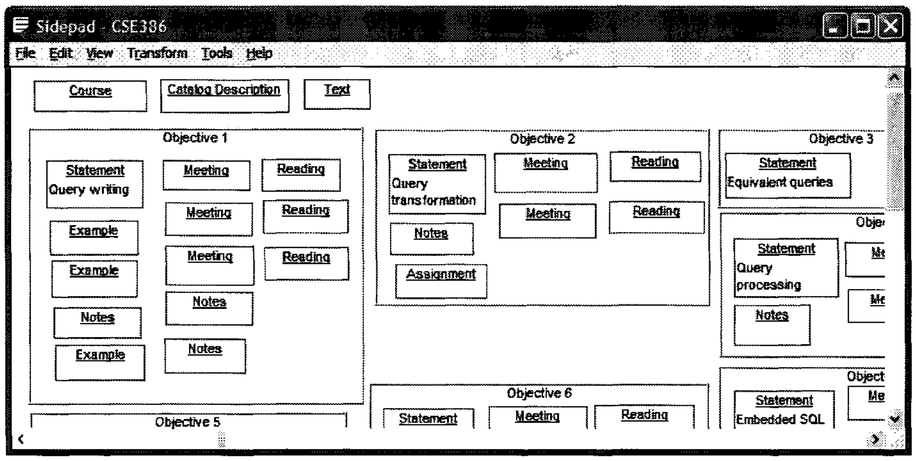
We have employed bi-level queries in a wide variety of applications. We introduce three such applications in this section. Chapter 10 reviews a further application to interchange bi-level information. We have also developed a graphical user interface to bi-level query processors with which a user can compose and execute bi-level queries over arbitrary Sixml documents.

9.4.2.1. Drafting a Survey Paper

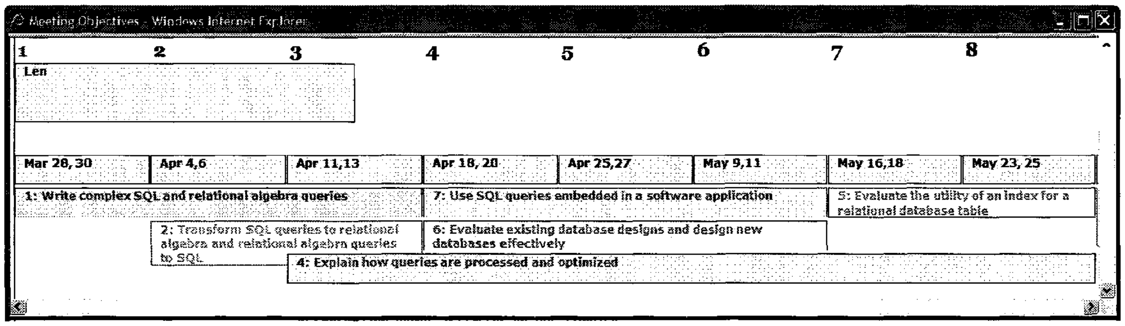
Figure 1.5 illustrates a simple use of bi-level querying. We obtained the HTML source used in this figure by transforming a Sixml representation of the Sidepad document shown in Figure 1.3. The source Sidepad document was one of three documents originally created to collect and organize information for a survey paper as a term project. The transformation was the result of applying an XSLT style sheet to generate an HTML outline of the survey paper, which was then imported into MS Word. The paper was completed in MS Word and converted to PDF for submission.

9.4.2.2. Creating Alternative SI Structures

We now illustrate the use of bi-level queries to create alternative SI representations (such as a timeline) from existing SI (such as a Sidepad document).



(a)



Mar 28, 30 | Apr 4,6 | Apr 11,13 | Apr 18, 20 | Apr 25,27 | May 9,11 | May 16,18 | May 23, 25

1: Write complex SQL and relational algebra queries

7: Use SQL queries embedded in a software application

5: Evaluate the utility of an index for a relational database table

2: Transform SQL queries to relational algebra and relational algebra queries to SQL

6: Evaluate existing database designs and design new databases effectively

4: Explain how queries are processed and optimized

(b)

Figure 9.11: Bi-level information displayed as a timeline. (a) A Sidepad document with marks to online course material; (b) The Sidepad document in Part (a) and the referenced base information transformed to a timeline

Figure 9.11(a) shows a Sidepad document with marks into online course material for a university class offering [89]. Most of the items in this document are grouped by the learning objectives for the class. For example, the group labeled Objective 1 contains an item with a statement of the first learning objective. The group also contains items for the three class meetings in which the learning objective is addressed, and items for notes and readings. In all, the document contains 58 items that incorporate 49 marks into 12 base documents. (Some marks are used more than once.)

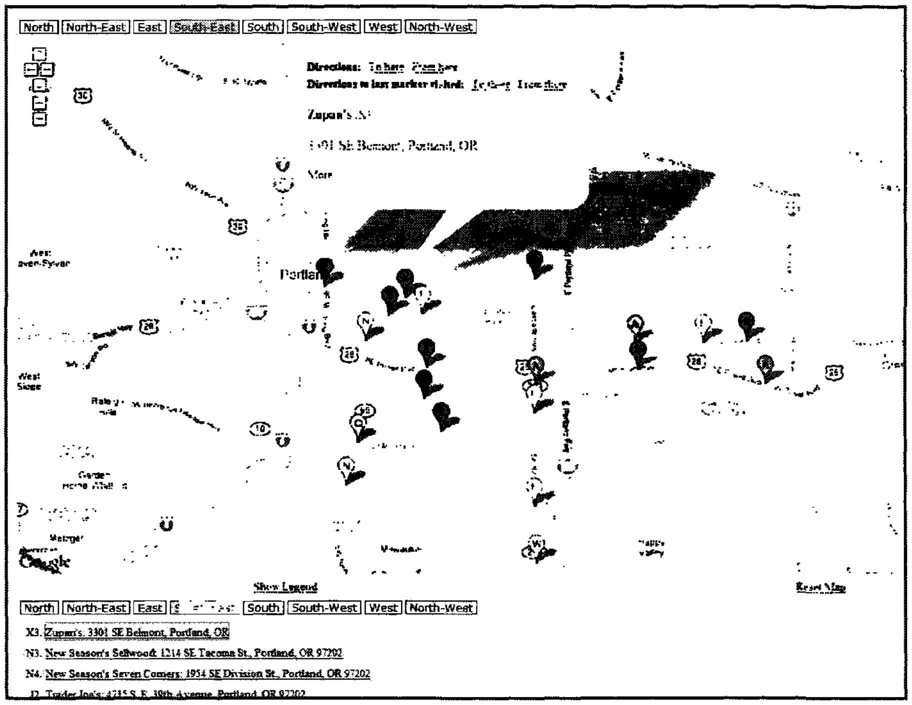
Figure 9.11(b) shows the Sidepad document of Figure 9.11(a) transformed to a timeline using a pair of XSLT style sheets: One style sheet generates timeline data from the Sidepad document, another draws the generated timeline data in a web browser. (The latter style sheet is based on the work of Kruchten [83].) The timeline lists the sequence of weeks in which the class meets, and shows the learning objectives met in each week. The name(s) of the instructor(s) teaching that week is (are) shown when the mouse cursor hovers over a week. All information displayed on the timeline (except the week numbers displayed at the top) is obtained from the base layer at query-execution time.

Any change to the source Sidepad document can easily be reflected on the timeline by simply re-applying the two style sheets.

9.4.2.3. Creating Mash-ups

Web applications called *mash-ups* combine information of varying granularity from different, possibly disparate, sources. We have created a utility called *Mash-o-matic* [115] to extract, clean, and combine disparate information fragments, and to automatically generate data for mash-ups and to generate the mash-ups themselves.

At its core, Mash-o-matic is a set of schemas for Sixml documents together with a set of bi-level queries to combine Sixml documents with the referenced base information, and a set of queries to convert the combined bi-level information into different display formats. For example, Mash-o-matic defines a schema to represent information about geographic landmarks as a Sixml document, and a set of XSLT style sheets to transform a Sixml document to data suitable for use by JavaScript [73] applications that use a map API such as Google Maps [53], Google Earth [52], and Yahoo! Maps [179]. (The same Sixml document can be transformed for use with different map APIs.) The style sheets also interact with third-party web services to *geo-code* landmark addresses (that is, to transform landmark addresses to map coordinates).



X3. Zupan's, 3301 SE Belmont, Portland, OR
N3. New Season's Sellwood: 1214 SE Tacoma St., Portland, OR 97202
N4. New Season's Seven Corners: 1954 SE Division St., Portland, OR 97202
D. Trader Joe's: 4715 S.E. 39th Avenue, Portland, OR 97202

Figure 9.12: A map-based mash-up. The mash-up shows grocery stores in the metropolitan area of Portland, OR (USA). This mash-up was developed for the Oregon Department of Agriculture

Mash-o-matic automates much of the process of preparing data for mash-ups. For example, we built a mash-up that displays a campus map for Portland State University in only about 5.5 hours (including data collection). We built a much more sophisticated map of grocery stores in the metropolitan area of Portland, OR (USA) in only 16.5

hours (including collecting and cleaning data). Much of the savings in the time and effort needed to develop these mash-ups was due to bi-level queries. (The grocery store mash-up, shown in Figure 9.12, was developed for the Oregon Department of Agriculture.)

9.4.3. Experiments

We now present the results of an experimental evaluation of the bi-level navigator. The navigator used the MSX implementation of Sixml DOM described in Section 7.6.1 to represent Sixml data in memory. A marks repository was maintained in an MS SQL Server 2005 (MSSQL) database and managed using the persistent marks repository implementation described in Section 7.6.1.

All C# code was compiled using MS Visual Studio 2005 [102]. The experiments were run in MS's distribution of the .NET Common Language Runtime (Version 2.0) [128] on an Intel Core Duo 1.66 GHz processor [65] with 1 GB of main memory. The operating system was MS Windows XP (Service Pack 2) [104].

The experiments involved executing query workloads on the Sixml documents listed in Table 7.2. The workloads were executed on documents in both the normalized schema and the nested schema. Documents in the *normalized schema* include only SI and their mark associations, but not mark descriptors. The mark descriptors for these documents are obtained on demand from a persistent marks repository at query-execution time. Documents in the *nested schema* include SI, the mark associations, and the mark descriptors. For both flavors of documents, context information was obtained on demand using the bulk accessor.

Figure 9.13 shows a thumbnail version of the normalized schema documents used in the evaluation. In a SISRS document, a mark is used only once. Each Comment element has an associated EMark. The attribute title of each Paper element has an AMark and the attribute's value is set to be the text excerpt retrieved from the mark.

In an SSIB document, a mark is used thrice within the element Event: twice with attributes and once with text content. Error and Update elements each use three distinct marks. All attributes and text content associated with a mark derive their values from marks, mostly using complex path expressions over context.

Table 9.6 lists the combinations of queries, documents, navigator types, schemas, query scopes, and query languages used in the experiments. (Appendix C lists the actual queries used.) In all, we evaluated 219 of these combinations: 88 combinations of the queries common to both SISRS and SSIB documents with the bi-level navigator, 64 combinations with the traditional navigator, 64 combinations specific to SISRS, and 3 combinations specific to SSIB. For brevity, we present the results for only representative combinations. We executed a query in each combination thrice and report the average execution time.

In the rest of this section, we use the term *Bi-level-X* to mean the bi-level navigator with scope *X*. *X* can be S, A, D, or C, and means the navigator scope is set to SI, Association, Descriptor, and Context, respectively. For example, Bi-level-A means the bi-level navigator with scope set to Association. The notion of scope does not apply to the traditional navigator.

Table 9.6: Queries used to evaluate the bi-level navigator. The entries in the last two columns show the query scope and the query language used. The letter S denotes the scope SI; A denotes the scope Association; D denotes Descriptor; and C denotes Context.

Query, document, and navigator combination	Information queried	Normalized schema	Nested schema
Queries common to all documents, executed using both traditional and bi-level navigators			
Q1. Retrieve all SI	SI	S (XPath), A (XSLT)	S (XPath), D (XSLT)
Q2. Retrieve all mark associations	Associations	A (XSLT), D (XSLT)	A (XSLT), D (XSLT)
Queries common to all documents, executed using the bi-level navigator			
Q3. Retrieve unique mark descriptors	Descriptors	D (XSLT)	D (XSLT)
Q4. List the base documents referenced	Descriptors	D (XSLT)	D (XSLT)
Queries over SISRS documents only, executed using the bi-level navigator			
Q5: Retrieve the text of all comments (three variations)	SI	S (XPath), A (XSLT)	S (XPath), A (XSLT)
Q6: Retrieve paper titles	Context	S (XPath), C (XPath)	-
Queries over SSIB documents only, executed using the bi-level navigator			
Q7. List the base documents for security events (SSIB-1 only)	Descriptors	D (XPath)	-
Q8. Create a timeline of "application hang" events (SSIB-8 only)	Context	S (XSLT), C (XSLT)	-

9.4.3.1. Retrieving SI (Q1)

This experiment compares the performance of Bi-level-S with the traditional navigator (which uses DOM for run-time representation) and with Bi-level-A when retrieving just the SI portion of a Sixml document.

```
<Reviews xmlns:sixml="http://schema.sixml.org">
 <Paper title="">
  <Comment reviewer="1">Comment 1<sixml:EMark sixml:markID="20051271054160.R-302.pdf_1"/></Comment>
  <sixml:AMark sixml:markID="20051271054160.R-302.pdf-title" sixml:target="title" sixml:valueSource="True"/>
 </Paper>
</Reviews>
```

(a)

```
<SSIB xmlns:sixml="http://schema.sixml.org" xmlns:e="urn:schemas-microsoft-com:office:spreadsheet">
 <Computer name="C3">
  <Events>
   <Event dateTime ="" kind="System" source="">
    <Description>
     <sixml:TMark sixml:markID="Ev_C3Sys_00001" sixml:valueSource="True"
      sixml:valueExpression="/Context/Content/XML/e:Workbook/e:Worksheet/e:Table/e:Row/e:Cell[position()=9]/e:Data"/>
    </Description>
    <sixml:AMark sixml:markID="Ev_C3Sys_00001" sixml:target="dateTime" sixml:valueSource="True"
     sixml:valueExpression="/Context/Content/XML/e:Workbook/e:Worksheet/e:Table/e:Row/e:Cell[position()=1 or position()=2]/e:Data"/>
    <sixml:AMark sixml:markID="Ev_C3Sys_00001" sixml:target="source" sixml:valueSource="True"
     sixml:valueExpression="/Context/Content/XML/e:Workbook/e:Worksheet/e:Table/e:Row/e:Cell[position()=3]/e:Data"/>
   </Event>
  </Events>
  <Errors>
   <Error dateTime="" source="" description=""><Notes></Notes>
    <sixml:AMark sixml:markID="Er_C3Err_001" sixml:target="dateTime" sixml:valueSource="True"/>
    <sixml:AMark sixml:markID="Er_C3Err_002" sixml:target="source" sixml:valueSource="True"/>
    <sixml:AMark sixml:markID="Er_C3Err_003" sixml:target="description" sixml:valueSource="True"/>
   </Error>
  </Errors>
  <Updates>
   <Update dateTime="2008-Jan-12"><Title><sixml:TMark sixml:markID="Upd_0132_title" sixml:valueSource="True"/></Title>
    <Description><sixml:TMark sixml:markID="Upd_0132_desc" sixml:valueSource="True"/></Description>
    <Reason></Reason><sixml:EMark sixml:markID="Upd_0132"/>
   </Update>
  </Updates>
 </Computer>
</SSIB>
```

(b)

Figure 9.13: Thumbnail of the test documents in the normalized schema. SI is bolded. (a) A SISRS document; (b) An SSIB document

Retrieving SI using Bi-level-S is easy because the simple XPath expression "." suffices (with the document root as the context node). In contrast, retrieving SI with the traditional navigator requires a 96-line XSLT style sheet containing 8 templates that employ 23 path expressions. Accomplishing the same task with Bi-level-A needs a 56-line style sheet containing four templates and 11 path expressions. (The number of path expressions in a query is sometimes used as a measure of query complexity [70] and query workload [76].)

Table 9.7 shows the time (in milliseconds) needed to retrieve SI for the SISRS and SSIB datasets in the normalized schema, for different navigator-document combinations. Figure 9.14 shows that Bi-level-S saves at least 50% time over the traditional navigator, and it saves at least 60% time over Bi-level-A.

Table 9.7: Time (in milliseconds) to retrieve SI and mark associations for different navigator and document combinations in the normalized schema. The suffixes S and A for the bi-level navigator denote the query scope SI and Association, respectively. Table 7.2 describes the documents used

	Time to access SI (ms)			Time to access mark associations (ms)		
Document	Bi-level-S (XPath)	Traditional (XSLT)	Bi-level-A (XSLT)	Bi-level-A (XSLT)	Traditional (XSLT)	Bi-level-D (XSLT)
SISRS-1	5.21	10.42	12.98	15.62	15.62	20.83
SISRS-2	10.42	20.83	26.04	31.25	31.25	46.88
SISRS-4	20.83	41.67	52.08	46.88	46.88	78.12
SISRS-8	36.46	78.12	109.38	93.75	93.75	156.25
SSIB-1	78.12	213.54	276.04	484.38	484.38	606.42
SSIB-2	156.25	411.46	572.92	640.62	718.75	828.00
SSIB-4	312.50	848.96	1093.75	1375.00	1666.67	1850.00
SSIB-8	643.75	1531.25	2239.58	2520.83	3333.33	3500.00

Bi-level-S retrieves SI faster than the other two navigators because it examines fewer nodes: The traditional and Bi-level-A navigators attempt 62% more node movements for the SISRS dataset. For the SSIB dataset, the traditional navigator attempts 93% more moves, and Bi-level-A attempts 73% more moves. Bi-level-S is faster also because of the use of XPath: XPath returns existing nodes, whereas XSLT always constructs new nodes (even when an existing node is to be returned as is) [177].

Bi-level-A is slower than the traditional navigator, despite attempting fewer node movements, partly due to the overhead to support bi-level navigation. It is also slower because of the initial work Sixml DOM performs to pair mark-association elements with target nodes. As in the case of traversing SI using Sixml DOM (described in Section 7.6.3.5), the performance gap between Bi-level-A and the traditional navigator narrows as the number of executions of a query increases and the cost of the initial work is amortized. We omit illustrating this phenomenon because neither Bi-level-A nor the traditional navigator is a viable alternative to Bi-level-S when retrieving just SI.

In terms of scalability, the execution times in Table 9.7 illustrate that the performance of all three navigators (Bi-level-S, traditional, and Bi-level-A) scales up well with the number of mark associations. Similarly, Figure 9.14 shows that Bi-level-S retains its advantage even for large inputs.

Retrieving SI from documents in the nested schema produced results similar to those obtained for the normalized schema in terms of execution time, but many more nodes were created due to the presence of mark descriptors. For example, retrieving SI from SISRS-1 created only 13,785 nodes in the normalized schema, but it created 49,083 nodes in the nested schema. The query execution times were higher due to the creation of additional nodes. Table 9.8 compares the time to retrieve SI for the SISRS dataset in the normalized and nested schemas.

In summary, when retrieving SI from a Sixml document, cloaking mark associations (using Bi-level-S) saves time, and using the normalized schema saves memory.

Table 9.8: Time (in milliseconds) to retrieve SI and mark associations for the SISRS dataset in the normalized and nested schemas

	Time (ms) to access SI using Bi-level-S (XPath)		Time (ms) to access mark associations using Bi-level-A (XSLT)	
Document	**Normalized schema**	**Nested schema**	**Normalized schema**	**Nested schema**
SISRS-1	5.21	5.21	15.62	15.62
SISRS-2	10.42	15.62	31.25	41.67
SISRS-4	20.83	26.88	46.88	72.92
SISRS-8	36.46	46.88	93.75	140.62

9.4.3.2. Retrieving Mark Associations (Q2)

This experiment compares the performance of Bi-level-A with the traditional navigator and with Bi-level-D when retrieving mark associations from a Sixml document, with the mark descriptor for each mark association excluded from the result.

Retrieving mark associations requires XSLT even with Bi-level-A because XPath lacks the functionality needed to select custom-named mark associations associated with elements. (XPath does not provide a means to lookup the URI for a namespace prefix). The style sheet for Bi-level-A has 74 lines, 6 templates, and 17 path expressions. The style sheet for the traditional navigator has 138 lines, 13 templates, and 34 path expressions. The Bi-level-D style sheet has 79 lines, 7 templates, and 17 path expressions. (The style sheets for Bi-level-A and Bi-level-D are identical except that the latter must explicitly exclude mark descriptors from the result.)

