Modelling complex documents

Weigand, H.

Publication date:
1991

Document Version
Publisher's PDF, also known as Version of record

[redacted]

ITK

RESEARCH
REPORT

INSTITUTE FOR LANGUAGE TECHNOLOGY AND ARTIFICIAL INTELLIGENCE

INSTITUUT VOOR TAAL- EN KENNISTECHNOLOGIE

ITK Research Report
May 21, 1991

Modelling Complex Documents

Hans Weigand

No. 25

SPRITE is an ESPRIT II project started in 1989 and developing a system for technical documentation. The partners are:

Océ Nederland (NL)
AEG Elektrokom (FRG)
TITN/Alcatel (F)
Trinity College Dublin (IR)
KUB/EIT (NL)

This report contains two articles written by members of the EIT group that is responsible for the design of the multimedia database:

Hederman, L., Weigand, H.: Versioned objects in a technical documentation system. In: Proc. ESPRIT Conference '90. North-Holland, Amsterdam, 1990 (10p).

Weigand, H.: An object-oriented approach in a multimedia database project. In: Kent, W., Meersman, R. (eds.), Object-oriented databases (DS-4), North-Holland, Amsterdam, 1991 (20p).

ISSN 0924-7807

ESPRIT Project 2001

S torage
P rocessing and
R etrieval of
I nformation in a
T echnical
E nvironment

Modelling complex documents

Project No. 2001

VERSIONED DOCUMENTS IN A TECHNICAL DOCUMENT MANAGEMENT SYSTEM

LUCY HEDERMAN
Dept. of Computer Science
Trinity College
Dublin 2
Ireland

HANS WEIGAND
EIT/KUB
P.O.Box 90153
5000 LE Tilburg
The Netherlands, Hans Weigand
fax: +31 13 663069
tel: +31 13 662688

ABSTRACT. A Document Management System (SPRITE) especially suited for technical documentation is described. The system includes a powerful WYSIWYG editor and scanning, archiving and printing facilities. A distinctive feature of SPRITE is its version model. SPRITE not only allows the user to maintain historical versions of documents, but also variants, or configurations, that represent slightly different versions of a certain document. The SPRITE version model is described in detail and compared with other versioning mechanisms.

1. Introduction

The commercial market of today offers a number of systems for technical documentation. They range from simple systems working with single documents up to sophisticated systems for an integral document management system, from batch-oriented systems up to state-of-the-art interactive WYSIWYG workstations (Walter, 1988). Examples are Documenter (Xerox) , The Publisher (ArborText) and KEEPS (Kodak).

Technical documentation differs from normal documentation as created by average text processing systems in the following areas:

- Technical documents are often very large in size. Documents exceeding 1000 pages are no exception.

- Technical documents are created by a group of authors often working concurrently.

- Technical documents have a long lifetime, following the progressive development of the product described. The development has to be supported by multiple versions of one document.

- Technical documents have to incorporate information from other sources (other documents, but also paper and remote CAD/CAM or database systems).

- Since technical documents are generally complex, there is a need for management support for such documents.

- Since technical products are often designed in series, as configurations differing on details only, the various documents describing the products also exist as close variants of each other.

To cope with these requirements, we are developing the SPRITE Document Management System as an integrated system for the production and maintenance of technical documents. We call such a system a Technical Document Management System (TDMS).

The rest of this paper is organized as follows. In section 2, we give a short overview of the functionality of the SPRITE system. In section 3, we spell out one feature, that is, how SPRITE supports document versions. The versioning mechanism is compared with other sytems, such as MINOS (a multimedia database), EXODUS, ORION and ONTOS (object-oriented database systems).

2. Overview of the SPRITE system

The document management system consists essentially of six components:

- a screen-oriented WYSIWYG document processor
- a browser and retrieval component
- a high-quality printing component
- a scanning & recognition component
- an information acquisition component
- a multimedia database (MMD)

Both the database and the document processor allow the use of text, graphic and raster data. In composing a document, an author can access other documents (by means of an *import* mechanism), or retrieve information from other systems (for example, CAD systems) by means of the information acquisition component, or extract information from paper by means of the scanning & recognition component.

The system offers several functions for project management. Work on documents can be delegated to several authors with well-defined permissions. Project information can be

attached to a document reflecting its lifecycle ("draft", "ready for review" etc).