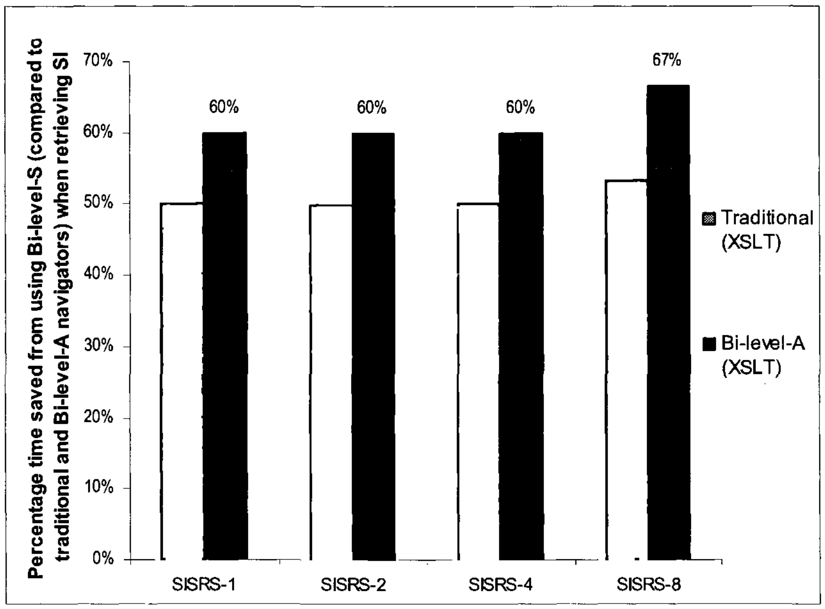
Table 9.7 shows the time (in milliseconds) to retrieve mark associations for documents in the normalized schema. Figure 9.15 shows that Bi-level-A performs at least as well as the traditional navigator, and it saves at least 20% of the time over Bi-level-D.

For documents in the nested schema, the relative performance of the three navigators was the same as with the normalized schema, but query execution consumed more memory due to the presence of mark descriptors. Query execution times were also higher due to the time needed to instantiate additional nodes. Table 9.8 compares the time to retrieve mark associations for the SISRS dataset in the normalized and nested schemas.

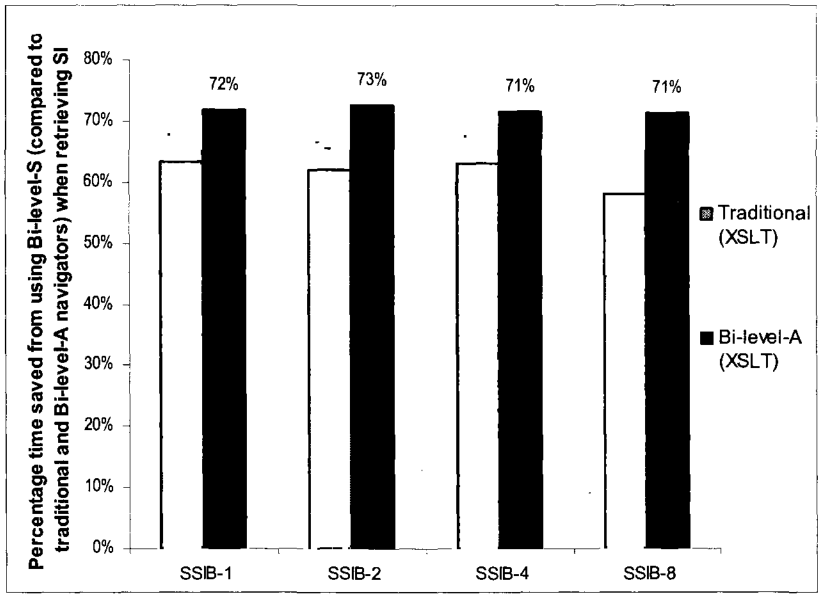
In summary, when retrieving mark associations from a Sixml document, cloaking mark descriptors (with Bi-level-A) saves time, and using the normalized schema saves memory.

9.4.3.3. Retrieving Mark Descriptors (Q3)

This experiment compares the performance of Bi-level-D when retrieving *unique* mark descriptors in the normalized schema and in the nested schema. In practice, this query is used to collect descriptors when interchanging bi-level information (as described in Chapter 10).

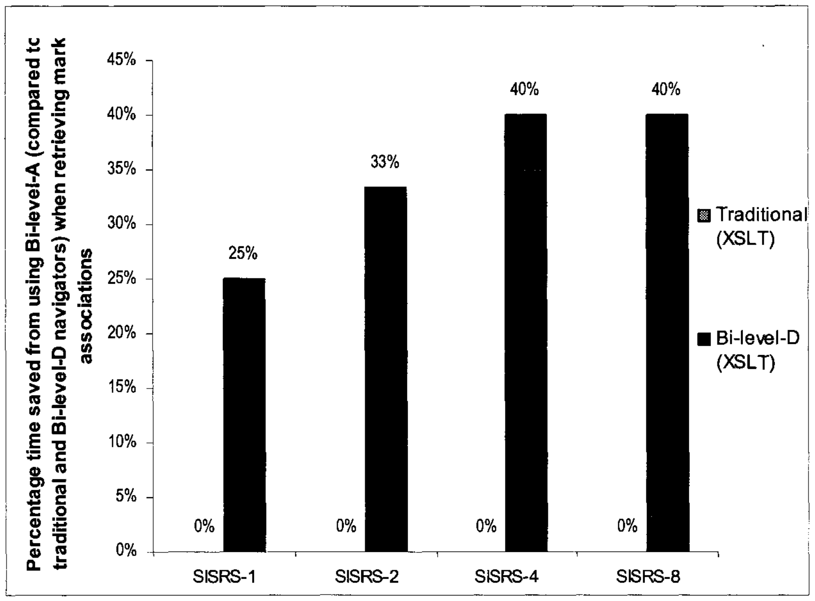


(a)

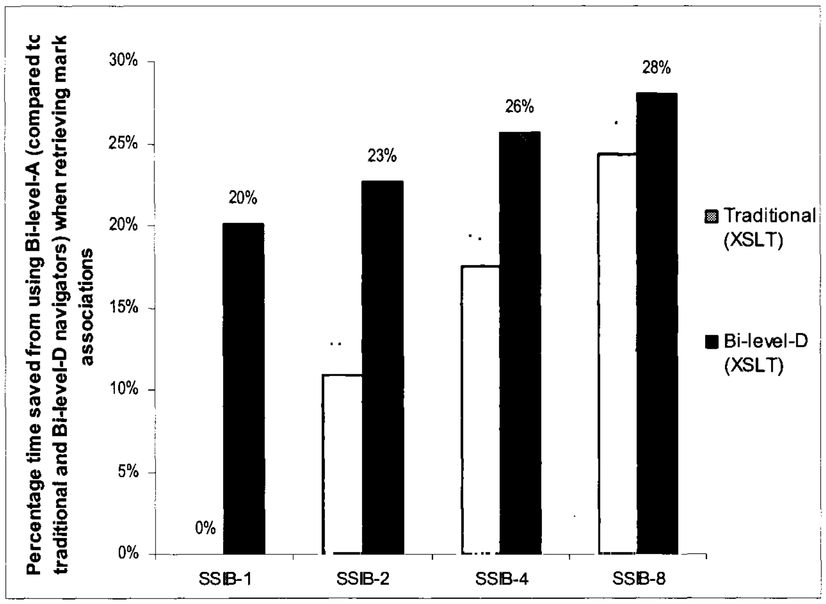


(b)

Figure 9.14: Percentage time saved using the bi-level navigator with scope SI when retrieving SI in the normalized schema. The column labeled 'Traditional' denotes the savings over the traditional navigator, 'Bi-level-A' denotes the savings over the bi-level navigator with scope Association. (a) Savings for the SISRS dataset; (b) Savings for the SSIB dataset



(a)



(b)

Figure 9.15: Percentage time saved using the bi-level navigator with scope Association when retrieving mark associations in the normalized schema. The column labeled 'Traditional' denotes the savings over the traditional navigator, 'Bi-level-D' denotes the savings over the bi-level navigator with scope Descriptor. (a) Savings for the SISRS dataset; (b) Savings for the SSIB dataset

A mark association in the normalized schema contains only mark IDs but not mark descriptors. Thus, it suffices to test the equality of only the descriptors of marks with distinct IDs. In contrast, a mark association in the nested schema includes a mark descriptor. (As described in Section 7.4.3.1, a mark association in the nested schema may include a mark ID, but a mark repository may employ an equivalent descriptor with a different ID. The nested schema documents used in the evaluation do not include mark IDs.) Thus, determining a descriptor's uniqueness requires that the descriptor be tested against all other descriptors.

If each mark ID in a normalized-schema document is used only once (as is the case with SISRS documents), the test for equality of mark IDs always fails, giving a performance edge to its nested-schema counterpart. If a mark ID is reused within a normalized-schema document (as is the case with SSIB documents), fewer mark descriptors are created, giving it an edge over its nested-schema counterpart. In this context, the speed with which mark descriptors can be retrieved from the repository (in case of the normalized schema), and the effect of the increased number of mark descriptors (in case of the nested schema) can influence performance significantly.

Two mark descriptors are equal if their serialized string representations are equal, but neither XPath nor XSLT provide an easy and efficient means to perform this test. We work around this limitation by building an on-the-fly hash index on mark descriptors.

Table 9.9 shows the time (in seconds) to retrieve unique mark descriptors for the complete SISRS dataset and for the document SSIB-1 in the SSIB dataset. We abandoned this experiment with the other three documents in the SSIB dataset due to the excessive amount of time they needed to complete, and completing them would not have given us any new insight. The last row in the table shows the savings obtained, or the overhead incurred, by using the nested schema.

Table 9.9: Time (in seconds) to retrieve unique mark descriptors. A positive value in the last row denotes savings from using the nested schema; a negative value indicates an overhead

Schema	SISRS-1	SISRS-2	SISRS-4	SISRS-8	SSIB-1
Normalized	3.25	13.92	60.09	247.39	1638.08
Nested	2.53	11.34	51.38	226.75	3052.88
Savings (overhead) due to nested schema	22.1%	18.5%	14.5%	8.3%	-86.3%

For the SISRS dataset, using the nested schema saves time because each mark is used only once in a document, but the savings decline as the number of mark associations increases because mark descriptors can be retrieved more efficiently from the persistent repository (thanks to MSSQL) at high volumes than retrieving them from a disk file. The number of mark descriptors created was the same for both schemas.

For the SSIB dataset, using the nested schema causes an overhead because each mark is used thrice within a document. Each use of the mark creates a mark descriptor in the nested schema, but only one mark descriptor is created in the normalized schema regardless of the number of times a descriptor is used. For example, for SSIB-1, 38,883 mark descriptors were created in the nested schema, but only 12,961 mark descriptors were created in the normalized schema.

In summary, when retrieving unique mark descriptors from a Sixml document, using the normalized schema saves time and memory when a mark is used more than once within the document. The nested schema saves time when marks are used only once, but the savings decline as the number of mark associations increases.

9.4.3.4. Listing Base Documents Referenced (Q4, Q7)

Here we compare the performance of Bi-level-D when retrieving a list of unique base documents a Sixml document references (Query Q4). In practice, this query is used when interchanging bi-level information. (This experiment retrieves the location of base documents, not the base documents themselves.)

Table 9.10: Time (in seconds) to list the unique base documents referenced. The results shown correspond to Query Q4. A positive value in the last row denotes savings from using the nested schema; a negative value indicates an overhead

Schema	SISRS-1	SISRS-2	SISRS-4	SISRS-8	SSIB-1
Normalized	1.00	3.95	17.94	82.83	1109.17
Nested	0.95	3.78	17.81	83.41	1474.05
Savings (overhead) due to nested schema	5.0%	4.3%	0.7%	-0.7%	-32.9%

To retrieve the list of unique base documents, we simply tested the uniqueness of the location (for example, file path) of base documents. Table 9.10 shows the time (in seconds) to retrieve the list of unique base documents. The last row shows the savings obtained, or the overhead incurred, by using the nested schema. Not surprisingly, the results are similar to those obtained in the experiments to retrieve unique mark descriptors.

We also evaluated the following simple XPath expression (Query Q7) over SSIB documents to retrieve a list of base documents referenced by attributes of "security" events:

```
SSIB/Computer/Events/Event[@kind='Security']/@*//sixml:Descriptor/Doc
```

The result of this query is empty because no SSIB document tested contains a "security" event. No marks need to be instantiated when evaluating this query because all components of the path expression up to `@kind='Security'/@*` reference only SI nodes, and no SI node satisfies these components.

When evaluating this expression, indeed, no marks were instantiated for documents of either schema. No mark descriptors were instantiated when using the normalized schema, but all mark descriptors in the input document (38,883 for SSIB-1) were instantiated for the nested schema.

9.4.3.5. Focused and Unfocused Path Expressions (Q5)

This experiment illustrates the ability to employ unfocused path expressions, yet avoid some of the performance penalties associated with such expressions.

A *focused path expression* is an expression that guides the query processor strictly along the path of interest. An *unfocused path expression* does not guide the processor in this manner. Table 9.11 lists an example expression of each kind of expression. Activities such as data exploration and ad-hoc querying are easier with unfocused expressions.

A focused expression causes fewer navigator movements, but is sensitive to schema revisions. An unfocused expression causes more navigator movements, but tends to be resilient to schema revisions and is easier to develop. Cloaking can reduce the number of navigator movements for unfocused expressions, thereby reducing query-execution effort while making it easier to express queries.

Table 9.11: Number of navigator movements attempted to retrieve comment text. The results shown correspond to Query Q5 evaluated over the document SISRS-1

Path expression	Traditional	Bi-level-S	Savings due to Bi-level-S
Focused: `/Reviews/Paper/Comment/text()`	7742	5834	33%
Unfocused: `//text()`	13521	9705	39%

Table 9.11 shows the number of movements Bi-level-S and the traditional navigator attempt when evaluating equivalent focused and unfocused expressions to retrieve comment text from a SISRS document (Query Q5). The last column shows the percentage savings in navigator movements (over the traditional navigator) due to Bi-level-S. Bi-level-S makes fewer movements because it cloaks mark associations.

9.4.3.6. Micro Queries (Q6, Q8)

This experiment illustrates the benefits of using micro queries. A *micro query* is a query used to derive the value of an SI node from the context of marks using the mark association attributes valueSource and valueExpression (as described in Section 7.4.3.4). For example, in Figure 9.13(a), the value of the attribute title in the element Paper is the text excerpt retrieved from the associated mark. In Figure 9.13(b), the text content of the element Description inside an Event element is the content of the 9th column of a spreadsheet row.

A micro query *implicitly* obtains context information even with the query scope set to SI. Without micro queries, a bi-level query needs to use path expressions that navigate *explicitly* to context information, but doing so requires that the navigator scope is set to Context. As demonstrated in experiments described thus far, the narrower query scope (SI) possible with micro queries improves query execution in several cases (because the query processor potentially visits fewer nodes).

Table 9.12 compares the query-execution performance with and without micro queries, when retrieving paper titles (Query Q6) from the document SISRS-8, and when creating a timeline of "application hang" events (Query Q8) from SSIB-8. The first version of each query exploits the micro queries employed in Figure 9.13. The second version explicitly accesses context information. The table shows that using micro queries saved 17.3% time for Q6 and 85.3% time for Q8.

In summary, micro queries provide an easy and efficient means of integrating parts of context information into SI.

Table 9.12: A comparison of the performance of queries that exploit micro queries with queries that do not. All queries were executed over normalized-schema documents

Query	Scope	Time (ms)	Movements
Q6: Retrieve paper titles (SISRS-8)			
`/Reviews/Paper/@title`	SI	2765.62	1282
`/Reviews/Paper/@title/*/Context/Content/Text`	Context	3343.75	10179
Q8: Create a timeline of "application hang" events (SSIB-8)			
XSLT style sheet using micro query	SI	625.00	374
XSLT style sheet without using micro query	Context	4250.00	758

9.4.4. Evaluation Summary

Our implementation of the bi-level navigator establishes the feasibility of our representation choice and our design of the navigator. The applications described in Section 9.4.2 and the queries used in the experimental evaluation illustrate that the bi-level navigator is of general purpose.

The experimental evaluation highlights the following aspects:

- The bi-level navigator works with existing query processors to support the full range of queries possible in XPath and XSLT. Bi-level queries can be expressed without using any extensions to the query language.
- The bi-level navigator performs better than the traditional navigator when retrieving the SI portion of a Sixml document. It performs at least as well as the traditional navigator when retrieving mark associations.
- The normalized schema provides better performance than the nested schema in most cases. The nested schema performs better for queries that examine mark descriptors in smaller documents, but it generally increases the memory footprint.
- Cloaking makes query expression easier, and it can provide significant time and memory savings. It reduces the performance penalties for unfocused path expressions, making activities such as data exploration easier.
- Micro queries embedded in Sixml documents provide an easy and efficient means to integrate context information with SI even when context information is cloaked to the query.

9.5. Related Work

In this section, we provide an overview of two systems (Active XML and MetaXPath) related to bi-level querying. We also compare our approach to traditional data integration approaches and to two tools for producing data mash-ups.

9.5.1. Active XML

Section 7.7.1.2 introduced the representation aspects of Active XML (AXML) [3]. Here, we give an overview of the query-processing aspects of AXML. For ease of reading, we repeat here parts of the text from Section 7.7.1.2.

AXML [3] provides a means to describe parts of an XML document intensionally using *service-call elements* that encode calls to *web services* [161] (which provide a means of executing code located on a remote computer). The following is a hypothetical AXML representation of a part of the information in the element Comment in Figure 7.1.

```
<Comment xmlns:axml="http://activexml.net">
  <axml:sc>sixml.org/getExcerpt(<mark ID="23">)</axml:sc>
</Comment>
```

The element axml:sc denotes a service call, and its child elements denote service parameters. The URI sixml.org/getExcerpt identifies a hypothetical web service to obtain the text excerpt of a mark. An AXML service-call element is similar to our mark-association element.

AXML uses a special query processor to execute service calls. At run time, this query processor *replaces* each service-call element by the XML element the web service returns. For example, the following AXML fragment shows a possible result of executing the example service call. Here, the result element Excerpt has replaced the service-call element.

```
<Comment xmlns:axml="http://activexml.net">
  <Excerpt>provides...</Excerpt>
</Comment>
```

The AXML query processor completes service calls *lazily*, so that only the calls needed to answer a query are completed. It achieves this goal by analyzing the query (to determine the service calls in the path of a query), and by comparing the type information associated with a service call to the type of the information that the corresponding web service returns. The query processor might also push some predicates to web services to reduce the number of replacement operations performed over the input tree. Also, the processor can invoke web services in parallel to reduce the overall query-execution time.

AXML extends XQuery so that service-call elements can be queried. For example, the path expression `Comment/getExcerpt()` selects the service-call elements contained in the element Comment. Here, the extension is to allow the use of parentheses to indicate access to a service-call node, instead of accessing the result from a service call. However, it is unclear how a service call can be queried after it has been replaced by its results. (No query-language extension would be needed to access a service-call node if the result were inserted as a child of axml:sc.)

We do not use a special query processor, but only a special navigator, to facilitate queries, and queries are expressed in *existing* query languages. Also, mark associations can be queried even after external data is added to the document.

The AXML query processor and the bi-level query processor both lazily expand a source document. Both processors retrieve external data only if a query needs that data, but only the AXML processor pushes predicates on external data to the external source (potentially reducing the amount of external data retrieved).

The path expression `/*` provides an interesting point of comparison for the two query processors. When evaluating this expression, the AXML query processor evaluates all the service calls embedded in the source document, even if the user wishes to examine only the service-call nodes. With the bi-level query processor, the user can set the scope of the query so that no external data is retrieved if he does not wish to examine external data.

9.5.2. MetaXPath

MetaXPath [40] extends the XPath data model to allow metadata to be associated with a node. The metadata of a node is a document also represented in the MetaXPath data model. Thus, a node in the metadata document may have its own metadata.

Figure 9.16 shows a data document and its metadata represented in the MetaXPath data model. The data document is shown under the heading Level 0; the metadata document for the nodes in Level 0 is shown under the heading Level 1; and the meta-metadata document is shown under the heading Level 2. For brevity, the document in Level 2 is not shown completely.

MetaXPath adds a property called "meta" to *element nodes* and nodes that can be children of an element node. A node uses this property to reference its metadata document. A node inherits this property from its parent, but the inherited value may be overridden. An attribute cannot have metadata, because only an element node and its children may have the "meta" property.

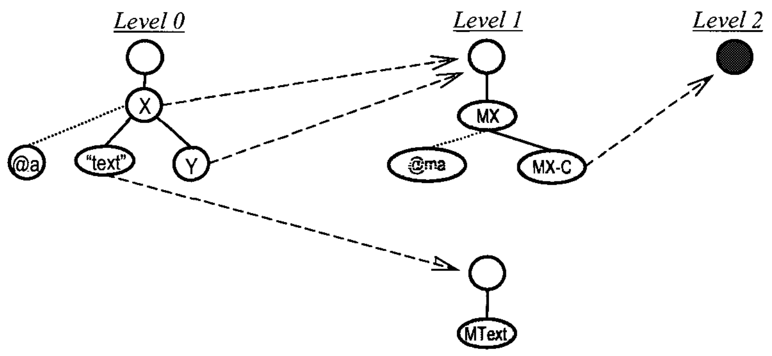


Figure 9.16: Example data document with metadata populated using the MetaXPath data model. A dashed arrow indicates connection between a node and its metadata document

The dashed arrows in Figure 9.16 indicate the use of the "meta" property. The element Y has the same metadata as its parent X because Y does not override the inherited metadata. The text child of X has its own metadata.

MetaXPath extends XPath by the *level-shift operator* ^, which navigates from the current node to the root node of the metadata document for the current node. For example, the path expression `/X/^MX` returns the metadata element MX. However, no means is defined to navigate from a metadata node to its data nodes.

MetaXPath serializes each document and metadata document as a separate XML document, but it does not define a means to specify the connection from a node to its metadata document. Also, it does not define an API to create this connection at run time.

MetaXPath supports a limited form of cloaking, because a node's metadata is visible only when the level-shift operator is used. For example, the expression `X` returns only the element X. However, there is no way to cloak data within a document. For example, the expression x returns X including the text node and Y.

The property "meta" can be used to associate mark associations, and the level-shift operator can be used to navigate from an SI node to its mark associations. However, this approach limits the node types with which marks may be associated, and it does not allow navigation from a mark association to its target node.

9.5.3. Data Integration Systems

Data integration is the process of unifying data in different sources to present a single view of the unified data. Traditionally, data integration involves specification of mappings at data model, schema, and semantic levels [133]. Bi-level querying can be seen as providing data integration, but it does not perform schema mapping or semantic mapping. It does transform data from base information models to match the SI model.

Using the traditional data-integration approach typically requires much design-time effort. Thus, it is not suitable for *situational applications* [75] (which are applications developed for a small group of users and often designed to be short-lived) and for most mash-ups. Our "lightweight" approach is better suited for these applications. For example, the SA Superimposed Scholarly Review System (SISRS, described in Section 4.9.2) can be used for a specific conference without much up-front effort. Similarly, a mash-up that displays a campus map that includes information retrieved from web pages of different academic departments is easily assembled using our system. (One such mash-up is available at http://sparce.cs.pdx.edu/cmap.)

The traditional data-integration approach typically requires each source to present a single schema, but that expectation might not be reasonable for some applications. For example, in the SISRS application, a paper being reviewed can be broken down in several ways: pages and lines; or, sections, paragraphs, sentences, and words. Choosing a single schema in this application also forces a single addressing scheme. Our approach allows different schemas to be superimposed over the same source.

Using our approach does not preclude the use of traditional data-integration approaches. Initially using our lightweight approach, and gradually integrating some sources using the heavy-duty traditional approach, would be consistent with a "Pay-as-you-go" approach [87] to data integration.

9.5.4. Tools to Produce Data Mash-ups

Damia [145] is a tool to produce data mash-ups from XML sources and from sources that can be transformed to XML. (A *data mash-up* is a document that contains information drawn from different sources [120].) Each source is transformed to XML and represented using a variation of the XQuery data model [175], and parts of the transformed XML are processed using special operators. A mash-up may use only parts of a source, but the *complete* source is transformed to XML.

In contrast to Damia, in our approach, only the base parts that a mash-up uses (for example, just the sub-documents referenced; not the entire containing documents) are transformed to XML, and the transformation is *on demand.* Also, the transformed

XML can be processed using existing query languages and query processors. (Section 11.2.1 further discusses the operations on data mash-ups.)

Yahoo! Pipes [180] is a visual editor to assemble data mash-ups using *complete* information sources, not fragments. It supports operations such as sort and filter over web feeds, but it does not support the expression and manipulation of a mash-up using standard XML tools. (Yahoo! Pipes might internally represent a network of pipes as XML, but that representation is not exposed.)

In general, both Yahoo! Pipes and Damia are designed to assist non-technical people in assembling mash-ups. In contrast, our approach allows a developer to produce mash-ups, and might form the basis for a tool such as Yahoo! Pipes and Damia.

9.6. Summary and Conclusions

In this chapter, we have presented the design, implementation, and evaluation of a bi-level navigator for use with traditional XML query processors to evaluate bi-level queries over Sixml documents. Our design separates query evaluation from tree navigation, making it possible to use custom navigators based on application needs. We use one such custom navigator to enable bi-level querying.

Our bi-level navigator design also allows the navigator to internally represent a Sixml document as a Sixml DOM tree. Using Sixml DOM simplifies the design of the navigator and it allows the navigator to exploit the following features of Sixml DOM: the use of the bulk accessor to retrieve context information, on demand retrieval of mark descriptors and context information, and caching of mark descriptors and context information.

Table 9.13: A summary of capabilities that the different combinations of XML tools provide to a developer in a bi-level query setting. The entry A^* in the column "Information queried" means the query must explicitly recognize mark associations. The entry "SI (auto)" in the column "Cloaks possible" means the scope is automatically SI because the traditional navigator cannot recognize mark associations

Document type	Schema	DOM kind XML/Sixml	Navigator type Traditional/Bi-level	Information queried	Cloaks possible	Micro queries
XML	Any	Either	Either	SI	None	No
Sixml	Normalized	XML	Either	SI, A^*	None	No
Sixml	Nested	XML	Either	SI, A^*, D	None	No
Sixml	Normalized, nested	Sixml	Traditional	SI	SI (auto)	Yes
Sixml	Normalized, nested	Sixml	Bi-level	SI, A, D, C	SI, A, D, C	Yes

Both Sixml DOM and the bi-level navigator are designed to interoperate with DOM and the traditional navigator. They also support both the nested schema and the normalized schema. An SA developer can choose the DOM, navigator, and schema combination that is appropriate to the task at hand. For example, the developer may execute different queries over the same document instance using either the bi-level navigator or the traditional navigator. He can also mix mark associations in the nested schema and the normalized schema in the same document. Table 9.13 summarizes the different capabilities the various combinations of these components and the two schemas provide to a developer.

The bi-level navigator, together with the Sixml representation and Sixml DOM, satisfies all the seven goals we set in Section 5.3.1 for transformation of bi-level information in the XML model:

- An SA developer is free to use any SI schema (Goal G1), and associate any number of marks with any part of SI (G2). The developer can choose to indicate only mark associations in a document and omit mark descriptors, or embed mark descriptors in the document, or use a combination approach (within the same document).
- Most bi-level queries are executed more efficiently (G3) than possible with the traditional navigator. If needed, the developer may switch to the traditional navigator to query the same document instance.
- Query execution performance scales up well to large documents involving thousands of marks (G4).
- Navigation from SI to marks to mark contexts is done naturally using path expressions (G5).
- Cloaking preserves queries over SI and their results (G6).
- No new operators or functions are needed to express bi-level queries (G7).

In terms of performance, the bi-level navigator performs better than the traditional navigator in many cases. The traditional navigator is better when retrieving mark associations from smaller documents, but it does not automate bi-level querying tasks such as recognizing mark associations and retrieving context information. Also, it does not support cloaking. We have shown that cloaking improves query execution in many cases.

Several improvements to the bi-level navigator are possible. For example, we currently allow specification of one scope for an entire query. This approach has proven quite useful in improving query efficiency, but assigning different scopes to different parts of a query can further improve performance for some queries.

For instance, consider the task of retrieving mark associations for all SI attributes. Currently, the path expression `//@*/*` accomplishes this task if the query scope is set to Association, but the query processor examines non-SI attributes even though they cannot have marks. An improvement would be to use the scope SI to retrieve attributes and then to use the scope Association to retrieve mark associations. For example, we might use the expression `scope-A(scope-SI(//@*)/*)`, where the function `scope-SI` executes its argument expression in scope SI, and the function `scope-A` executes its argument in the scope Association.

This proposed extension violates our Goal G7 for bi-level querying by introducing new functions to the query language, but it can improve query execution. For example, in the current approach, the navigator attempts 29,581 movements to retrieve mark associations for all SI attributes in the document SISRS-1. In contrast, the new approach would attempt only 16,226 movements, a savings of over 45%.

This possibility for improvement motivates our future work on formalizing and implementing assignment of scope to query parts.

This chapter concludes our discussion on transformation of bi-level information. Chapter 10 discusses the use of the results from this chapter (especially, Queries Q3 and Q4) in interchanging bi-level information among SA users.

10. Interchanging Bi-level Information

This chapter discusses a means to interchange bi-level information among SA users. It introduces the notion of *SI-dependency graphs* and describes a run-time service that can be used with any SA to interchange bi-level information.

Thus far in this dissertation, we have used the term *bi-level information* to mean SI and the referenced base parts. However, in this chapter, we use that term to mean SI and the referenced base documents.

10.1. Introduction

Several situations exist where interchanging SI would be beneficial: collaboration, publishing and archiving (for example, depositing a Sidepad document in a digital library collection), studying an expert's comments, reviewing a paper, and moving information from one computer (setting) to another. Our experience building SAs has shown us that the task of interchanging SI is non-trivial and that a run-time service to interchange SI can save much SA-development effort.

Interchanging SI means interchanging bi-level information because fully exploiting SI requires access to the referenced mark descriptors and base documents. For example, activating the mark attached to a Sidepad item received from another user requires that the descriptor for the mark associated with the item, and the corresponding base document also be accessible from the receiver's computer.

Though we discuss interchanging bi-level information, for ease of writing, we use the phrase "interchanging SI" in the rest of this chapter.

Some of the challenges in interchanging SI are due to the use of an identifier (ID) in SI to reference a descriptor in a repository, and due to the differences across computers in the location of base documents. (For example, one user might store base documents on a local disk; another might save them on a network drive.) Other challenges are due to our desire to support interchanging of SI in any schema and data model.

One way to interchange SI is to use a shared descriptor repository and a shared base-document repository. (Section 3.2.2 introduced the notion of descriptor repository.) This approach necessitates an access-control mechanism, because users might wish to share only some of the SI, and with only some users. Further, it would not address the need to share across "SI worlds" (caused by different repositories), and it would not facilitate disconnected operations. For example, a researcher might need to work offline when traveling.

The 'Save As Web Page, complete' (or just 'Save As') feature in web browsers such as Firefox [46] to save a web page to a user's local disk suggests an alternative solution. This feature also serves to illustrate some of the considerations for a service to interchange arbitrary SI.

When the user invokes the 'Save As' feature in a web browser (as observed in Firefox 1.5 [46] and MS Internet Explorer 7.0 [95]), the browser saves the source web page to a local folder the user chooses. It also saves the resources (such as images and frames) that the saved page contains to a *resources folder* in the same folder where the web page is saved. For example, if the user saves a web page with the title 'index' to the folder `C:\Out`, the browser saves the resources to the folder `C:\Out\index_files`. After creating the resources folder, the browser *alters* the saved copy of the web page to use the contents of the resources folder. It also *replaces* each relative URL (that is, a URL that does not specify a server) in the saved page with an absolute URL.

To interchange the saved web page, the user sends the saved page *and* the resources folder to other users. A receiver is free to save the received page in any folder on his local disk, but he must save both the page and the resources folder to the same folder. For example, if the receiver saves the page in the folder `E:\In`, he must also place the resources folder inside the folder `E:\In`.

An SA-independent service to interchange arbitrary SI cannot directly use a web browser's 'Save As' approach because it entails the possibility of changing IDs of descriptors (as will be described in Section 10.4). Changing descriptor IDs in turn would require changes to SI (to reflect the changed IDs), but the interchange service cannot alter SI because, by design, it is unaware of the SI models.

Figure 10.1 shows a reference model for our run-time service to interchange arbitrary SI. The dashed arrows indicate data flow. The service consists of two parts. A *packing* part places the SI document, and the descriptors and (optionally) the base documents on which the SI document depends, into a single *SI package file*. An *unpacking* part lets a package receiver extract the SI document and base documents to any accessible location, and updates the receiver's descriptor repository, all without altering SI. The receiver need not follow the sender's folder structure for the received documents. In fact, the receiver may extract each base document to a different folder (or drive or computer), if desired.

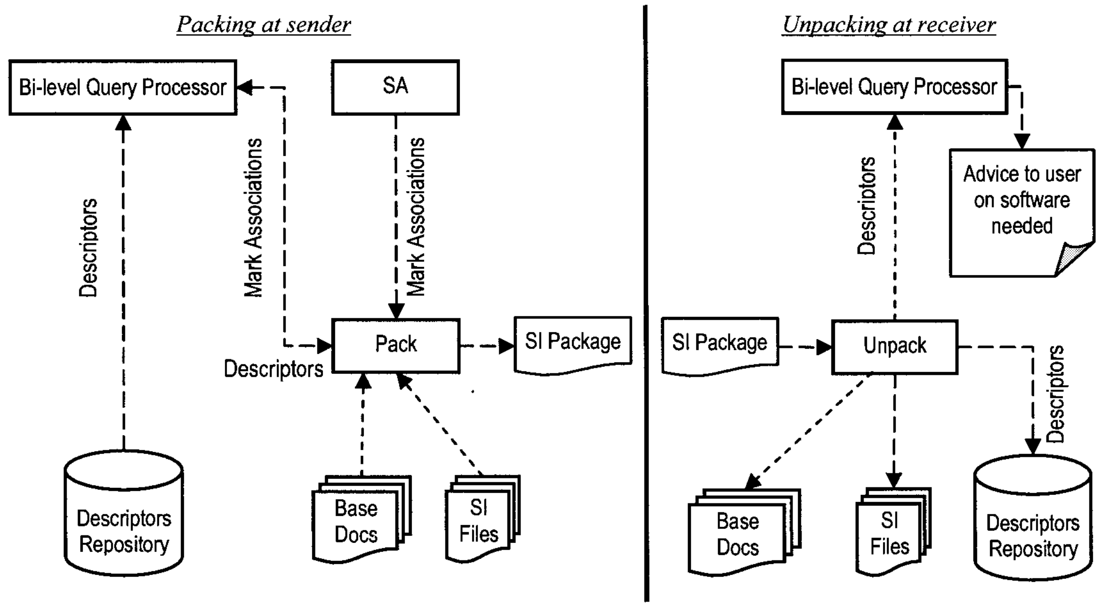


Figure 10.1: A reference model for the run-time service to interchange bi-level information

Notable aspects of the interchange service are:

- A package file is modeled as an *SI-dependency graph* (described in Section 10.2).
- The packing process (described in Section 10.3) uses the bi-level query processor to analyze the information dependencies of the SI to be packaged.
- The packing process can package SI for any SA able to supply SI as files.
- The unpacking process (described in Section 10.4) can unpack SI packages for any SA. In fact, as seen in Figure 10.1, an SA is not involved in the unpacking process.
- The service works with SI that uses IDs to reference descriptors in a repository *and* with SI that directly includes mark descriptors.

An SI package does not include application software (code). A receiver needs to have the necessary software such as SAs, base applications, and context agents. The unpacking process does advise the receiver as to the base applications and context agents on which the unpacked SI depends.

10.2. SI-Dependency Graphs

In this section, we informally introduce the notion of SI-dependency graphs. An *SI-dependency graph* is a directed acyclic graph we use to model an SI package. It provides a conceptual basis for our approach to interchanging SI.

Figure 10.2 shows a partial SI-dependency graph for the Sidepad document shown in Figure 1.3. Dashed horizontal lines distinguish different regions of the graph. The nodes in the region labeled SI represent the SI to be interchanged. The nodes in the Descriptors region denote the distinct descriptors SI references and the descriptors the referenced descriptors depend on, and so on. For example, the SI items S1 and S2 reference the mark descriptor M4. M4 in turn references the document descriptor D6, which in turn references the application descriptor A8. (Figure 3.4 shows this organization of descriptors.)

The nodes in the Base region denote the distinct base documents that the document descriptors reference. A base document might reference other documents. For example, a web page might reference an image file. Nodes in the External region denote such external documents, and dotted arrows indicate references to external documents.

Edges in an SI-dependency graph denote information references. We have annotated representative edges in Figure 10.2 to indicate how the various references are materialized: An SI element (such as a Sidepad item) references a descriptor using its ID. A descriptor also references another descriptor using an ID. A document descriptor references a base document using a "path", such as a file-system path or a URL. A base document may reference an external document using any mechanism such as URLs and links in the Object Linking and Embedding protocol (OLE) [18].

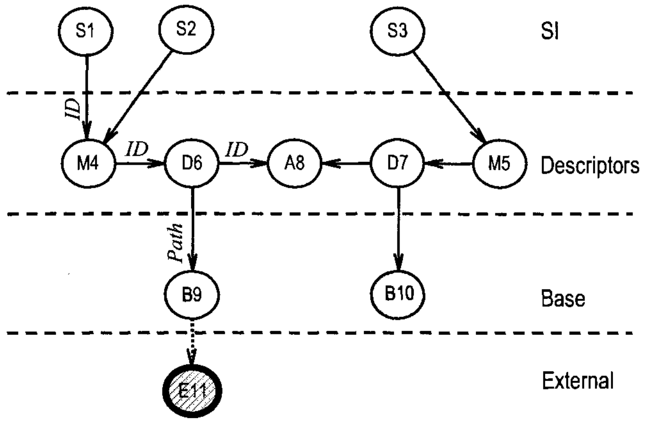


Figure 10.2: An SI-dependency graph

We now discuss the acyclic nature of an SI-dependency graph: Edges within the SI region may contain cycles, but we ignore all edges within the SI region because we wish to interchange SI without knowledge of the SI semantics. That is, for the purpose of interchanging SI, we can assume that the SI region has just one node and that all edges from the SI region to the Descriptors region originate from that node.

By design, the edges within the Descriptors region cannot cause cycles. Similarly, an edge between SI and a descriptor cannot cause cycles, nor can an edge between a document descriptor and a base document.

Edges within the Base region may contain cycles, but we ignore them because those edges are handled outside our service. We ignore the nodes in the External region and the edges to and from those nodes because they too are handled outside our service. For example, in Figure 10.2, we ignore the node E11 and its incoming edge. Analyzing these edges would allow us to build a more complete package, but such analysis is not central to sharing SI. That is, for the purpose of interchanging SI, we can assume that the External region does not exist.

10.3. Creating Packages

The packing process analyzes information dependencies by examining the descriptors used by the SI to be interchanged. For this purpose, the SA must provide the packing process a Sixml document containing the mark associations for the SI to be interchanged. This document does not need to represent the SI, and it can use any type of mark association. The need to supply this Sixml document does not mean the SA must store its SI as Sixml data, but an SA that represents its SI as Sixml data has an advantage, because it may supply the SI document as is.

For example, the Sidepad application stores its SI in a proprietary format, but to interchange a Sidepad document, it prepares a Sixml document that describes only the mark associations used in the Sidepad document. Figure 10.3 shows a Sixml document that describes the mark associations for the three Sidepad items depicted in Figure 10.2. This Sixml document does not at all represent the organization of information in a Sidepad document (for example, groups and items are not discernible), but it expresses all the mark associations used. Also, it uses the simplest mark-association type, EMark, to express mark associations.

```
<Marks xmlns:sixml="http://schema.sixml.org">
  <sixml:EMark sixml:markID="M4"/>
  <sixml:EMark sixml:markID="M5"/>
</Marks>
```

Figure 10.3: A Sixml document describing the mark associations an SI document uses

In contrast, the SA SuperMix (introduced in Section 1.2.2) represents its SI as a Sixml document. Thus, it can supply its SI document, as is, to the packing process.

The packing process proceeds in four phases: gathering descriptors, gathering base documents, gathering SI, and packaging.

Phase 1: The packing process retrieves unique descriptors by executing a variation of the bi-level query Q3 described in Section 9.4.3.3. The query is executed over the Sixml document that the SA provides.

Phase 2: The packing process executes the bi-level query Q4 described in Section 9.4.3.4 to retrieve the list of unique base documents referenced. It then allows the SA user to choose which of the referenced base documents to include in the package. For example, the user might include base documents stored on his local file system, but leave out documents available on the web. However, a descriptor for a base document is included in the package even if the document itself is excluded. For example, in Figure 10.2, the document descriptor D7 would be included even if the user excludes its base document B10. As Section 10.4 shows, a document descriptor for an omitted document needs to be included to ensure consistency of the receiver's descriptor repository.

Phase 3: The SA provides to the packing process the path to the file that contains the SI to be interchanged. The SA and its user may include additional files in the package. For example, when packaging the Sidepad document in Figure 1.3, the user can also include the transformation shown in Figure 1.5. The packing process does not analyze the contents of these files. In the rest of this chapter, we refer to these files as *SI files*.

Phase 4: In this phase, the packing process first prepares an XML document called the *manifest* that lists the contents of the SI package. It also includes in the manifest the unique descriptors extracted in Phase 1. The packing process then bundles the manifest, the base documents selected in Phase 2, and the SI files selected in Phase 3 into a *single* SI package file. The SA user can then send this package file to other users.

An SI package file realizes an SI-dependency graph, but without the following graph elements: nodes in the Base region related to the documents excluded in Phase 2; nodes in the External region; and the edges into and within the External region. For example, the SI package file corresponding to the graph in Figure 10.2 would not include the nodes B10 and E11 and the edge incident to E11 (assuming the user omits B10; E11 is omitted because it is an external document).

10.4. Unpacking Packages

We now describe the process of unpacking an SI package. We begin with an introduction to some necessary concepts and terms.

10.4.1. Concepts and Terms

Consistent descriptors: A descriptor is *consistent* if the corresponding base part can be "activated": An application descriptor is consistent if the corresponding base application can be launched; a document descriptor is consistent if the corresponding base document can be opened in an appropriate base application; a mark descriptor is consistent if its document descriptor is consistent and the context of the subdocument the mark descriptor identifies can be accessed.

A descriptor repository is *consistent* if all descriptors in the repository are consistent.

We assume that the descriptors in an SI package come from a consistent repository and that the unpacking process updates a consistent repository. Under these conditions, the unpacking process leaves the updated repository consistent. The repository is allowed to be inconsistent during the unpacking process.