Documents may exist in the system in different versions. By means of the browser component the user can easily view all versions of a document. The MMD supports efficient data sharing between different versions.

The system can also maintain links and interdependencies between documents. Such links can be used either to locate related documents with the browser (hypertext-like function), or to trigger an action in case one of a group of related documents has been updated and the other documents have to be updated as well.

Since technical documentation generally uses a large amount of data and has a long lifetime, the MMD is supplied with mass storage capabilities, which will be accomplished by integration of an *optical disk* within the MMD.

The SPRITE system also supports *multi-authoring*. It is possible for several authors to edit a document simultaneously. Provisions are taken so that concurrent access to a document does not cause inconsistencies, that authors have easy access to assigned document parts, and that a manager may easily supervise the progress of the work.

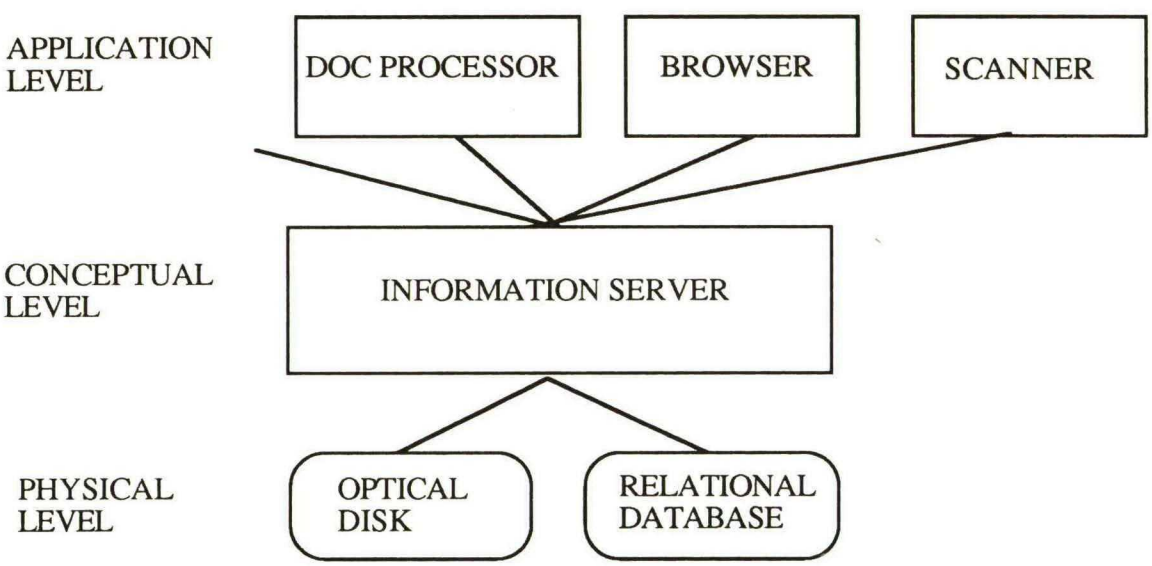

Figure 1: SPRITE system architecture

The architecture of the Document Management System is split up over three levels to enhance data-independence and extensibility (see figure 1). The bottom layer (physical layer) is responsible for the storage of content and structure of the documents. Structural information is stored in a relational database, and content on magnetic and optical disk. The middle layer (conceptual layer) comprises an object-oriented multimedia database (MMD). The MMD reflects the conceptual model of the document and the document space and supports conceptually meaningful operations such as CREATE document, INSERT document IN folder, SELECT (search condition) etc. On top of the MMD, the various

applications are located, such as the document processor, the browser, and the scanning & recognition.

The MMD of the SPRITE system concentrates on all media that can be printed on paper, such as text, drawings, photographs, tables (structured information) and business graphics. Sound and video are not included. The document model used by SPRITE has been based on ODA (ISO 8631 international standard for Office Document Architecture; for an introduction, see e.g. Krönert, 1987) but for practical reasons this standard was not followed completely.

3. Version model

As stated in the introduction, a technical document management system (TDMS) should support the use of versions. Two types of versions must be distinguished:

historical versions

configurations

Historical versions correspond with either the derivational or logical history of a document. Technical documents are developed on a project basis, in consecutive steps, over a long period of time. The TDMS should support the derivation of new versions of documents.

Different configurations of a product require (slightly) different versions of the documentation. For example, a car may exist in a manual transmission or automatic configuration. The documentation system should support corresponding configurations of the car's manual. It should also support versions that differ in style or language.