Known and New Descriptors: A descriptor in a received manifest is *known* to the receiver if a descriptor with the received descriptor's ID is in the receiver's repository. Otherwise the received descriptor is *new*. It is safe to determine a received descriptor's "newness" using its ID because the ID is globally unique (as mentioned in Section 3.2.2).

A version of a received descriptor could already be in a repository for several reasons. For example, the package might contain an updated version of a previously received SI document, and some of the descriptors previously received are received again.

A received base document is *known* if the corresponding document descriptor is known. Otherwise, the base document is *new*.

Conflicting Descriptors: A known descriptor causes a *conflict* if it is not equal to its repository counterpart. Two descriptors are *equal* if their serialized string representations are equal.

Application descriptors are unlikely to cause conflicts because a different descriptor is maintained for each version of the application. Also, changes to an application descriptor do not affect dependent descriptors (that is, descriptors that reference the application descriptor) or SI because the descriptor is always referenced by its ID.

Conflicts in document descriptors are frequently due to differences in base-document locations. For example, with the SI-dependency graph in Figure 10.2, the document descriptor D6 causes a conflict if the location of its base document (B9) in the received version is `C:\Out`, but the location is `E:\In` in the receiver's repository.

10.4.2. The Unpacking Process

The process of unpacking an SI package proceeds in four phases: adapting the manifest to suit the receiver's environment, extracting base documents, updating the receiver's descriptor repository, and extracting SI files.

Phase 1: This phase alters the *received manifest* to resolve conflicting descriptors (present in the manifest) and to reflect the receiver's choice of base-document locations. It also determines which of the received descriptors need to be added (from the manifest) to the receiver's descriptor repository. All changes to the manifest are made in memory so that the package can be reused, if necessary.

The following kinds of changes may be made to the manifest:

- Assign a new ID to a conflicting document or application descriptor, and update dependent *new descriptors* to reflect the new ID. For example, if a document descriptor is assigned a new ID, the dependent new mark descriptors are updated to reference the new document-descriptor ID.
- Remove from the manifest the descriptors for mark associations in the nested schema, because those descriptors are directly included in SI and are to be retrieved at run-time from the SI document, not from the descriptor repository. (Section 3.2.2 describes the storage choices an SA has about descriptors.)
- Remove known mark descriptors (from the manifest) because of the consistency assumption.
- Remove known application descriptors that do not cause conflict because no user action in the unpacking process can change these descriptors.
- Retain known document descriptors, even if they do not cause conflicts, because the user has several choices for a known base document: He may ignore the document, overwrite his version of the document with the version in the package, or extract the document to a new location. Note the user's choice for each base document corresponding to a document descriptor.

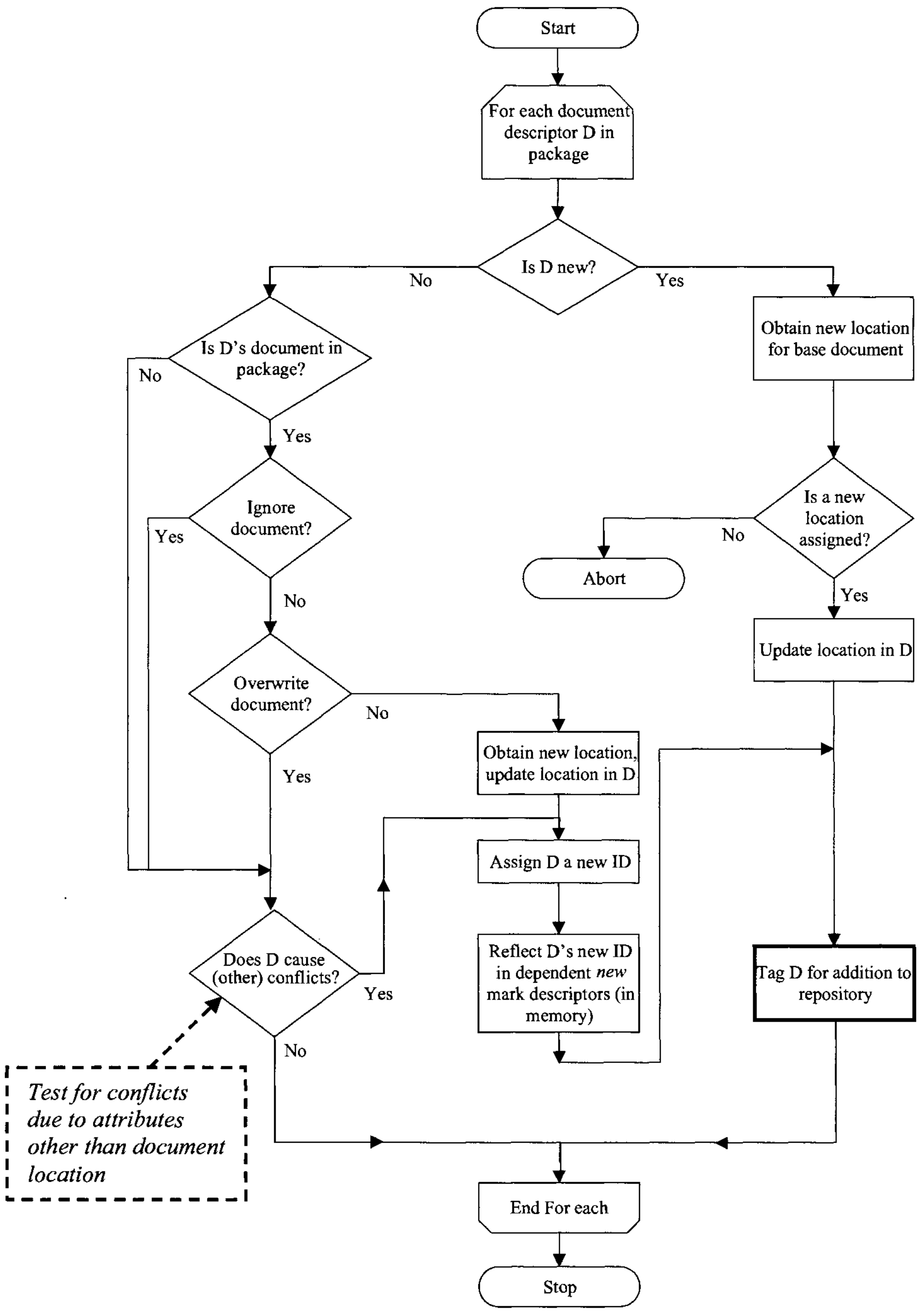

Start
For each document descriptor D in package
Is D new?
No
Yes
Is D's document in package?
No
Yes
Obtain new location for base document
Ignore document?
Yes
No
Is a new location assigned?
No
Yes
Abort
Update location in D
Overwrite document?
No
Yes
Obtain new location, update location in D
Assign D a new ID
Reflect D's new ID in dependent *new* mark descriptors (in memory)
Does D cause (other) conflicts?
Yes
No
Tag D for addition to repository
Test for conflicts due to attributes other than document location
End For each
Stop

Figure 10.4: A procedure to process document descriptors when unpacking an SI package

In effect, this phase leaves only new descriptors in the manifest and it alters conflicting known descriptors in the manifest such that they are new to the receiver's repository.

Figure 10.4 outlines the process of determining the document descriptors to add to the repository and the process of assigning a location to each base document. Each document descriptor that passes through the box labeled 'Tag D for addition to repository' (the box with thick borders) is added to the receiver's repository.

Phase 2: In this phase, each new base document and each known base document that the user chooses to extract is extracted to the location the user chooses.

Phase 3: This phase adds all descriptors left in the changed manifest to the receiver's descriptor repository.

Phase 4: This phase extracts each SI file in the package to a location the user indicates.

10.4.3. Exceptions

We now discuss two kinds of exceptions that might arise during and after the unpacking process. The unpacking process warns the user of these exceptions.

A descriptor embedded in an unpacked SI document can cause exceptions if the user extracts the descriptor's base document to a location different from that indicated in the descriptor. (The unpacking process does not alter SI.)

Assigning a new ID to a conflicting document or application descriptor (as is done in Phase 1 of the unpacking process) can cause exceptions if the SI document also references the conflicting descriptor. For example, assume the Sidepad item S2 in Figure 10.2 references the document descriptor D6 instead of referencing the mark descriptor M4. Assume D6 causes a conflict and that the conflict is resolved by assigning this descriptor the new ID D12. Now, the unpacked Sidepad item S2 would reference the repository version of descriptor D6, not the version that was received and given a new ID.

This exception is avoided if SI references a document or an application *indirectly* using the object model described in Section 3.2.4, or by using a bi-level query over a mark. However, indirectly referencing descriptors in this manner requires that the SI document also reference at least one mark in the document to be indirectly referenced. (To indirectly reference an application, the SI document must reference at least one mark in some document that uses the application.) Section 10.7 explores an alternative solution that does not have this requirement.

10.5. Evaluation

We have implemented the SI-interchange service described in this chapter in a component called *SuperPack* as an ActiveX server using Microsoft Visual Basic 6.0 [101]. The component creates package files as cabinet files using the Microsoft Cabinet Software Development Kit [94]. A *cabinet file* is a compressed archive of disk files, and is frequently used to bundle software installation files.

SuperPack is integrated into both Sidepad and SuperMix. We have used the integration to share Sidepad documents with research partners. With assistance from our research partners at Villanova University [60], we have also used SuperPack in combination with the bi-level query capability to add Sidepad documents to digital-library collections.

We have used SuperPack from SuperMix to share compositions with friends. For example, we shared the SuperMix composition described in Section 1.2.2 with friends in Germany to introduce them to traditional South Indian weddings (before the friends' arrival in India for such a wedding).

Integrating SuperPack into an SA is quite easy: The developer needs to add a reference to the SuperPack component library in his application, and call just one method (often using just one line of code) to initiate package creation. For example, the following line of Visual Basic code is used in SuperMix to package a composition:

```
SuperPack.Pack(compositionFilepath, compositionDoc)
```

The first parameter supplies the path to the disk file containing the composition to be packaged. The second parameter is a reference to the Sixml document listing the mark associations used in the composition. In this case, the second parameter is the composition document itself, because SuperMix represents a composition as a Sixml document and manipulates it using Sixml DOM. (In contrast, Sidepad needs to construct a Sixml document such as that shown in Figure 10.3 for the second parameter, because it represents a document in a proprietary format; not as a Sixml document.)

No code is required in any SA to unpack a package (because unpacking is independent of an SA). SuperPack extends the MS Windows shell so that a user can initiate the unpacking process by simply double-clicking on a package file.

We share an anecdote [122] that illustrates the flexibility and reusability of the packing and unpacking processes: Some of our collaborators were in the process of developing the SA called SIMPEL [123], and they needed us to test an early version of the application. They had created a couple of test documents in SIMPEL, but they could not package them because they had not yet integrated SuperPack into SIMPEL.

To work around the inability to interchange the test documents, our collaborators created a proxy Sidepad document using the same set of marks the test documents used, created a package file from the Sidepad document using SuperPack (already integrated into Sidepad), and sent us the package file. They also sent us the test SIMPEL documents as mail attachments. We unpacked the received Sidepad package to update our descriptor repository and to retrieve the base documents that the test SIMPEL documents referenced. We discarded the proxy Sidepad document and successfully used the test documents in SIMPEL.

Through these applications and experiences, we are convinced that SuperPack satisfies the relevant application capabilities and architectural qualities listed in Sections 1.1 and Section 2.1.3, respectively.

10.6. Related Work

We now briefly review some systems related to the SI-interchange service.

Our approach to sharing SI is similar to the 'Save As' functionality some web browsers provide, but it is also quite different from that functionality as outlined in Section 10.1. We now illustrate a limitation in the web browsers' approach to interchanging web pages.

We mentioned in Section 10.1 that web browsers alter relative URLs in a saved web page to absolute URLs, but this action can change the navigation structure of the saved page. For example, assume the page `http://pdx.edu/index.html` references the page `friends.html` using a relative URL. If a user saves the former page to his local disk, the browser changes the relative URL `friends.html` to the absolute URL `http://pdx.edu/friends.html`, possibly changing the web page author's intentions for the navigation structure: The author intends to link pages in the same folder of a computer, not pages on two different computers.

Microsoft PowerPoint includes a facility called 'Package for CD' [4] to package a presentation file and the files the presentation references (limited to two levels deep). However, this facility can package only a PowerPoint presentation, and files must be linked using the OLE linking protocol (OLE) [18]. PowerPoint does not natively support relocation of extracted files, but depends on OLE's support for relocation.

OLE provides a means of creating *compound documents* that can contain information obtained via links to parts of other documents. (Section 3.7.6 reviews OLE compound documents in detail.) Interchanging an OLE compound document requires a user to manually package the compound document and the linked documents. To use all parts of a received compound document, a receiver generally needs to recreate the sender's folder structure (including drive letters and folder names). Some OLE conventions can reduce the burden of recreating the folder structure, but the conventions generally constrain document names or document locations. For example, if a linked document is not found in the expected folder, OLE looks for the document in the folder where the compound document is saved. This convention requires each document in the package to have a distinct name.

The hypertext systems Dexter and IRIS (reviewed in Section 3.7.3) define interchange formats for hypertext data [57, 141]. In both systems, support for interchanging hypertext data essentially consists of utilities to export and import parts of a hypertext network via database dumps. Neither system considers issues such as conflicts and document locations.

10.7. Summary and Conclusions

In this chapter, we have discussed the various considerations in interchanging bi-level information among SA users and described an SA-independent service for such interchange. We have introduced the notion of SI-dependency graphs, which form the conceptual basis for interchange, and showed how an SI-package embodies an SI-dependency graph. We have also presented SuperPack, an implementation of the interchange service, and illustrated the ease with which it can be integrated into SAs.

We see scope for some improvement in usability in particular scenarios of interchanging bi-level information. For example, the current implementation works well for interactive use (that is, user-guided extraction) when the number of base documents to extract is relatively small. When the number of base documents to be extracted is large, assigning locations to base documents and generally choosing the right action when a known document is included in a package can be cumbersome. An SA or its user might wish to automate choosing the locations of base documents in this situation.

In Section 10.4, we mainly discussed conflicts due to differences in document locations, but conflicts can also occur due to differences in other attributes. For example, the receiver might employ a context agent implemented using a different technology than the sender does (for example, Java [71] instead of ActiveX [93]).

In general, a descriptor can cause conflicts when the conditions of SI use change. We call each condition or combination of conditions of SI use a *perspective*. We envision a system in which users create (or a single user creates) a number of perspectives (but probably operates from only one perspective at any time). A descriptor could then vary among perspectives, yet retain its identity, unlike the current system which gives the descriptor a new identity for each variation. For example, the entry for the context agent class in a descriptor could point to a Java library in the sender's perspective, but it could point to an ActiveX library in the receiver's perspective. With the proposed extension, the descriptor would have the same ID in either perspective.

In this chapter, we have only informally defined the notion of SI-dependency graphs. We plan to define the actions currently performed in the unpacking process as a set of formal mappings over a dependency graph produced by the packing process. Such a treatment enables us to more clearly state what properties can be guaranteed for the packing and unpacking processes, and the conditions under which those guarantees hold.

This chapter concludes the detailed description of this dissertation research. The next chapter summarizes the research and presents concluding remarks.

11. Summary, Future Work, and Conclusions

Chapters 1 through 10 have provided a detailed description of this research. This chapter summarizes the research, and discusses some future work, including two application areas we like to pursue.

11.1. Summary

We begin the chapter with a summary of the developments in the earlier chapters.

Chapter 1 introduced the notion of *superimposed information* (SI), *base information* (BI), and *superimposed application* (SA). It also presented three SAs and outlined our real-world and research objectives to support the design, development, and deployment of SAs.

Chapter 2 outlined the contributions of this research, provided an overview of our framework to meet the research objectives, and gave a summary of the evaluation of the framework components. The contributions called out were: the concept of *context information* for sub-documents, documents, and applications; the concept of *bi-level information*, which is a combination of SI and context information; mechanisms to represent, access, transform, and interchange bi-level information, and an evaluation of these mechanisms; a set of run-time services called the *Superimposed Application Shareable Services* (SASS), including architectural desiderata, an architectural reference model, and a reference implementation; and a set of *deployment guidelines* for SAs and the components of SASS. Chapters 3 through 10 described these contributions in detail (except the deployment guidelines [112], which we excluded from this dissertation for brevity).

Chapter 3 explained different ways to create, describe, and activate *marks* (which are BI references). It introduced some abstractions an SA can use to reference, activate, and retrieve context information from marks to arbitrary BI types. It also introduced an abstraction called *context agent*, which represents pluggable software wrappers used to interact with BI. These abstractions were presented as a part of *SPARCE* [110], our middleware architecture to facilitate SA development. Chapter 3 also presented an evaluation of SPARCE and the representation schemes for BI descriptors. The evaluation shows that the context-agent abstraction allows support for new BI types to be added easily and incrementally, and that the SAs, context agents, and base applications can all evolve independently.

Whereas Chapter 3 examined SI management from a software-architecture perspective, *Chapter 4* provided an information-architecture perspective. It presented a framework [113] to explicitly represent the use of marks by employing a set of conventions to augment the *Entity-Relationship* (ER) model [25]. The framework has three independent parts: a model for marks and the use of marks, a model for mark descriptors, and a model for context information. The part related to mark descriptors can express the specification of a link's endpoint in any linking technology (such as SPARCE and XPointer [167]). Each part of the framework provides a systematic way to transform a conceptual schema in the augmented ER model to logical schemas in the relational and the XML models. The chapter also introduced *Sixml*, which is "SI represented as XML" [118, 120].

Chapter 5 introduced the notion of a *bi-level query system* to help filter and transform bi-level information using queries in existing languages. It presented two alternative representation schemes—nested and normalized—for XML bi-level information, and explored how each scheme impacts query expression and execution. It also illustrated that SI-only queries deserve special attention when designing a bi-level query system. Chapter 5 also identified seven goals for a bi-level query system, and presented a strategy to meet these goals in the XML model. Chapters 6 through 9 described the different components of a bi-level query system.

Chapter 6 isolated the problem of retrieving context information from a large number of marks when executing a bi-level query, and proposed a component called the *bulk accessor* [121] as a solution. This component pools context-agent instances so that the cost of accessing base sources is amortized over the entire set of marks involved in a query. The chapter identified several pooling policies for bulk access, and provided heuristics to choose a policy based on certain data characteristics. It also described an implementation of the bulk accessor and showed experimentally that the accessor provides significant improvement over naïve methods for even a small number of marks. Several means of further improving bulk-access performance were also outlined.

Chapter 7 discussed serialization and validation considerations for Sixml data and arrived at six kinds of XML content (element, attribute, text content, CData section, comment, and processing instruction) with which marks may be associated. The chapter also described *Sixml DOM* [120], an extension of the XML Document Object Model (DOM) [34], to easily and efficiently manipulate Sixml data at run time. Two strategies to implement Sixml DOM and three implementations of Sixml DOM were presented along with an experimental evaluation of the implementations. The evaluations showed that accessing mark associations and SI using Sixml DOM requires less development effort than using DOM, and that Sixml DOM saves time when accessing mark associations. Sixml DOM can have overhead when retrieving mark associations for SI such as text content and CData sections, but it provides savings when retrieving such SI itself.

Chapter 8 introduced a means to selectively *cloak* (that is, hide) parts of data from a query processor to improve the expression and execution of certain classes of queries. The chapter presented both a formal model and an architectural reference model for a cloaking query processor. It also illustrated that the formal and architectural models are independent of applications and data models by applying the models in both bi-level and non-bi-level query settings.

Chapter 9 presented the design, implementation, and evaluation of a *bi-level navigator* called the *Sixml Navigator* [120]. A bi-level navigator supports navigation over bi-level information. The Sixml Navigator supports bi-level navigation in the nested schema over Sixml documents in either the nested schema or the normalized schema.

The Sixml Navigator is designed as an alternative to the traditional path navigator used in existing query processors. It internally represents a Sixml document as a Sixml DOM tree, thereby deriving benefits such as the use of the bulk accessor, *on demand* retrieval of mark descriptors and context information, and caching of mark descriptors and context information. The navigator performs better than the traditional navigator in many cases, and its support for cloaking improves query expression and execution.

Chapter 10 discussed the key considerations in interchanging bi-level information among SA users and described an SA-independent runtime service for such interchange. It introduced the notion of *SI-dependency graphs*, which form the conceptual basis for interchanging bi-level information, and showed how an *SI package* embodies an SI-dependency graph. The chapter also presented *SuperPack*, an implementation of the interchange service, and illustrated the ease with which the service can be integrated into SAs.

11.2. Future Work

We now briefly mention some key areas of possible improvements to our framework and describe two application prospects for the framework. We begin with the improvement areas.

Some issues related to mark robustness exist, largely due to base-layer updates after mark creation. We believe that our context-management mechanism can be useful in resolving displaced and missing marks under these circumstances, but this use needs to be verified.

Our design of SASS is independent of operating platforms, but our reference implementation is specific to the Microsoft Windows platforms [104]. We believe the design is portable to most modern operating platforms and is amenable to implementation in most modern programming languages, but we wish to verify this assertion.

We also wish to improve bi-level query execution by implementing the following strategies: push down selections over SI to possibly reduce the number of base accesses (to retrieve context information); exploit the data-management capabilities of base applications such as relational database management systems; and bind scope to query parts, instead of binding scope to entire queries.

In the rest of this section, we discuss two application areas—mash-up production and information retrieval—we wish to explore for our research framework.

11.2.1. Declaratively Producing Data Mash-ups

We see much potential for our framework in the production of data mash-ups. A *mash-up* combines information of varying granularity from disparate sources; a *data mash-up* is a document that is a mash up. (By this definition, an SA is a mash-up application, and an SI document is a data mash-up.)

In our view [120], a data mash-up has three forms: condensed, reconstituted, and formatted. A *condensed mash-up* contains references to external source fragments, but it does not yet include the actual external data. A *reconstituted mash-up* includes the external data that the condensed form specifies. A *formatted mash-up* is an alternative representation of a condensed or reconstituted mash-up. Multiple formatted mash-ups might be generated from the same condensed or reconstituted mash-up. (Each formatted mash-up might re-purpose the same information-set for a different audience.)

Our position is that Sixml, Sixml DOM, and the Sixml Navigator, together facilitate *declarative* production of the three aforementioned forms of data mash-ups. Specifically, Sixml provides a means to specify a condensed data mash-up. For example, Figure 7.1 shows a condensed form of a data mash-up with comments on a paper, represented as a Sixml document. Figure 9.13 shows two other Sixml documents that are condensed data mash-ups.

We say Sixml supports declarative specification because the SA developer uses the attributes `sixml:valueSource` and `sixml:valueExpression` to simply state that the run-time value of a mash-up part (such as text content) is obtained from external sources, without stating how the value is obtained and assigned. Section 7.4.3.4 provides the details.

Sixml DOM provides a means to programmatically create and manipulate a condensed mash-up. It also automatically reconstitutes the mash-up. For example, Figure 7.5 shows the run-time representation of a reconstituted mash-up corresponding to the condensed mash-up in Figure 7.1. Creating, manipulating, and reconstituting a mash-up with Sixml DOM is declarative because the mash-up producer does not need to specify how these actions are mapped to the underlying Sixml representation.

The Sixml Navigator, in combination with a traditional query processor, provides a means of producing a formatted mash-up. The formatting process is declarative because the mash-up producer can use a declarative query language. For example, Figure 4.27 shows a declarative query to format the condensed comment mash-up of Figure 7.1 as a web page containing author feedback. Section 9.4.2.3 discusses mash-ups formatted as maps for use with Google Maps [53] and Yahoo! Maps [179].

A class of mash-ups called enterprise mash-ups especially interests us because of the potential for a wider adoption of our framework. An *enterprise mash-up* is a mash-up of business information such as employee, customer, and order information, possibly personalized for each "user" (that is, an employee or a customer). For example, a mash-up personalized for a dispatcher in a trucking business might show his check-in time for the day (pulled from the attendance tracking system), the expected check-out time (computed from check-in time), and the pending transportation requests (obtained from order information) displayed on a map with markers at customer locations (obtained from customer information). For his personal consumption, the dispatcher might also direct the mash-up to show news headlines and the title of the most recent post by his favorite blogger.

We now discuss two key requirements, usability and scalability, in an enterprise mash-up setting. *Usability* is a requirement because an end user must be able to easily compose the mash-up and interact with it. For example, a truck dispatcher must be able to compose and personalize a mash-up without possessing programming skills.

Scalability is a key requirement for an enterprise mash-up framework because a large number of mash-ups might execute simultaneously (in an enterprise with a large user base). Also, some mash-ups can reference a large number of fragments in a large number and variety of sources.

Our approach to producing mash-ups has the potential to satisfy both the usability and the scalability requirements. Our approach can satisfy usability because we allow interactive creation of marks into heterogeneous source fragments using the familiar copy-and-paste operations. (See Section 3.1, especially Figure 3.3, for information on mark creation.) Also, in our approach, a mash-up can be easily composed as a web page. Figure 1.8 illustrates how marks can be interactively incorporated in a web page.

Our framework can satisfy the scalability requirement because the bulk accessor, Sixml DOM, and the Sixml Navigator are able to support the use of hundreds of thousands of mark associations in a single Sixml document. (Sections 6.4.2, 7.6.3, and 9.4.3, respectively provide experimental verifications of the scalability of these three components.) Producing multiple mash-ups simultaneously is also feasible because each mash-up can be produced using a separate query-processor instance. Also, our framework can share context information across Sixml documents and queries, and a single bulk-accessor instance can be employed for multiple mash-ups.

Encouraged by the possibility of a wider application of our research, we are currently examining use cases and environments to produce enterprise data mash-ups.

11.2.2. Improving the Information-Retrieval Experience

Searching (for example, web search) is a common means of finding and retrieving information, but much improvement is possible in the current search approaches. We discuss one such improvement that also provides an opportunity to apply our research. (The improvement we discuss is complementary to using *semantic components* [138], which are selected segments of base-layer text used to enhance information retrieval.)

Currently, there is a disconnection between the information a user seeks and the results a search engine returns: The user seeks just the *sub-documents* that satisfy his needs (often expressed as a set of keywords), but the search engine returns a *list of documents*. Consequently, the user *clicks through* (that is, visits or opens) each result document in an appropriate application and invokes the search function that is already built into the application to locate in the document the same keywords he has already submitted to the search engine. Then, the user clicks through each keyword occurrence and examines the context (for example, the containing paragraph) to determine sub-document relevance. (In some applications, the user might need to scroll through a document, manually identifying each occurrence of the keywords.)

Some search engines include in their results document excerpts with some keyword occurrences highlighted, but the user still needs to click through each document to locate and identify relevant sub-documents. The search functions in some applications (for example, Adobe Acrobat [8]) give the user a list of the occurrences of the searched keywords (after the user manually invokes the search function), but the user has to click through each keyword occurrence because the information in the result list is rarely sufficient to determine relevance.

The use of marks and context information can help reduce the number of click-through operations required to determine the relevance of documents and sub-documents [116]. For example, document search engines can be enhanced to return a mark to each sub-document that contains (or is otherwise related to) the searched keywords. The user can then examine the context information for a mark without activating the marks. He can also use the mark to directly navigate to the sub-document, alleviating the need to locate the keywords again within a document.

Similarly, search functions within applications can also be enhanced to return marks and let users explore context information without the user clicking through each keyword occurrence.

To evaluate the feasibility of using our framework to improve information-retrieval experience, we have implemented a prototype solution (called *SuperSearch*) using our framework. This prototype extends the functionality of two popular (third-party) search engines to let the user view sub-documents that match keywords, examine the context information for any result sub-document, and navigate directly to a sub-document. We are also in the process of extending search functions in a few popular applications to provide functionality similar to that in our search-engine extension.

The key challenge in this application area is ranking and aggregating sub-documents, because several sub-documents might relate to the searched keywords. There can also be a need to transfer marks from one document type to another. For example, an extended search engine might return marks to sub-documents of a Microsoft Word document [96], but the user might not have installed this application. To assist the user in such situations, search engines typically provide result documents in HTML format [61]. In this case, we would need to transfer each MS Word mark in the result to an HTML mark.

11.3. Conclusions

With the concepts, frameworks, components, and applications described in these chapters, we have successfully met both our real-world and research objectives. Specifically, our framework helps SA developers support the seven application capabilities listed in Section 1.1, and the runtime system, SASS, satisfies the architectural desiderata listed in Section 2.1.3.

Our research facilitates a wide range of applications as illustrated throughout this dissertation. Chapter 1 introduced *Sidepad*, a scratch pad tool that lets a user collect and organize information fragments in a nested model; a multi-media composer and player called *SuperMix*; and a word-processor-style application called the *HTML+M Editor*. Chapter 4 introduced the Superimposed System-Information Browser (*SSIB*) that lets a computer system administrator view and query information such as event logs and operating-system updates; and the Superimposed Scholarly Review System (*SISRS*) to facilitate reviewing of documents and generation of artifacts such as author feedback. Section 11.2 introduced the use of our research in enterprise data mash-ups and in information retrieval. Chapter 3 mentions applications developed by others using our research framework.

The aforementioned SAs satisfy different user goals but they all use the same runtime system. Instances of these applications (including multiple instances of the same SA) can simultaneously use our runtime services on the same computer. Also, there can be more than one simultaneous instance of any of our runtime services.

The aforementioned SAs also differ in the different data models and schemas employed. For example, Sidepad uses a proprietary data model, SuperMix uses the XML model, the HTML+M Editor uses HTML, and we have used SSIB and SISRS in both the relational and XML models. Regardless of these data model and schema differences, the information in each application is conceptually modeled using our framework and logical schemas are generated in either the relational or the XML model.

In general, an SA is free to represent its information in any data model. Bi-level information derived from the SI represented in (or SI that can be mapped to) the relational or XML model can be transformed via queries in existing languages, using existing query processors.

This chapter concludes the main body of this dissertation. The rest of the dissertation provides supplementary information in the form of appendices.

Bibliography

1. Abiteboul, S., Cluet, S., Milo, T. 1993. Querying and Updating the File. In *Proceedings of 19th International Conference on Very Large Data Bases (VLDB'93)*, Aug. 24-27, Dublin, Ireland.
2. Abiteboul, S., Hull, R., Vianu, V. 1995. *Foundations of Databases*. 1st edition. Addison-Wesley.
3. Abiteboul, S., Benjelloun, O., Cautis, B., Manolescu, I., Milo, T., Preda, N. 2004. Lazy Query Evaluation for Active XML. In *Proceedings of ACM SIGMOD International Conference on Management of Data (SIGMOD'04)*, Paris, France.
4. About Packaging and Copying a Presentation to CD. Microsoft Corporation. Available from http://office.microsoft.com/en-us/powerpoint/HP052727561033.aspx. Accessed Jan. 25, 2008.
5. ACM SIGMOD Record. ACM SIGMOD. Available from http://www.sigmod.org/sigmod/record/xml/index.html. Accessed Jan. 25, 2008.
6. Adobe. Portable Document Format. Adobe Systems, Inc. Available from http://partners.adobe.com/public/developer/pdf/index_reference.html. Accessed Jan. 25, 2008.
7. Adobe. Acrobat Interapplication Communication Overview. Adobe Systems Inc. Available from http://partners.adobe.com/public/developer/en/acrobat/sdk/pdf/iac/IACOverview.pdf. Accessed Jan. 25, 2008.
8. Adobe Acrobat. Adobe Systems Inc. Available from http://www.adobe.com/products/acrobat. Accessed Jan. 25, 2008.
9. Amaya. W3C. Available from http://www.w3.org/Amaya/. Accessed Jan. 25, 2008.
10. Archer, D., Delcambre, L. 2006. Capturing and Reusing Human Attention in Corporate Decision Making. In *Proceedings of International ACM Workshop on Contextualized Attention Metadata: Collecting, Managing and Exploiting of Rich Usage Information*, Nov. 11, Arlington, Virginia.
11. Archer, D., Delcambre, L., Corubolo, F., Cassel, L., Price, S., Murthy, U., Maier, D., Fox, E. A., Murthy, S., McCall, J., Kuchibotla, K., Suryavanshi, R. 2008. Superimposed Information Architecture for Digital Libraries. In *Proceedings of European Conference on Research and Advanced Technology for Digital Libraries*, Sep. 14-18, Aarhus, Denmark.
12. Arenas, M., Libkin, L. 2002. A Normal Form for XML Documents. In *Proceedings of 21st ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, Jun. 3-5, Madison, Wisconsin.

13. Bass, L., Clements, P., Kazman, R. 1998. *Software Architecture in Practice*. Addison-Wesley.

14. Berners-Lee, T. 1994. Uniform Resource Locators (URL). IETF. Available from http://www.ietf.org/rfc/rfc1738.txt. Accessed Jan. 25, 2008.

15. Berners-Lee, T., Fielding, R., Masinter, L. 2005. Uniform Resource Identifier (URI): Generic Syntax. IETF. Available from http://www.ietf.org/rfc/rfc3986.txt. Accessed Jan. 25, 2008.

16. Booker, P. S. K., Granger, R. K., Guest, E. J., Norton, S. A., Price, J. E., Glaser, H. 1999. Software Agents and their Use in Mobile Computing. Report# DSSE-TR-99-5. Dept. of Electronics & Computer Science, University of Southampton.

17. Bowers, S., Delcambre, L., Maier, D. 2002. Superimposed Schematics: Introducing E-R Structure for In-Situ Information Selections. In *Proceedings of 21st International Conference on Conceptual Modeling (ER'02)*, Oct. 7-11, Tampere, Finland.

18. Brockschmidt, K. 1994. *Inside OLE 2*. Microsoft Press.

19. Buneman, P., Naqvi, S., Tannen, V., Wong, L. 1995. *Principles of Programming with Complex Objects and Collection Types*. Berlin, Germany. Elsevier Science Publishers B.V.

20. Buneman, P., Cheney, J., Vansummeren, S. 2007. On the Expressiveness of Implicit Provenance in Query and Update Languages. In *Proceedings of 11th International Conference on Database Theory (ICDT'07)* Jan. 10-12, Barcelona, Spain.

21. Buneman, P., Tan, W. C. 2007. Provenance in Databases. In *Proceedings of ACM SIGMOD International Conference on Management of Data (SIGMOD'07)*, Jun. 11-14, Beijing, China.

22. Bush, V. 1945. As We May Think. *The Atlantic Monthly*, July 1945.

23. C# Programming Guide. Microsoft Corporation. Available from http://msdn2.microsoft.com/en-us/library/67ef8sbd(VS.80).aspx. Accessed Oct. 27, 2007.

24. Casanova, M. A., Tucherman, L., Lima, M. J. D., Netto, J. L. R., Rodriguez, N. R., Soares, L. F. G. 1991. The Nested Context Model for Hyperdocuments. In *Proceedings of Hypertext 1991*, San Antonio, Texas.

25. Chen, P. P. 1976. The Entity-Relationship Model – Towards a Unified View of Data. *ACM Transactions on Database Systems* 1(1), 9-36.

26. Conklin, J. 1987. Hypertext: An Introduction and Survey. *IEEE Computer* 20(9), 17-41.

27. Consens, M. P., Milo, T. 1994. Optimizing Queries on Files. In *Proceedings of ACM SIGMOD International Conference on Management of Data (SIGMOD'94)*, May 24-27, Minneapolis, Minnesota.

28. Cormen, T. H., Leiserson, C. E., Rivest, R. L., Stein, C. 2001. *Introduction to Algorithms*. 2nd edition. Cambridge, Massachussets. The MIT Press.

29. Cysneiros, L. M., Leite, J. C., Neto, J. M. 2001. A Framework for Integrating Non-Functional Requirements into Conceptual Models. *Requirements Engineering* 6(2), 97-115.

30. Dayal, U., Lomet, D., Alonso, G., Lohman, G., Kersten, M., Cha, S. 2006. VLDB 2006 - A Word from the PC Chair. Available from http://aitrc.kaist.ac.kr/~vldb06/intro_PC_chair.html. Accessed Jan. 25, 2008.

31. Delcambre, L., Maier, D., Reddy, R., Anderson, L. 1997. Structured Maps: Modeling Explicit Semantics over a Universe of Information. *International Journal on Digital Libraries* 1(1), 20-35.

32. Delcambre, L., Maier, D., Bowers, S., Weaver, M., Deng, L., Gorman, P., Ash, J., Lavelle, M., Lyman, J. 2001. Bundles in Captivity: An Application of Superimposed Information. In *Proceedings of 17th International Conference on Data Engineering (ICDE'01)*, Apr. 2-6, Heidelberg, Germany.

33. Delcambre, L. 2006. Personal Communication. Sep. 12.

34. Document Object Model. W3C. Available from http://www.w3.org/DOM. Accessed Jan. 25, 2008.

35. Document Object Model (DOM) Level 1 Specification. 1998. W3C. Available from http://www.w3.org/TR/REC-DOM-Level-1. Accessed Jan. 25, 2008.

36. Document Object Model (DOM) Level 2 Core Specification. 2000. W3C. Available from http://www.w3.org/TR/DOM-Level-2-Core. Accessed Jan. 25, 2008.

37. Document Object Model (DOM) Level 3 Core Specification. 2004. W3C. Available from http://www.w3.org/TR/DOM-Level-3-Core. Accessed Jan. 25, 2008.

38. Document Object Model (DOM) Level 3 Load and Save Specification. 2004. W3C. Available from http://www.w3.org/TR/DOM-Level-3-LS. Accessed Jan. 25, 2008.

39. Document Object Model for MathML. 2003. W3C. Available from http://www.w3.org/TR/MathML2/appendixd.html. Accessed Jan. 25, 2008.

40. Dyreson, C. E., Bohlen, M. H., Jensen, C. S. 2001. METAXPath. In *Proceedings of International Conference on Dublin Core and Metadata Applications*, Oct. 22-26, Tokyo, Japan.

41. Elmasri, R., Navathe, S. B. 2003. *Fundamentals of Database Systems*. 4th edition. Addison-Wesley.

42. Elmasri, R., Li, Q., Fu, J., Wu, Y., Hojabri, B., Ande, S. 2005. Conceptual Modeling for Customized XML Schemas. *Data and Knowledge Engineering* 54(1), 57-76.

43. Extensible Markup Language (XML) 1.0. 2006. W3C. Available from http://www.w3.org/TR/xml. Accessed Jan. 25, 2008.

44. Feinberg, G. 1967. Possibility of Faster-Than-Light Particles. *Physical Review* 159(5), 1089-1105.

45. Fielding, R., Gettys, J., Mogul, J., Frystyk, H., Masinter, L., Leach, P., Berners-Lee, T. 1999. Hypertext Transfer Protocol – HTTP/1.1. IETF. Available from http://www.ietf.org/rfc/rfc2616.txt. Accessed Jan. 25, 2008.

46. Firefox. Mozilla. Available from http://www.mozilla.com/firefox/. Accessed Jan. 25, 2008.

47. Gamma, E., Helm, R., Johnson, R., Vlissides, J. 1995. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley.

48. Garzotto, F., Mainetti, L., Paolini, P. 1993. HDM2: Extending the E-R Approach to Hypermedia Application Design. In *Proceedings of 12th International Conference on the Entity-Relationship Approach (ER'93)*, Dec. 15-17, Arlington, Texas.

49. Garzotto, F., Paolini, P., Schwabe, D. 1993. HDM – A Model-based Approach to Hypertext Application Design. *ACM Transactions on Information Systems* 11(1), 1-26.

50. Geerts, F., Kementsietsidis, A., Milano, D. 2006. MONDRIAN: Annotating and Querying Databases through Colors and Blocks. In *Proceedings of 22nd International Conference on Data Engineering (ICDE'06)*, Apr. 3-7, Atlanta, Georgia.

51. A Gentle Introduction to SGML. Available from http://www.isgmlug.org/sgmlhelp/g-index.htm. Accessed Jan. 25, 2008.

52. Google Earth API. 1. Google. Available from http://code.google.com/apis/earth/. Accessed Jan. 25, 2008.

53. Google Maps API. 1. Google. Available from http://www.google.com/apis/maps/. Accessed Jan. 25, 2008.

54. Gopalakrishna, D. S. 2006. Personal Communication. Jan. 30.

55. Haan, B. J., Kahn, P., Riley, V. A., Coombs, J. H., Meyrowitz, N. K. 1992. IRIS Hypermedia Services. *Communications of the ACM* 35(1), 36-51.

56. Halasz, F. G., Moran, T. P., Trigg, R. H. 1987. NoteCards in a Nutshell. *ACM SIGCHI Bulletin* 17(SI), 45-52.

57. Halasz, F. G., Schwartz, F. 1994. The Dexter Hypertext Reference Model. *Communications of the ACM* 37(2), 30-39.

58. Hardman, L., Bulterman, D. C. A., Rossum, G. 1994. The Amsterdam Hypermedia Model: Adding Time and Context to the Dexter Model. *Communications of the ACM* 37(2), 50-62.

59. Hlousek, P. 2005. XPath 2.0: It Can Sort! In *Proceedings of 2nd International Workshop on XQuery Implementation, Experience and Perspectives (XIME-P'05)*, Jun. 16-17, Baltimore, Maryland.

60. Home Page of Dr. Lillian (Boots) Cassel. Available from http://csc.villanova.edu/faculty/lillian.cassel. Accessed Feb. 6, 2008.

61. Hypertext Markup Language (HTML). W3C. Available from http://www.w3.org/MarkUp. Accessed Jan. 25, 2008.

62. IBM DB2 Database for Linux, UNIX, and Windows. 9. IBM Corporation. Available from http://publib.boulder.ibm.com/infocenter/db2luw/v9. Accessed Jan. 25, 2008.

63. IHMC CmapTools. 2006. Institute for Human and Machine Cognition. Available from http://cmap.ihmc.us. Accessed Jan. 25, 2008.

64. Information Processing – Hypermedia/Time-based Structuring Language (HyTime). 1997. Available from http://www1.y12.doe.gov/capabilities/sgml/wg8/document/n1920/. Accessed Jan. 25, 2008.

65. Intel Core Duo processor. Intel Corporation. Available from http://www.intel.com/support/processors/mobile/coreduo/. Accessed Nov. 25, 2007.

66. ISO 8879: Information Processing – Text and Office Systems – Standard Generalized Markup Language (SGML). 1986. ISO.

67. ISO/IEC 13250 Topic Navigation Maps. 1998. ISO. Available from http://www.ornl.gov/sgml/sc34/document/0008.htm. Accessed Jan. 25, 2008.