The MINOS system (Christodoulakis et al, 1986) and the EXODUS system (Carey et al, 1986) also support the derivation of new versions of a document, but no distinction is made between historical versions and configurations. However, the combination of these two dimensions is not trivial, and can easily lead to chaos if no special organizational measures are taken. Distinguishing the two has the advantage of a clear conceptual picture. In several situations, it also allows a greater level of data sharing, that is, higher efficiency in storage use. A distinction between historical versions and alternatives is made in several CAD/CAM systems and in the object-oriented database system ONTOS (Andrews, 1989). For an overview of some open questions about versions, see (Kent, 1989).

3.1. HISTORICAL VERSIONS

Technical documents exist over a long period of time and are developed in several steps. SPRITE allows the user to keep historical versions of a certain document, alternatively called *checkpoints*. This mechanism has several functions:

* recovery from mistakes. In the course of development, the author may find out that he is on the wrong track, and wants to start again from some previous version. In

that case, he can use the browser to locate that old version and start editing again from there.

* data sharing. When the author wants to rewrite some existing document, he need not copy its entire content. Deriving a new version from it is sufficient and guarantees efficient content sharing.
* project management. Technical documents typically have more than one edition. The version mechanism represents the logical relationship between the subsequent editions.

The historical version mechanism in SPRITE is implemented by the following operations:

NEW_CHECKPOINT(oid): oid
FREEZE(oid)
CHECKIN(oid)
CHECKOUT(oid)
DELETE(oid)
ARCHIVE(oid)

NEW_CHECKPOINT takes an object identifier as argument and returns the object identifier of a new object. This new object (document) is initially the same as the old object; attribute values are copied and the content is shared. When the user starts editing the document, the affected components in the logical structure are automatically replaced by new versions. Replacing a component by a new version triggers the replacement of the parent component by a new version, up to the root component. The updates are performed on the new versions. In this way, the data sharing is maximal; it is essentially the same as used in the EXODUS system.

The new checkpoint is connected to the old checkpoint by means of a previous/next relationship. In this way, it is easy to go back in the derivation history of a document.

Note that new checkpoints can be derived from any existing checkpoint. The relationships between checkpoints therefore form a *tree*.

The effect of FREEZE is to make an object no longer revisable. Any attempts to update its attributes or content are blocked. However, frozen documents can be displayed, printed and used to derive new (revisable) checkpoints. At present, a FREEZE (of the old document) is triggered by NEW_CHECKPOINT, so that all internal nodes of the checkpoint tree are always frozen, and hence immutable, but FREEZE can also be done directly by the user.

Not all checkpoints are equally important in the project history. Usually, authors will work on a document for some time, and then decide to turn the last checkpoint into an edition. An *edition* is defined as a special checkpoint with a certain public relevance; it may be the checkpoint that is actually printed and shipped to the clients, or it is accepted by the author's manager. The operation CHECKIN is used to promote a checkpoint to the status of an edition. Editions have editionnumbers, so that it is easy to go through all editions of a document. Editions are always frozen, and, even more strictly, cannot be used to derive

new checkpoints. An explicit CHECKOUT command is needed beforehand.

DELETE just deletes a checkpoint. Any checkpoint can be deleted, unless it has been archived (by means of the ARCHIVE command).

Figure 2 gives an example of a document history. Note the difference between the derivation history and the edition history. Checkpoint 4 is the next edition of checkpoint 8, but is derived from checkpoint 2.