68. ISO/IEC JTC 1/SC34 Topic Maps. 1999. ISO. Available from http://www.ornl.gov/sgml/sc34/document/0058.htm. Accessed Jan. 25, 2008.

69. ISO/IEC JTC 1/SC 34 Topic Maps – XML Syntax. 2004. ISO. Available from http://www.jtc1sc34.org/repository/0495.htm. Accessed Jan. 25, 2008.

70. Jagadish, H. V., Lakshmanan, L. V. S., Scannapieco, M., Srivastava, D., Wiwatwattana, N. 2004. Colorful XML: One Hierarchy Isn't Enough. In *Proceedings of ACM SIGMOD International Conference on Management of Data (SIGMOD'04)*, Jun. 13-18, Paris, France.

71. Java Technology. Sun Microsystems. Available from http://java.sun.com. Accessed Jan. 25, 2008.

72. Javadoc Tool. Sun Microsystems. Available from http://java.sun.com/j2se/javadoc/. Accessed Jan. 25, 2008.

73. JavaScript. Mozilla Foundation. Available from http://developer.mozilla.org/en/docs/JavaScript. Accessed Jan. 25, 2008.

74. Jeswin, P. Xml Performance: XmlMark revisited: Java, Mono and .Net. Available from http://www.process64.com/articles/xmlmark1/. Accessed Oct. 29, 2007.

75. Jhingran, A. 2006. Enterprise Information Mashups: Integrating Information, Simply. In *Proceedings of 32nd International Conference on Very Large Data Bases (VLDB'06)*, September 12-15, Seoul, Korea.

76. Jian, J., Su, H., Rundensteiner, E. A. 2003. Automaton Meets Query Algebra: Towards a Unified Model for XQuery Evaluation over XML Data Streams. In *Proceedings of 22nd International Conference on Conceptual Modeling (ER'03)*, Oct. 13-16, Chicago, Illinois.

77. Josefsson, S. 2006. The Base16, Base32, and Base64 Data Encodings. IETF. Available from http://www.ietf.org/rfc/rfc4648.txt. Accessed Jan. 25, 2008.

78. Kahan, J., Koivunen, M., Prud'Hommeaux, E., Swick, R. R. 2001. Annotea: An Open RDF Infrastructure for Shared Web Annotations. In *Proceedings of 10th International World Wide Web Conference (WWW'01)*, May 1-5, Hong Kong.

79. Kannada-English Transliteration. Baraha. Available from http://www.baraha.com/html_help/sdk_docs/kantrans_eng.htm. Accessed Sep. 11, 2006.

80. Kay, M. H. 2001. *XSLT Programmer's Reference*. 2nd edition. Birmingham, UK. Wrox Press Ltd.

81. Kay, M. H. 2004. *XSLT 2.0 Programmer's Reference*. 3rd edition. Indianapolis, IN. Wiley Publishing, Inc.

82. Kleiner, C., Lipeck, U. W. 2001. Automatic Generation of XML DTDs from Conceptual Database Schemas. In *Proceedings of GI Jahrestagung 2001*, Sep. 25-28, Vienna, Austria.

83. Kruchten, N. Context. Available from http://nicolas.kruchten.com/context.html. Accessed Jan. 15, 2008.

84. Kumar, A. 2001. Third Voice Trails Off.... Wired News. Available from http://www.wired.com/news/business/0,1367,42803,00.html. Accessed Jan. 25, 2008.

85. Lakshmanan, L. V. S., Sadri, F., Subramanian, S. N. 2001. SchemaSQL: An Extension to SQL for Multidatabase Interoperability. *ACM Transactions on Database Systems* 26(4), 476-519.

86. LeFevre, K., Agrawal, R., Ercegovac, V., Ramakrishnan, R., Xu, Y., DeWitt, D. J. 2004. Limiting Disclosure in Hippocratic Databases. In *Proceedings of 30th International Conference on Very Large Data Bases (VLDB'04)*, Aug. 31-Sep. 3, Toronto, Canada.

87. Madhavan, J., Jeffery, S. R., Cohen, S., Dong, X., Ko, D., Yu, C., Halevy, A. 2007. Web-scale Data Integration: You can only afford to Pay As You Go. In

Proceedings of 3rd Biennial Conference on Innovative Data Systems Research (CIDR'07), January 7-10, 2007, Asilomar, California.

88. Maier, D., Delcambre, L. 1999. Superimposed Information for the Internet. In *Proceedings of ACM SIGMOD Workshop on the Web and Databases*, Jun. 3-4, Philadelphia, Pennsylvania.
89. Maier, D., Shapiro, L. 2005. CS 386/586 Introduction to Databases, Spring 2005. Portland State University. Available from http://web.cecs.pdx.edu/~len/386/. Accessed May 08, 2005.
90. Making Component-Based Systems Scale with BEA Tuxedo® CORBA. 2002. Report# CWP0434E0702-1A. BEA Systems, Inc.
91. McEveety, V. 1966. Balance of Terror. In *Star Trek Season 1, Episode 14*: Paramount Pictures.
92. Melton, J., Simon, A. R. 2001. *SQL: 1999: Understanding Relational Language Components*. 2nd edition. Morgan Kaufmann.
93. Microsoft. 1995. *COM: The Component Object Model Specification*. Microsoft Corporation.
94. Microsoft Cabinet Software Development Kit. Microsoft Corporation. Available from http://support.microsoft.com/kb/310618. Accessed Jan. 19, 2008.
95. Microsoft Internet Explorer. Microsoft Corporation. Available from http://www.microsoft.com/windows/ie. Accessed Jan. 25, 2008.
96. Microsoft Office. Microsoft Corporation. Available from http://office.microsoft.com. Accessed Jan. 25, 2008.
97. Microsoft Office Developer Center. Microsoft Corporation. Available from http://msdn.microsoft.com/office/. Accessed Jan. 25, 2008.
98. Microsoft OLE DB. Microsoft Corporation. Available from http://msdn2.microsoft.com/en-us/library/ms722784.aspx. Accessed Jan. 25, 2008.
99. Microsoft SQL Server. Microsoft Corporation. Available from http://www.microsoft.com/sql. Accessed Jan. 25, 2008.
100. Microsoft Support Web Site. Microsoft Corporation. Available from http://support.microsoft.com/. Accessed Jan. 25, 2008.
101. Microsoft Visual Basic 6.0. Microsoft Corporation. Available from http://msdn2.microsoft.com/en-us/vbrun. Accessed Jan. 25, 2008.
102. Microsoft Visual Studio. Microsoft Corporation. Available from http://msdn2.microsoft.com/en-us/vstudio/default.aspx. Accessed Nov. 25, 2007.
103. Microsoft Windows Media Player. Microsoft Corporation. Available from http://www.microsoft.com/windows/windowsmedia. Accessed Jan. 25, 2008.

104. Microsoft Windows XP. Microsoft Corporation. Available from http://www.microsoft.com/windows/products/windowsxp/default.mspx. Accessed Nov. 25, 2007.

105. Microsoft Word Visual Basic Reference. Microsoft Corporation. Available from http://www.msdn.microsoft.com/library/. Accessed Jan. 25, 2008.

106. Mono. Mono Project. Available from http://www.mono-project.com/. Accessed Jan. 25, 2008.

107. MS XML 4.0 Software Development Kit. Microsoft Corporation. Available from http://www.microsoft.com/downloads/details.aspx?FamilyID=3144b72b-b4f2-46da-b4b6-c5d7485f2b42. Accessed Jan. 25, 2008.

108. MSDN Library Archive. Microsoft Corporation. Available from http://msdn.microsoft.com/archive. Accessed Jan. 25, 2008.

109. Murthy, S., Maier, D. 2004. SISRS: The Superimposed Scholarly Review System. Available from http://sparce.cs.pdx.edu/pubs/SISRS-WP.pdf. Accessed Jan. 25, 2008.

110. Murthy, S., Maier, D., Delcambre, L., Bowers, S. 2004. Putting Integrated Information in Context: Superimposing Conceptual Models with SPARCE. In *Proceedings of 1st Asia-Pacific Conference of Conceptual Modeling*, Jan. 22, Dunedin, New Zealand.

111. Murthy, S. 2005. Sidepad User Guide. Available from http://sparce.cs.pdx.edu/apps/Sidepad/userguide. Accessed Jan. 25, 2008.

112. Murthy, S., Maier, D., Delcambre, L. 2005. Distribution Alternatives for Superimposed Information Services in Digital Libraries. In *Peer-to-Peer, Grid, and Service-Orientation in Digital Library Architectures*, edited by Türker, C., Agosti, M., and Schek, H. Springer.

113. Murthy, S., Delcambre, L., Maier, D. 2006. Explicitly Representing Superimposed Information in a Conceptual Model. In *Proceedings of 25th International Conference on Conceptual Modeling (ER'06)*, Nov. 6-9, Tucson, Arizona.

114. Murthy, S., Maier, D. 2006. A Framework for Relationship Pattern Languages. Report# TR-08-03. Department of Computer Science, Portland State University.

115. Murthy, S., Maier, D., Delcambre, L. 2006. Mash-o-matic. In *Proceedings of 6th ACM Symposium on Document Engineering*, Oct. 10-13, Amsterdam, Netherlands.

116. Murthy, S., Murthy, U., Fox, E. A. 2006. Using Superimposed and Context Information to Find and Re-find Sub-documents. In *Proceedings of Personal Information Management 2006*, Aug. 10-11, Seattle, Washington.

117. Murthy, S. 2007. Sixml DOM. Available from http://dom.sixml.org. Accessed Jan. 25, 2008.

118. Murthy, S. 2007. Sixml.org. Available from http://www.sixml.org. Accessed Jan. 25, 2008.

119. Murthy, S. 2007. Sixml Schema. Available from http://schema.sixml.org. Accessed Jan. 25, 2008.

120. Murthy, S., Maier, D. 2008. Declaratively Producing Data Mash-ups. In *Proceedings of 14th International Conference on Management of Data (COMAD'08)*, Dec. 17-19, Mumbai, India.

121. Murthy, S., Maier, D., Delcambre, L. 2008. Speeding up On-the-Fly Integration of DB and Exo-DB Data. In *Proceedings of Workshop on Information Integration Methods, Architectures, and Systems*, Apr. 11-12, Cancun, Mexico.

122. Murthy, U., Ahuja, K. 2005. Personal Communication. Nov. 27.

123. Murthy, U., Ahuja, K., Murthy, S., Fox, E. A. 2006. SIMPEL: A Superimposed Multimedia Presentation Editor and Player. In *Proceedings of 6th ACM/IEEE-CS Joint Conference on Digital Libraries (JCDL'06)*, Jun. 11-15, Chapel Hill, North Carolina.

124. Murthy, U., Fox, E. A., Delcambre, L. 2006. Enhancing Concept Mapping Tools Below and Above to Facilitate the Use of Superimposed Information. In *Proceedings of 2nd International Conference on Concept Mapping*, Sep. 5-8, San Jose, Costa Rica.

125. Namespaces in XML 1.0. 2006. W3C. Available from http://www.w3.org/TR/xml-names. Accessed Jan. 25, 2008.

126. Nelson, T. H. 1965. A File Structure for the Complex, the Changing and the Indeterminate. In *Proceedings of ACM 20th National Conference*, Aug. 24-26, Cleveland, Ohio.

127. Nelson, T. H. 1999. Xanalogical Structure, Needed Now More than Ever: Parallel Documents, Deep Links to Content, Deep Versioning, and Deep Re-Use. *ACM Computing Surveys* 31(4)

128. .NET Common Language Runtime. Microsoft Corporation. Available from http://msdn2.microsoft.com/en-us/library/8bs2ecf4(VS.71).aspx. Accessed Nov. 25, 2007.

129. .NET Framework Developer Center. Microsoft Corporation. Available from http://msdn.microsoft.com/netframework. Accessed Jan. 25, 2008.

130. *OLE Automation Programmer's Reference*. 1996. Microsoft Press.

131. OMG IDL Syntax and Semantics. 2004. In *CORBA 3.0*. Object Management Group, Inc.

132. OpenDoc Design Team. 1994. The OpenDoc Technical Summary. In *Proceedings of Apple World Wide Developers Conference*, Apr. 14, San Jose, California.

133. Parent, C., Spaccapietra, S. 1998. Issues and Approaches of Database Integration. *Communications of ACM* 41(5es), 166-178.

134. PDFBox. Available from http://www.pdfbox.org. Accessed Jan. 25, 2008.

135. Phelps, T. A. 1998. Multivalent Documents: Anytime, Anywhere, Any Type, Every Way User-Improvable Digital Documents and Systems. Report# UCB/CSD-98-1026. University of California, Berkley.

136. Phelps, T. A., Wilensky, R. 2000. Robust Intra-document Locations. In *Proceedings of 9th International World Wide Web Conference (WWW'00)*, May 15-19, Amsterdam, Netherlands.

137. Phelps, T. A., Wilensky, R. 2000. Multivalent Documents. *Communications of the ACM* 43(6), 83-90.

138. Price, S. L., Nielsen, M. L., Delcambre, L. M. L., Vedsted, P. 2007. Semantic Components Enhance Retrieval of Domain-specific Documents. In *Proceedings of 16th ACM Conference on Information and Knowledge Management (CIKM'07)*, Nov. 6-9, Lisbon, Portugal.

139. Ramakrishnan, R., Gehrke, J. 2003. *Database Management Systems*. 3rd edition. McGraw Hill.

140. Resource Description Framework (RDF). W3C. Available from http://www.w3.org/RDF/. Accessed Jan. 25, 2008.

141. Riley, V. 1990. An Interchange Format for Hypertext Systems: The Intermedia Model. In *Proceedings of Hypertext Standardization Workshop*, Jan. 16-18, Gaithersburg, Maryland.

142. Runapongsa, K., Patel, J. M., Jagadish, H. V., Chen, Y., Al-Khalifa, S. 2006. The Michigan Benchmark: Towards XML Query Performance Diagnostics. *Information Systems* 31(2), 73-97.

143. Schmidt, A., Waas, F., Kersten, M., Carey, M. J., Manolescu, I., Busse, R. 2002. XMark: a Benchmark for XML Data Management. In *Proceedings of 28th International Conference on Very Large Data Bases (VLDB'02)*, Hong Kong, China.

144. Sengupta, A., Mohan, S., Doshi, R. 2003. XER - Extensible Entity Relationship Modeling. In *Proceedings of XML Conference and Exhibition 2003*, Dec. 7-12, Philadelphia, Pennsylvania.

145. Simmen, D. E., Altinel, M., Markl, V., Padmanabhan, S., Singh, A. 2008. Damia: Data Mashups for Intranet Applications. In *Proceedings of ACM SIGMOD International Conference on Management of Data (SIGMOD'08)*, Jun. 9-12, Vancouver, Canada.

146. Spivey, J. M. 1989. *The Z Notation: A Reference Manual*. Prentice-Hall, Inc.

147. SQL Server Books Online. Microsoft Corporation. Available from http://msdn.microsoft.com/library/. Accessed Jan. 25, 2008.

148. Standard ECMA-334 C# Language Specification. 2006. ECMA. Available from http://www.ecma-international.org/publications/standards/Ecma-334.htm. Accessed Jan. 25, 2008.

149. Star Trek. Paramount Pictures. Available from http://www.startrek.com/. Accessed Jan. 25, 2008.

150. Stillger, M., Lohman, G. M., Markl, V., Kandil, M. 2001. LEO - DB2's LEarning Optimizer. In *Proceedings of 27th International Conference on Very Large Data Bases (VLDB'01)*, Sep. 11-14, Rome, Italy.

151. SVG Document Object Model. 2003. W3C. Available from http://www.w3.org/TR/SVG/svgdom.html. Accessed Jan. 25, 2008.

152. Tanaka, A. K., Navathe, S. B., Chakravarthy, S., Karlapalem, K. 1991. ER-R: An Enhanced ER Model with Situation-Action Rules to Capture Application Semantics. In *Proceedings of 10th International Conference on Entity-Relationship Approach (ER'91)*, Oct. 23-25, San Mateo, California.

153. Terwilliger, J. T. 2006. Personal Communication. Aug. 13.

154. Terwilliger, J. T., Delcambre, L., Logan, J. 2006. The User Interface is the Conceptual Model. In *Proceedings of 25th International Conference on Conceptual Modeling (ER'06)*, Nov. 6-9, Tuscon, Arizona.

155. Terwilliger, J. T. 2007. Personal Communication. Jan. 24.

156. The Active XML Team. 2003. Active XML Primer. Gemo. Available from ftp://ftp.inria.fr/INRIA/Projects/gemo/gemo/GemoReport-307.pdf. Accessed Jan. 25, 2008.

157. The Unicode Consortium. 2007. *The Unicode Standard Version 5.0.0.* Addison-Wesley.

158. Topic Navigation Maps – An Overview. International SGML/XML Users' Group. Available from http://www.isgmlug.org/n3-4/n3-4-15.htm. Accessed Jan. 25, 2008.

159. Unified Modeling Language (UML) Version 2.0. 2004. OMG. Available from http://www.omg.org/technology/documents/formal/uml.htm. Accessed Jan. 25, 2008.

160. Visual Basic Scripting Edition. Microsoft Corporation. Available from http://msdn2.microsoft.com/en-us/library/t0aew7h6.aspx. Accessed Jan. 25, 2008.

161. Web Services. W3C. Available from http://www.w3.org/2002/ws/. Accessed Jul. 12, 2007.

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

163. The World Wide Web Consortium (W3C). Available from http://www.w3.org/. Accessed Jan. 25, 2008.

164. XML Linking Language (XLink) Version 1.0. 2001. W3C. Available from http://www.w3.org/TR/xlink/. Accessed Jan. 25, 2008.

165. XML Path Language (XPath) 2.0. 2007. W3C. Available from http://www.w3.org/TR/xpath20/. Accessed.
166. XML Path Language (XPath) Version 1.0. 1999. W3C. Available from http://www.w3.org/TR/xpath. Accessed Jan. 25, 2008.
167. XML Pointer Language (XPointer). 2002. W3C. Available from http://www.w3.org/TR/xptr. Accessed Jan. 25, 2008.
168. XML Pointer Language (XPointer) Framework. 2003. W3C. Available from http://www.w3.org/TR/xptr-framework/. Accessed Jan. 25, 2008.
169. XML Pointer, XML Base and XML Linking. 2005. W3C. Available from http://www.w3.org/XML/Linking. Accessed Jan. 25, 2008.
170. XML Schema. 2001. W3C. Available from http://www.w3.org/XML/Schema. Accessed Jan. 25, 2008.
171. XML Schema Part 0: Primer Second Edition. 2004. W3C. Available from http://www.w3.org/TR/xmlschema-0/. Accessed Aug. 15, 2007.
172. XmlNode Class. Microsoft Corporation. Available from http://msdn2.microsoft.com/en-us/library/system.xml.xmlnode.aspx. Accessed Jan. 25, 2008.
173. XPointer element() Scheme. 2002. W3C. Available from http://www.w3.org/TR/xptr-element/. Accessed Jan. 25, 2008.
174. XPointer xpointer() Scheme. 2002. W3C. Available from http://www.w3.org/TR/xptr-xpointer/. Accessed Jan. 25, 2008.
175. XQuery 1.0 and XPath 2.0 Data Model (XDM). 2007. W3C. Available from http://www.w3.org/TR/xpath-datamodel/. Accessed Jan. 25, 2008.
176. XQuery 1.0: An XML Query Language. 2005. W3C. Available from http://www.w3.org/TR/xquery/. Accessed Jan. 25, 2008.
177. XSL Transformations (XSLT). 1999. W3C. Available from http://www.w3.org/TR/xslt. Accessed Jan. 25, 2008.
178. XSL Transformations (XSLT) Version 2.0. 2007. W3C. Available from http://www.w3.org/TR/xslt20/. Accessed Jan. 25, 2008.
179. Yahoo! Maps Web Services. Yahoo! Available from http://developer.yahoo.com/maps. Accessed Jan. 25, 2008.
180. Yahoo! Pipes. Yahoo! Inc. Available from http://pipes.yahoo.com. Accessed Jan. 25, 2008.
181. Yankelovich, N., Haan, B. J., Meyrowitz, N. K., Drucker, S. M. 1988. Intermedia: The Concept and the Construction of a Seamless Information Environment. *IEEE Computer* 21(1), 81-83, 90-96.

182. Yee, K. 2002. CritLink: Advanced Hyperlinks Enable Public Annotation on the Web. In *Proceedings of ACM 2002 Conference on Computer Supported Cooperative Work*, Nov. 16-20, New Orleans, Louisiana.

Appendix A: Sixml Element Types

This appendix describes the complete set of element types we have defined to represent mark associations in a Sixml document. (Chapters 4 and 7 described Sixml).

Figure A.1 shows the hierarchy of Sixml element types in the form of a static class diagram drawn using the Unified Modeling Language. Section 7.3.2 introduces the types shown, except MarkAssociation and MarkValueAssociation. MarkAssociation defines the basic structure of a mark association. MarkValueAssociation includes the information needed to derive XML content from the context of marks.

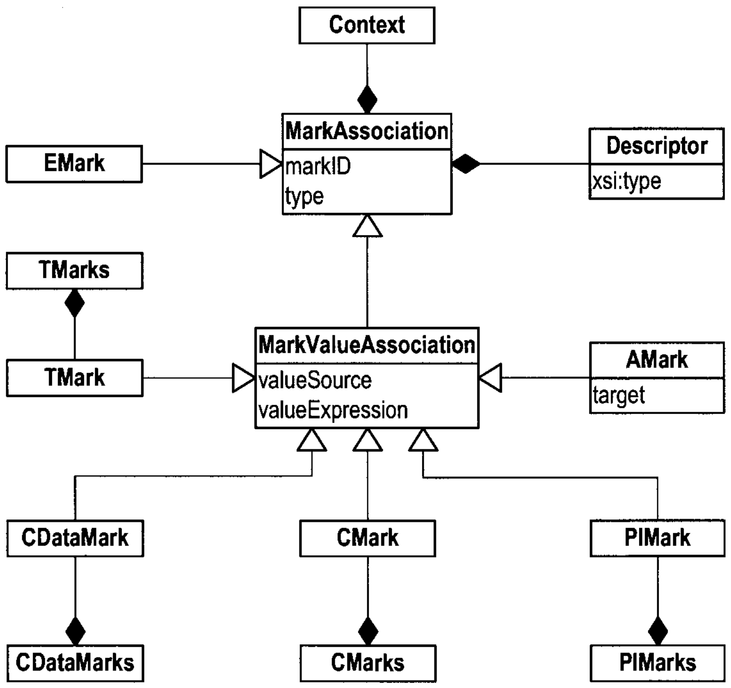


Figure A.1: Hierarchy of Sixml element types

The rest of this appendix lists the XML Schema instance document that defines the element types shown in Figure A.1. This document is also available online from http://schema.sixml.org.

```
<?xml version="1.0" encoding="utf-8"?>
<xs:schema targetNamespace="http://schema.sixml.org" xmlns:sixml="http://schema.sixml.org"
           xmlns:xs="http://www.w3.org/2001/XMLSchema">

 <!-- Base type for descriptors -->
 <xs:complexType name="Descriptor" mixed="true" block="" abstract="true" final="">
  <xs:complexContent mixed="true">
   <xs:extension base="xs:anyType">
   </xs:extension>
  </xs:complexContent>
 </xs:complexType>

 <!--Context information of arbitrary internal structure -->
 <xs:element name="Context">
  <xs:complexType mixed="true">
   <xs:sequence minOccurs="0">
    <xs:any minOccurs="0" maxOccurs="unbounded" namespace="##any" processContents="skip"/>
   </xs:sequence>
   <xs:anyAttribute namespace="##any" processContents="skip"/>
  </xs:complexType>
 </xs:element>

 <!-- Base type for all mark association types -->
 <xs:element name="Descriptor" type="sixml:Descriptor"/>
 <xs:attribute name="type" type="xs:QName"/>
 <xs:attribute name="markID" type="xs:string"/>
 <xs:complexType name="MarkAssociation" final="restriction" abstract="true" mixed="true" block="#all">
  <xs:complexContent mixed="true">
   <xs:restriction base="xs:anyType">
    <xs:sequence minOccurs="0" maxOccurs="1">
     <!-- A mark association element may use an instance of a type derived from sixml:Descriptor. -->
     <xs:element ref="sixml:Descriptor" minOccurs="0" maxOccurs="1"/>
     <xs:element ref="sixml:Context" minOccurs="0" maxOccurs="1">
     </xs:element>
    </xs:sequence>
    <!--
     Denote the type of the mark association. The value of this attribute must be the qualified name of a mark association type.
     This attribute is useful to define a mark association element of arbitrary name when Sixml data is processed
     with a DOM implementation that does not conform to Level 3 (that is, does not recognize type information).
    -->
    <xs:attribute ref="sixml:type" use="optional"/>
    <xs:attribute ref="sixml:markID" use="optional"/>
    </xs:restriction>
  </xs:complexContent>
</xs:complexType>

<!-- Base type for mark association types that support value derivation -->
<xs:attribute name="valueSource" type="xs:boolean"/>
<xs:attribute name="valueExpression" type="xs:string"/>
<xs:complexType name="MarkValueAssociation" final="restriction" abstract="true" mixed="true" block="#all">
  <xs:complexContent mixed="true">
    <xs:extension base="sixml:MarkAssociation">
      <!-- Denotes whether the associated mark contributes to the value of the target node -->
      <xs:attribute ref="sixml:valueSource" use="optional"/>
      <!-- Denotes what base value contributes to the value of the target node: must be empty or a path expression over context information -->
      <xs:attribute ref="sixml:valueExpression" use="optional"/>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>

<!-- Type to associate a mark with an element -->
<xs:complexType name="EMark" final="restriction" mixed="true" block="#all">
  <xs:complexContent mixed="true">
    <xs:extension base="sixml:MarkAssociation"/>
  </xs:complexContent>
</xs:complexType>

<!-- Type to list QNames used by type AMark -->
<xs:simpleType name="QNameList">
  <xs:list itemType="xs:QName" />
</xs:simpleType>

<!-- Type to associate a mark with attributes -->
<xs:attribute name="target">
  <xs:simpleType>
    <!-- An AMark instance associates a mark with one or more attributes of an element -->
    <xs:restriction base="sixml:QNameList">
      <xs:minLength value="1"/>
    </xs:restriction>
  </xs:simpleType>
</xs:attribute>
<xs:complexType name="AMark" final="restriction" block="#all" mixed="true">
  <xs:complexContent mixed="true">
    <xs:extension base="sixml:MarkValueAssociation">
```

```
      <xs:attribute ref="sixml:target" use="required"/>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>

<!-- Type to associate a mark with text -->
<xs:complexType name="TMark" final="restriction" block="#all" mixed="true">
  <xs:complexContent mixed="true">
    <xs:extension base="sixml:MarkValueAssociation"/>
  </xs:complexContent>
</xs:complexType>

<!-- Type to associate multiple marks with text -->
<xs:complexType name="TMarks" final="restriction" block="#all" mixed="true">
  <xs:complexContent mixed="true">
    <xs:extension base="xs:anyType">
      <xs:sequence minOccurs="1">
        <!-- Any element that satisfies the schema of sixml:TMark will do -->
        <xs:element ref="sixml:TMark"/>
      </xs:sequence>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>

<!-- Type to associate a mark with a CData section -->
<xs:complexType name="CDataMark" final="restriction" block="#all" mixed="true">
  <xs:complexContent mixed="true">
    <xs:extension base="sixml:MarkValueAssociation"/>
  </xs:complexContent>
</xs:complexType>

<!-- Type to associate multiple marks with a CData section -->
<xs:complexType name="CDataMarks" final="restriction" block="#all" mixed="true">
  <xs:complexContent mixed="true">
    <xs:extension base="xs:anyType">
      <xs:sequence minOccurs="1">
        <xs:element ref="sixml:CDataMark"/>
      </xs:sequence>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
```

```
<!-- Type to associate a mark with a comment -->
<xs:complexType name="CMark" final="restriction" block="#all" mixed="true">
  <xs:complexContent mixed="true">
    <xs:extension base="sixml:MarkValueAssociation"/>
  </xs:complexContent>
</xs:complexType>

<!-- Type to associate multiple marks with a comment -->
<xs:complexType name="CMarks" final="restriction" block="#all" mixed="true">
  <xs:complexContent mixed="true">
    <xs:extension base="xs:anyType">
      <xs:sequence minOccurs="1">
        <xs:element ref="sixml:CMark"/>
      </xs:sequence>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>

<!-- Type to associate a mark with a processing instruction -->
<xs:complexType name="PIMark" final="restriction" block="#all" mixed="true">
  <xs:complexContent mixed="true">
    <xs:extension base="sixml:MarkValueAssociation"/>
  </xs:complexContent>
</xs:complexType>

<!-- Type to associate multiple marks with a processing instruction -->
<xs:complexType name="PIMarks" final="restriction" block="#all" mixed="true">
  <xs:complexContent mixed="true">
    <xs:extension base="xs:anyType">
      <xs:sequence minOccurs="1">
        <xs:element ref="sixml:PIMark"/>
      </xs:sequence>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
```

```
  <!--
    Elements to help define a schema that uses default mark association names in the sixml namespace.
    The schema of an element may use the "ref" attribute to reference one of these elements.
  -->
  <xs:element name="EMark" type="sixml:EMark"/>
  <xs:element name="AMark" type="sixml:AMark"/>
  <xs:element name="TMark" type="sixml:TMark"/>
  <xs:element name="TMarks" type="sixml:TMarks"/>
  <xs:element name="CDataMark" type="sixml:CDataMark"/>
  <xs:element name="CDataMarks" type="sixml:CDataMarks"/>
  <xs:element name="CMark" type="sixml:CMark"/>
  <xs:element name="CMarks" type="sixml:CMarks"/>
  <xs:element name="PIMark" type="sixml:PIMark"/>
  <xs:element name="PIMarks" type="sixml:PIMarks"/>

</xs:schema>
```

Appendix B: Sixml DOM Interface Definition

This appendix defines the complete application-programming interface of Sixml DOM. This interface corresponds to Alternative 3 described in Section 7.5. The interface is defined using the Interface Definition Language. (See http://www.omg.org.)

This interface definition (as well as the interface definitions for Alternatives 1, 2, and 4 outlined in Section 7.5) is also available online from http://dom.sixml.org.

B.1. Level 1 Core

```
/*
 * This file extends the W3C DOM Level 1 Core specification to allow mark associations
 * (Alternative 3).
 * Modifications are tagged by comments starting with the string "Sixml"
 *
 * This file: http://dom.sixml.org/a3/dom-1.idl
 *
 * Modified by: Sudarshan Murthy. smurthy period cs period pdx period edu.
 * Modified on: July 11, 2007
 * For more information on Sixml DOM, visit http://dom.sixml.org.
 *
 * Modifications (c) Sudarshan Murthy. All rights reserved.
 * Permission to use modifications for non-commercial purpose. Use at your own risk.
 * No warranties expressed or implied.
 */

#ifndef _DOM_IDL_
#define _DOM_IDL_

#pragma prefix "w3c.org"
module dom
{

  valuetype DOMString sequence<unsigned short>;

  interface DocumentType;
  interface Document;
  interface NodeList;
  interface NamedNodeMap;
  interface Element;

  exception DOMException {
    unsigned short   code;
  };
  // ExceptionCode
```

```
const unsigned short      INDEX_SIZE_ERR                 = 1;
const unsigned short      DOMSTRING_SIZE_ERR             = 2;
const unsigned short      HIERARCHY_REQUEST_ERR          = 3;
const unsigned short      WRONG_DOCUMENT_ERR             = 4;
const unsigned short      INVALID_CHARACTER_ERR          = 5;
const unsigned short      NO_DATA_ALLOWED_ERR            = 6;
const unsigned short      NO_MODIFICATION_ALLOWED_ERR    = 7;
const unsigned short      NOT_FOUND_ERR                  = 8;
const unsigned short      NOT_SUPPORTED_ERR              = 9;
const unsigned short      INUSE_ATTRIBUTE_ERR            = 10;


interface DOMImplementation {
  boolean            hasFeature(in DOMString feature, in DOMString version);
};

interface Node {

  // NodeType
  const unsigned short      ELEMENT_NODE                   = 1;
  const unsigned short      ATTRIBUTE_NODE                 = 2;
  const unsigned short      TEXT_NODE                      = 3;
  const unsigned short      CDATA_SECTION_NODE             = 4;
  const unsigned short      ENTITY_REFERENCE_NODE          = 5;
  const unsigned short      ENTITY_NODE                    = 6;
  const unsigned short      PROCESSING_INSTRUCTION_NODE    = 7;
  const unsigned short      COMMENT_NODE                   = 8;
  const unsigned short      DOCUMENT_NODE                  = 9;
  const unsigned short      DOCUMENT_TYPE_NODE             = 10;
  const unsigned short      DOCUMENT_FRAGMENT_NODE         = 11;
  const unsigned short      NOTATION_NODE                  = 12;

  readonly attribute DOMString        nodeName;
           attribute DOMString        nodeValue;
                                      // raises(DOMException) on setting
                                      // raises(DOMException) on retrieval

  readonly attribute unsigned short   nodeType;
  readonly attribute Node             parentNode;
  readonly attribute NodeList         childNodes;
  readonly attribute Node             firstChild;
  readonly attribute Node             lastChild;
  readonly attribute Node             previousSibling;
  readonly attribute Node             nextSibling;
  readonly attribute NamedNodeMap     attributes;
  readonly attribute Document         ownerDocument;
  Node               insertBefore(in Node newChild, in Node refChild)
                                      raises(DOMException);
  Node               replaceChild(in Node newChild, in Node oldChild)
                                      raises(DOMException);
  Node               removeChild(in Node oldChild) raises(DOMException);
  Node               appendChild(in Node newChild) raises(DOMException);
  boolean            hasChildNodes();
  Node               cloneNode(in boolean deep);
};

interface NodeList {
  Node               item(in unsigned long index);
  readonly attribute unsigned long    length;
};

interface NamedNodeMap {
  Node               getNamedItem(in DOMString name);
  Node               setNamedItem(in Node arg) raises(DOMException);
  Node               removeNamedItem(in DOMString name) raises(DOMException);
  Node                item(in unsigned long index);
  readonly attribute unsigned long    length;
};

interface CharacterData : Node {
           attribute DOMString        data;
                                      // raises(DOMException) on setting
                                      // raises(DOMException) on retrieval

  readonly attribute unsigned long    length;
  DOMString           substringData(in unsigned long offset, in unsigned long count)
                                      raises(DOMException);
  void                appendData(in DOMString arg) raises(DOMException);
  void                insertData(in unsigned long offset, in DOMString arg)
                                      raises(DOMException);
  void                deleteData(in unsigned long offset, in unsigned long count)
                                      raises(DOMException);
  void                replaceData(in unsigned long offset, in unsigned long count,
                                 in DOMString arg) raises(DOMException);
};

interface Attr : Node {
  readonly attribute DOMString        name;
  readonly attribute boolean          specified;
           attribute DOMString        value;
                                      // raises(DOMException) on setting
};

interface Element : Node {
  readonly attribute DOMString        tagName;
  DOMString           getAttribute(in DOMString name);
  void                setAttribute(in DOMString name, in DOMString value)
                                      raises(DOMException);
  void                removeAttribute(in DOMString name) raises(DOMException);
  Attr                getAttributeNode(in DOMString name);
  Attr                setAttributeNode(in Attr newAttr) raises(DOMException);
  Attr                removeAttributeNode(in Attr oldAttr) raises(DOMException);
  NodeList            getElementsByTagName(in DOMString name);
  void                normalize();
};

interface Text : CharacterData {
  Text                splitText(in unsigned long offset) raises(DOMException);
};

interface Comment : CharacterData {  };

interface CDATASection : Text {  };

interface DocumentType : Node {
  readonly attribute DOMString        name;
  readonly attribute NamedNodeMap     entities;
  readonly attribute NamedNodeMap     notations;
};

interface Notation : Node {
  readonly attribute DOMString        publicId;
  readonly attribute DOMString        systemId;
};

interface Entity : Node {
  readonly attribute DOMString        publicId;
  readonly attribute DOMString        systemId;
  readonly attribute DOMString        notationName;
};
```

```
interface EntityReference : Node {  };

interface ProcessingInstruction : Node {
  readonly attribute DOMString        target;
           attribute DOMString        data;
                                      // raises(DOMException) on setting

};

interface DocumentFragment : Node {  };

interface Document : Node {
  readonly attribute DocumentType     doctype;
  readonly attribute DOMImplementation  implementation;
  readonly attribute Element          documentElement;
  Element             createElement(in DOMString tagName) raises(DOMException);
  DocumentFragment    createDocumentFragment();
  Text                createTextNode(in DOMString data);
  Comment             createComment(in DOMString data);
  CDATASection        createCDATASection(in DOMString data)
                                      raises(DOMException);
  ProcessingInstruction createProcessingInstruction(in DOMString target,
                                                    in DOMString data)
                                      raises(DOMException);
  Attr                createAttribute(in DOMString name) raises(DOMException);
  EntityReference     createEntityReference(in DOMString name) raises(DOMException);
  NodeList            getElementsByTagName(in DOMString tagname);
};

//Sixml Sudarshan Murthy July 11, 2007
//Introduced to support mark associations

// ExceptionCode
const unsigned short      INSUFFICIENT_INFO_ERR          = 18;

interface MarkAssociation;

interface SixmlNode : Node {
  readonly attribute NodeList        markAssociations;
  boolean             hasMarkAssociations();

  MarkAssociation     insertMarkAssociationBefore(in Node newMarkAssociation,
                                                  in Node refMarkAssociation)
                                                     raises(DOMException);
  MarkAssociation     replaceMarkAssociation(in Node newMarkAssociation,
                                             in Node oldMarkAssociation)
                                                raises(DOMException);
  MarkAssociation     removeMarkAssociation(in Node oldMarkAssociation)
                                           raises(DOMException);
  MarkAssociation     appendMarkAssociation(in Node newMarkAssociation)
                                           raises(DOMException);

  NodeList            getMarkAssociationsByName(in DOMString name);
};

interface SixmlValueNode : SixmlNode {
  boolean             isValueFromMarks();
};

interface SixmlElement : SixmlNode, Element {  };

interface SixmlAttr : SixmlValueNode, Attr {  };

interface SixmlText : SixmlValueNode, Text {  };
```

```
  interface SixmlCDATASection : SixmlValueNode, CDATASection {  };

  interface SixmlComment : SixmlValueNode, Comment {  };

  interface SixmlProcessingInstruction : SixmlValueNode, ProcessingInstruction {  };

  interface Mark {
    readonly attribute  DOMString    markId;
    readonly attribute  DOMString    descriptor;
    DOMString          getValue(in DOMString valueExpression) raises(DOMException);
  };

  interface MarkFactory {
    readonly attribute  DOMString    markType;
    Mark               createMark(in DOMString markId, in DOMString descriptor)
                                  raises(DOMException);
  };

  interface MarkRepository {
    readonly attribute  DOMString    name;
    Mark               getMark(in DOMString markId, in DOMString descriptor)
                               raises(DOMException);
  };

  interface MarkAssociation : Element {
    readonly attribute  DOMString    markID;
    readonly attribute  DOMString    descriptor;
             attribute  boolean      isValueSource;
             attribute  DOMString    valueExpression;

    Element            getDescriptorElement() raises(DOMException);
    Element            getContextElement() raises(DOMException);
  };


  interface SixmlDocument : Document {
    Mark               getMark(in DOMString markId, in DOMString descriptor)
                               raises(DOMException);

    MarkAssociation    createMarkAssociation(in DOMString name, in DOMString markId,
                                      in DOMString descriptor)
                                         raises(DOMException);
  };

};

#endif // _DOM_IDL_
```

B.2. Level 2 Core

For this module, we show only the additions we have made to the existing interface definition.

```
interface SixmlNode : Node {
  readonly attribute NodeList        markAssociations;
  boolean              hasMarkAssociations();

  MarkAssociation    insertMarkAssociationBefore(in Node newMarkAssociation,
                                                 in Node refMarkAssociation)
                                                   raises(dom::DOMException);
  MarkAssociation    replaceMarkAssociation(in Node newMarkAssociation,
                                            in Node oldMarkAssociation)
                                              raises(dom::DOMException);
  MarkAssociation    removeMarkAssociation(in Node oldMarkAssociation)
                                             raises(dom::DOMException);
  MarkAssociation    appendMarkAssociation(in Node newMarkAssociation)
                                             raises(dom::DOMException);

  NodeList           getMarkAssociationsByName(in DOMString name);

  //Introduced in Level 2
  NodeList           getMarkAssociationByNameNS(in DOMString namespaceURI,
                                                in DOMString localName)
                                                raises(dom::DOMException);
};
```

B.3. Level 3 Load and Save

For this module, we show only the additions we have made to the existing interface definition.

```
//Introduced Sixml parser and serializer


//scope of parsing and serializing
const unsigned short     SI_SCOPE                     = 1;
const unsigned short     MARK_ASSOCIATION_SCOPE       = 2;
const unsigned short     MARK_DESCRIPTOR_SCOPE        = 3;

interface SixmlParser: ls::LSParser { attribute unsigned short scope; };

interface SixmlSerializer: ls::LSSerializer { attribute unsigned short scope; };
```

Appendix C: Queries Used in the Evaluation of the Bi-level Navigator

This appendix lists the queries used in experimental evaluation of the bi-level navigator. Section 9.4.3 described the queries and the experimental results. (See also Table 9.6.)

For ease of presentation, we first present the XPath expressions used, followed by the XSLT style sheets.

C.1. XPath Queries

Q1: Retrieve all SI (all documents).

Scope SI, both normalized schema and nested schema.

. (the character period by itself)

Q5: Retrieve the text of all comments (SISRS documents)

Scope SI, both normalized schema and nested schema (three variations).

```
//text()
```

```
//Comment/text()
```

```
/Reviews/Paper/Comment/text()
```

Q6: Retrieve paper titles (SISRS documents)

Scope SI, normalized schema:

```
/Reviews/Paper/@title
```

Scope Context, normalized schema.

```
/Reviews/Paper/@title/*/sixml:Context/Content/Text
```

Q7: List the base documents for security events (SSIB documents)

Scope Descriptor, normalized schema.

```
/SSIB/Computer/Events/Event[@kind='Security']/@*//sixml:Descriptor/Doc
```

C.2. XSLT Style Sheets

Q1: Retrieve all SI (all documents). Scope Association or Descriptor, both normalized schema and nested schema.

```
<!--
 Extract SI from a Sixml document represented using Sixml DOM.
 - Only EMark elements are explicitly removed because Sixml DOM handles other mark associations

 Last modified on: December 30, 2007.

 For more information on Sixml, visit http://schema.sixml.org.
 Contact: Sudarshan Murthy <firstLetterOfFirstNameThenAllofLastName at sixml dot org>

 (c) 2007 Sudarshan Murthy. All rights reserved.
 Permission to use for non-commercial purpose. Use at your own risk. No warranties expressed or
 implied.
-->
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                xmlns:sixml="http://schema.sixml.org">

 <!-- Include template to determine mark association type from attribute sixml:type -->
 <xsl:include href="GetMarkAssociationTypeFromTypeAttribute.xslt"/>

 <!-- Filter EMark elements with default name -->
 <xsl:template  match="sixml:EMark" priority="2"/>

 <!-- Filter EMark elements with custom name -->
 <xsl:template match="*[@sixml:type]" priority="1">
   <xsl:variable name="typeName">
     <xsl:call-template name="GetMarkAssociationTypeFromTypeAttribute"/>
   </xsl:variable>
   <xsl:if test="not($typeName = 'EMark')">
     <xsl:call-template name="CopySIElement"/>
   </xsl:if>
 </xsl:template>

 <!-- Copy SI element: copy attributes and process children -->
 <xsl:template name="CopySIElement" match="*[not(@sixml:type)]" priority="1">
   <xsl:copy>
     <xsl:copy-of select="@*"/>
     <xsl:apply-templates select="node()"/>
   </xsl:copy>
 </xsl:template>

</xsl:stylesheet>
```

Q2: Retrieve all mark associations (all documents). Scope Association, both normalized schema and nested schema.

```
<!--
 Extract mark associations (exclude mark descriptors) from a Sixml document represented using
 Sixml DOM
 - EMark elements are distinguished from other children of an element; Sixml DOM distinguishes other
   kinds of mark associations
 - Performs deep copy of a mark association element and its attributes
 - Query scope must be set to 'Associations'

 Last modified on: December 30, 2007.

 For more information on Sixml, visit http://schema.sixml.org.
 Contact: Sudarshan Murthy <firstLetterOfFirstNameThenAllofLastName at sixml dot org>

 (c) 2007 Sudarshan Murthy. All rights reserved.
 Permission to use for non-commercial purpose. Use at your own risk. No warranties expressed or
 implied.
-->
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                xmlns:sixml="http://schema.sixml.org">
 <!-- Include template to determine mark association type from attribute sixml:type -->
 <xsl:include href="GetMarkAssociationTypeFromTypeAttribute.xslt"/>
 <xsl:template  match="/">
  <MarkAssociations>
   <xsl:apply-templates select="*"/>
  </MarkAssociations>
 </xsl:template>
 <!-- Copy mark associations for attributes, text, comments, and PI: They are revealed as children -->
 <xsl:template match="@*|text()|comment()|processing-instruction()">
  <xsl:copy-of select="*"/>
 </xsl:template>
 <!-- Copy EMark elements that use default name -->
 <xsl:template match="sixml:EMark" priority="2">
  <xsl:copy-of select="."/>
 </xsl:template>
 <!-- Copy EMark elements that use custom names -->
 <xsl:template match="*[@sixml:type]" priority="1">
  <xsl:variable name="typeName">
   <xsl:call-template name="GetMarkAssociationTypeFromTypeAttribute"/>
  </xsl:variable>
  <xsl:choose>
   <xsl:when test="$typeName = 'EMark'">
    <xsl:copy-of select="*"/>
   </xsl:when>
   <xsl:otherwise>
    <xsl:call-template name="ProcessSIElement"/>
   </xsl:otherwise>
  </xsl:choose>
 </xsl:template>
 <!-- Process mark associations for children and attributes of SI elements -->
 <xsl:template name="ProcessSIElement" match="*[not(@sixml:type)]" priority="1">
  <xsl:apply-templates select="node()"/>
  <xsl:apply-templates select="@*"/>
 </xsl:template>
</xsl:stylesheet>
```

Q2: Retrieve all mark associations (all documents). Scope Descriptor, both normalized schema and nested schema.

```
<!--
  Extract mark associations (exclude mark descriptors) from a Sixml document represented using
  Sixml DOM
  - EMark elements are distinguished from other children of an element; Sixml DOM distinguishes other
    kinds of mark associations
  - Performs shallow copy of a mark association element and its attributes
  - Query scope can be set to "Associations" or greater

  Last modified on: December 30, 2007.

  For more information on Sixml, visit http://schema.sixml.org.
  Contact: Sudarshan Murthy <firstLetterOfFirstNameThenAllofLastName at sixml dot org>

  (c) 2007 Sudarshan Murthy. All rights reserved.
  Permission to use for non-commercial purpose. Use at your own risk. No warranties expressed or
  implied.
-->
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                xmlns:sixml="http://schema.sixml.org">
 <!-- Include template to determine mark association type from attribute sixml:type -->
 <xsl:include href="GetMarkAssociationTypeFromTypeAttribute.xslt"/>
 <xsl:template  match="/">
   <MarkAssociations>
     <xsl:apply-templates select="*"/>
   </MarkAssociations>
 </xsl:template>
 <!-- Copy mark associations for attributes, text, comments, and PI: They are revealed as children -->
 <xsl:template match="@*|text()|comment()|processing-instruction()">
   <xsl:apply-templates select="*" mode="Copy"/>
 </xsl:template>
 <!-- Copy EMark elements that use default name -->
 <xsl:template match="sixml:EMark" priority="2">
   <xsl:call-template name="CopyUniMark"/>
 </xsl:template>
<!-- Copy EMark elements that use custom names -->
 <xsl:template match="*[@sixml:type]" priority="1">
   <xsl:variable name="typeName">
     <xsl:call-template name="GetMarkAssociationTypeFromTypeAttribute"/>
   </xsl:variable>
   <xsl:choose>
     <xsl:when test="$typeName = 'EMark'">
       <xsl:call-template name="CopyUniMark"/>
     </xsl:when>
     <xsl:otherwise>
       <xsl:call-template name="ProcessSIElement"/>
     </xsl:otherwise>
   </xsl:choose>
 </xsl:template>
<!-- Process mark associations for children and attributes of SI elements -->
 <xsl:template name="ProcessSIElement" match="*[not(@sixml:type)]" priority="1">
   <xsl:apply-templates select="node()"/>
   <xsl:apply-templates select="@*"/>
 </xsl:template>
 <!-- Shallow-copy a mark association: Leave out mark descriptor and context, depending on query
       scope -->
 <xsl:template name="CopyUniMark" match="*" mode="Copy">
   <xsl:copy><xsl:copy-of select="@*"/></xsl:copy>
 </xsl:template>
</xsl:stylesheet>
```

Q1, Q2 Supplement: Template to get the type of mark-association element based on the attribute sixml:type. This template is used in Queries Q1 and Q2.

```
<!--
  Get the local name of the type of the mark association an element represents from the attribute
  @sixml:type
  - Utility template to include in other styles heets

  Last modified on: December 30, 2007.

  For more information on Sixml, visit http://schema.sixml.org.
  Contact: Sudarshan Murthy <firstLetterOfFirstNameThenAllofLastName at sixml dot org>

  (c) 2007 Sudarshan Murthy. All rights reserved.
  Permission to use for non-commercial purpose. Use at your own risk. No warranties expressed or
  implied.
-->
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                xmlns:sixml="http://schema.sixml.org">

 <!-- Get the local name of the type of the mark association that an element represents from
       @sixml:type
 -->
 <xsl:template name="GetMarkAssociationTypeFromTypeAttribute">
   <!-- extract the prefix and local name from the QName in @sixml:type -->
   <xsl:variable name="prefix" select="substring-before(@sixml:type, ':')"/>
   <xsl:if test="string(namespace::*[name()=$prefix])='http://schema.sixml.org'">
     <xsl:choose>
       <xsl:when test="contains(@sixml:type, ':')">
         <xsl:value-of select="substring-after(@sixml:type, ':')"/>
       </xsl:when>
       <xsl:otherwise>
         <xsl:value-of select="string(@sixml:type)"/>
       </xsl:otherwise>
     </xsl:choose>
   </xsl:if>
 </xsl:template>

</xsl:stylesheet>
```

Q3: Retrieve unique mark descriptors (all documents). Scope Descriptor, normalized schema.

```
<!--
 Extract unique mark descriptors from a Sixml document represented using Sixml DOM
 - Assumes input has mark associations with only mark ID and no mark descriptors
 - Query scope must be set to "Descriptors"
 - Reuses the templates to extract mark associations when the scope is set to "Associations"
   - Works because the included templates  "deep copy" mark associations, which includes descriptors
     as needed in this application
 - Reuses the templates to compute the "signature" of a mark descriptor
   - Signatures are used to test if two descriptors are equal.

 Last modified on: December 30, 2007.

 For more information on Sixml, visit http://schema.sixml.org.
 Contact: Sudarshan Murthy <firstLetterOfFirstNameThenAllofLastName at sixml dot org>

 (c) 2007 Sudarshan Murthy. All rights reserved.
 Permission to use for non-commercial purpose. Use at your own risk. No warranties expressed or
implied.
-->

<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                xmlns:sixml="http://schema.sixml.org"
                xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
                xmlns:msxsl="urn:schemas-microsoft-com:xslt">

 <!-- Include templates to extract mark associations -->
 <xsl:include href="q2_ExtractAssociations_Sixml_ScopeA.xslt"/>

 <!-- Include templates to compute the "signature" of a descriptor-->
 <xsl:include href="DescriptorSignature.xslt"/>

 <xsl:template  match="/">
  <MarkDescriptors>
   <xsl:variable name="associations">
    <xsl:apply-templates select="*"/>
   </xsl:variable>
   <xsl:call-template name="CopyUniqueMarkDescriptors">
    <xsl:with-param name="associations" select="msxsl:node-set($associations)/*"/>
    <xsl:with-param name="signatures">
     <xsl:apply-templates select="msxsl:node-set($associations)/*/sixml:Descriptor"
                          mode="Signature"/>
    </xsl:with-param>
   </xsl:call-template>
  </MarkDescriptors>
 </xsl:template>

 <!-- Recursively process the list of mark associations and output unique descriptors -->
 <xsl:template  name="CopyUniqueMarkDescriptors">
  <xsl:param name="associations"/>
  <xsl:param name="signatures"/>
  <xsl:param name="index" select="1"/>

  <xsl:if test="count($associations) >= $index">
```

```
    <xsl:if test="not($associations[$index]/@sixml:markID = $associations[$index >
                   position()]/@sixml:markID)">
      <xsl:if test="not(msxsl:node-set($signatures)/*[$index] = msxsl:node-set($signatures)/*[$index
                     > position()])">
        <!-- a descriptor for a mark association with this mark ID and a descriptor with this signature
             has not been copied thus far
        -->
        <xsl:copy-of select="$associations[$index]/sixml:Descriptor"/>
      </xsl:if>
    </xsl:if>
    <xsl:call-template name="CopyUniqueMarkDescriptors">
      <xsl:with-param name="associations" select="$associations"/>
      <xsl:with-param name="signatures" select="$signatures"/>
      <xsl:with-param name="index" select="$index+1"/>
    </xsl:call-template>
  </xsl:if>
</xsl:template>

</xsl:stylesheet>
```

Q3: Retrieve unique mark descriptors (all documents). Scope Descriptor, nested schema.

```
<!--
 Extract unique mark descriptors from a Sixml document represented using Sixml DOM
 - Assumes input has mark descriptors but no mark ID
 - Query scope must be set to "Descriptors"
 - Reuses the templates to extract mark associations when the scope is set to "Associations"
   - Works because the included templates "deep copy" mark associations, including descriptors
     as needed in this application
 - Reuses the templates to compute the "signature" of a mark descriptor
   - Signatures are used to test if two descriptors are equal.

 Last modified on: December 30, 2007.

 For more information on Sixml, visit http://schema.sixml.org.
 Contact: Sudarshan Murthy <firstLetterOfFirstNameThenAllofLastName at sixml dot org>

 (c) 2007 Sudarshan Murthy. All rights reserved.
 Permission to use for non-commercial purpose. Use at your own risk. No warranties expressed or
 implied.
-->

<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                xmlns:sixml="http://schema.sixml.org"
                xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
                xmlns:msxsl="urn:schemas-microsoft-com:xslt">

 <!-- Include templates to extract mark associations -->
 <xsl:include href="q2_ExtractAssociations_Sixml_ScopeA.xslt"/>

 <!-- Include templates to compute the "signature" of a descriptor-->
 <xsl:include href="DescriptorSignature.xslt"/>

 <xsl:template  match="/">
   <MarkDescriptors>
     <xsl:variable name="associations">
       <xsl:apply-templates select="*"/>
     </xsl:variable>
     <xsl:call-template name="CopyUniqueMarkDescriptors">
       <xsl:with-param name="associations" select="msxsl:node-set($associations)/*"/>
       <xsl:with-param name="signatures">
         <xsl:apply-templates select="msxsl:node-set($associations)/*/sixml:Descriptor"
                              mode="Signature"/>
       </xsl:with-param>
     </xsl:call-template>
   </MarkDescriptors>
 </xsl:template>

 <!-- Recursively process the list of mark associations and output unique descriptors -->
 <xsl:template  name="CopyUniqueMarkDescriptors">
   <xsl:param name="associations"/>
   <xsl:param name="signatures"/>
   <xsl:param name="index" select="1"/>

   <xsl:if test="count($associations) >= $index">
     <xsl:if test="not(msxsl:node-set($signatures)/*[$index] = msxsl:node-set($signatures)/*[$index
                  > position()])">
```

```
      <!-- a descriptor with this signature has not been copied thus far -->
      <xsl:copy-of select="$associations[$index]/sixml:Descriptor"/>
    </xsl:if>
    <xsl:call-template name="CopyUniqueMarkDescriptors">
      <xsl:with-param name="associations" select="$associations"/>
      <xsl:with-param name="signatures" select="$signatures"/>
      <xsl:with-param name="index" select="$index+1"/>
    </xsl:call-template>
  </xsl:if>
 </xsl:template>

</xsl:stylesheet>
```

Q3 Supplement: Template to compute the signature of a mark descriptor. This template is used by Query Q3.

```
<!--
  Compute "signature" of a mark descriptor
  - The signature of a descriptor is essentially its "outer xml", without the xml punctuations
  - Utility templates to include in other stylesheets

  Last modified on: December 30, 2007.

  For more information on Sixml, visit http://schema.sixml.org.
  Contact: Sudarshan Murthy <firstLetterOfFirstNameThenAllofLastName at sixml dot org>

  (c) 2007 Sudarshan Murthy. All rights reserved.
  Permission to use for non-commercial purpose. Use at your own risk. No warranties expressed or
  implied. -->
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                xmlns:sixml="http://schema.sixml.org">
 <!-- Place the signature as the text content of an element -->
 <xsl:template match="sixml:Descriptor" mode="Signature">
   <Signature>
     <xsl:call-template name="Signature"/>
   </Signature>
 </xsl:template>
 <!-- An element's signature is composed of its local name qualified by its namespace URI, the number
      of attributes, and the signatures of attributes and children -->
 <xsl:template name="Signature" match="*" mode="Signature">
   <xsl:value-of select="concat(namespace-uri(), local-name(), count(@*))"/>
   <xsl:apply-templates select="@*" mode="Signature"/>
   <xsl:apply-templates select="node()" mode="Signature"/>
 </xsl:template>
 <!-- An attribute's signature is its local name qualified by its namespace URI, and the attribute's value
-->
 <xsl:template match="@*" mode="Signature">
  <xsl:value-of select="concat(namespace-uri(), local-name(), .)"/>
 </xsl:template>
 <!-- A text node's signature is simply its content -->
 <xsl:template match="text()" mode="Signature">
  <xsl:value-of select="."/>
 </xsl:template>
 <!-- No need to compute signature for comments and PIs -->
</xsl:stylesheet>
```

Q4: List the base documents referenced by the superimposed information (all documents). Scope Descriptor, both normalized schema and nested schema

```
<!--
 Extract unique mark descriptors from a Sixml document represented using Sixml DOM
 - Query scope must be set to "Descriptors"
 - Reuses the templates to extract mark associations when the scope is set to "Associations"
   - Works because the included templates  "deep copy" mark associations, including descriptors
     as needed in this application

 Last modified on: December 30, 2007.

 For more information on Sixml, visit http://schema.sixml.org.
 Contact: Sudarshan Murthy <firstLetterOfFirstNameThenAllofLastName at sixml dot org>

 (c) 2007 Sudarshan Murthy. All rights reserved.
 Permission to use for non-commercial purpose. Use at your own risk. No warranties expressed or
 implied.
-->
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                xmlns:sixml="http://schema.sixml.org"
                xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
                xmlns:msxsl="urn:schemas-microsoft-com:xslt">

 <!-- Include templates to extract mark associations -->
 <xsl:include href="q2_ExtractAssociations_Sixml_ScopeA.xslt"/>

 <xsl:template  match="/">
  <MarkDescriptors>
   <xsl:variable name="associations">
    <xsl:apply-templates select="*"/>
   </xsl:variable>
   <xsl:call-template name="CopyUniqueBaseDocs">
    <xsl:with-param name="docs" select="msxsl:node-set($associations)/*/sixml:Descriptor/Doc"/>
   </xsl:call-template>
  </MarkDescriptors>
 </xsl:template>

 <!-- Recursively process the list of documents and output unique documents -->
 <xsl:template  name="CopyUniqueBaseDocs">
  <xsl:param name="docs"/>
  <xsl:param name="index" select="1"/>

  <xsl:if test="count($docs) >= $index">
   <xsl:if test="not($docs[$index] = $docs[$index > position()])">
     <xsl:copy-of select="$docs[$index]"/>
   </xsl:if>
   <xsl:call-template name="CopyUniqueBaseDocs">
    <xsl:with-param name="docs" select="$docs"/>
    <xsl:with-param name="index" select="$index+1"/>
   </xsl:call-template>
  </xsl:if>
 </xsl:template>

</xsl:stylesheet>
```

Q8: Create a timeline of “application hang” events (SSIB documents). Scope SI, normalized schema.

```
<!--
  Construct a timeline of applications hanging for Computer '3' (from a Sixml document represented
  using Sixml DOM).
  - Uses micro queries
  - Query scope of 'SI' is sufficient

  Last modified on: December 30, 2007.

  For more information on Sixml, visit http://schema.sixml.org.
  Contact: Sudarshan Murthy <firstLetterOfFirstNameThenAllofLastName at sixml dot org>

  (c) 2007 Sudarshan Murthy. All rights reserved.
  Permission to use for non-commercial purpose. Use at your own risk. No warranties expressed or
  implied.
-->

<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                xmlns:sixml="http://schema.sixml.org">

 <xsl:template match="/">
  <AppHangs>
   <!-- Use a focused path expression so the query is efficient -->
   <xsl:apply-templates select="SSIB/Computer[@name='C3']/Errors/Error[@source=
                                'Application Hang']"/>
  </AppHangs>
 </xsl:template>

 <!-- Describe one application hang event -->
 <xsl:template  match="Error[@source='Application Hang']">
  <AppHang dateTime="{@dateTime}">
   <!-- The event description is of the form 'Hanging application xyz.exe,' -->
   <xsl:value-of select="substring-before(substring-after(@description, 'Hanging application '), ',')"/>
  </AppHang>
 </xsl:template >

</xsl:stylesheet>
```

Q8: Create a timeline of "application hang" events (SSIB documents). Scope Context, normalized schema.

```
<!--
  Construct a timeline of applications hanging for Computer '3' (from a Sixml document represented
  using Sixml DOM).
  - Does not use micro queries
  - Query scope must be set to 'Context'

  Last modified on: January 10, 2008.

  For more information on Sixml, visit http://schema.sixml.org.
  Contact: Sudarshan Murthy <firstLetterOfFirstNameThenAllofLastName at sixml dot org>

  (c) 2007 Sudarshan Murthy. All rights reserved.
  Permission to use for non-commercial purpose. Use at your own risk. No warranties expressed or
  implied.
-->

<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                xmlns:sixml="http://schema.sixml.org">

 <xsl:template match="/">
  <AppHangs>
   <!-- Use a focused path expression so the query is efficient  -->
   <xsl:apply-templates select="SSIB/Computer[@name='C3']/Errors/Error[@source=
                                'Application Hang']"/>
  </AppHangs>
 </xsl:template>

 <!-- Describe one application hang event  -->
 <xsl:template  match="Error[@source='Application Hang']">
  <AppHang dateTime="{@dateTime/*/Context/Content/Text/text()}">
   <!-- The event description is of the form 'Hanging application xyz.exe,'  -->
   <xsl:value-of select="substring-before(
                          substring-after(@description/*/Context/Content/Text/text(),
                                          'Hanging application '), ',')"/>
  </AppHang>
 </xsl:template >

</xsl:stylesheet>
```