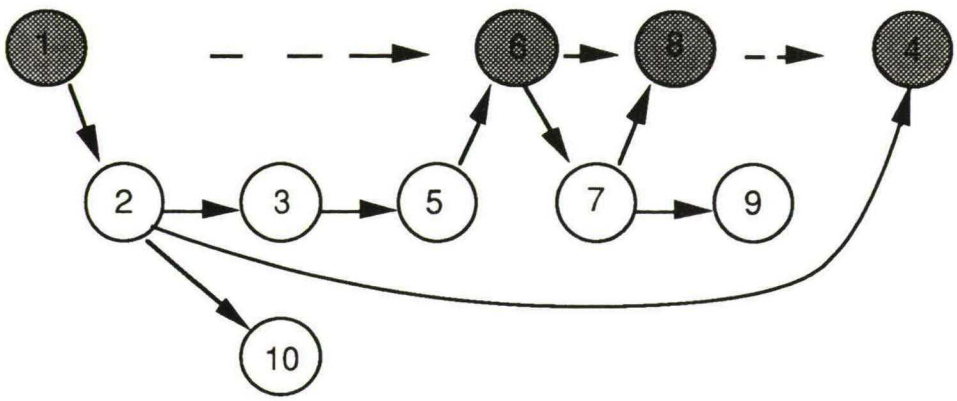

Figure 2: Checkpoint tree with editions

The history mechanism of SPRITE is especially aimed at documents, but we believe it also has a more general applicability. For example, instead of documents, we could also consider historical versions of employee objects in a personnel department. Each time the employee information is updated, for example, because of a new salary, a new version is created.

3.2. CONFIGURATIONS

SPRITE supports the creation and retrieval of documents describing different configurations of a product. Two perspectives are provided, one for authors or creators of configuration documents, called the aggregation view, and the other, the specialisation view, for those wishing to retrieve a configuration document.

Let us call the collection of documents for the different configurations of a product a document set. The individual documents in this set will be called final or configuration documents, to distinguish them from the objects from which they are built up, which are called building block documents. These terms will become clearer soon.

Our examples are based on documentation for a car. The car exists with automatic or manual transmission. Manual transmission cars have either four or five speeds, and come with either a sports-style gear stick or a standard one. See figure 3. Thus there are three configuration dimensions - "transmission", "speeds", and "stick", with the following domains of configuration values : {manual, automatic}, {4-speed, 5-speed}, {sports,

standard}. Speeds and stick are sub-configurations of the manual transmission configuration.

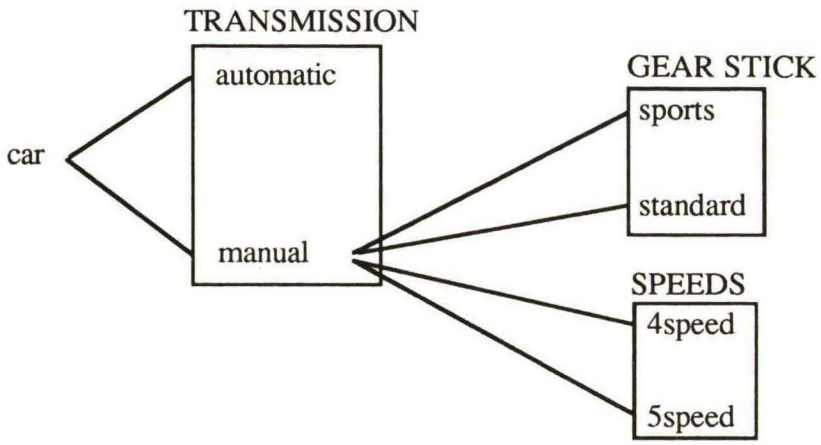

Figure 3: Configuration model example

All of the documents in the car manual document set will be largely the same. In some places they will be transmission specific. Similarly, the documents for manual transmission cars will have additional parts in common with each other, but will have some parts specific to the speeds dimension and other parts specific to the kind of stick.

The author's or creator's perspective of the SPRITE configuration model is of a configuration document as a complex object made up of common components and components dependent on particular configuration dimensions. The author creates all the components (chapters, text, graphics, ..) corresponding to one configuration value (e.g. the manual transmission, without any sub-configuration parts) in one building block document.

Each building block document is a separate identifiable object in the system, with its own history (as described earlier). It can be edited and manipulated separately from the other building blocks making up the document set. Separate building block documents can be assigned to separate authors.

The other important point about this model, from the point of view of creation and update of configuration documents, is that a particular component or sequence of components is stored in only one place, even if it appears in many of the final configuration documents. So when the picture of the sports stick needs to be changed in all the final documents, it is replaced in one place - in the sports stick building block document.

As well as containing content such as text and graphics, building blocks include information about how they fit together with other building block documents to create final documents. This is explained further in the next section. As an example the sports stick building block, the 5-speed building block and the manual transmission building block are combined with the building block containing content common to all the car manuals to

produce the final document for the sports stick, 5-speed, manual transmission car. It is this building up of final documents from building blocks which leads to the name "aggregation view" for this perspective.

The other perspective on the SPRITE configuration model is that of those retrieving configuration documents. In this case we are not concerned with the construction of the document, only with its content. In this perspective a document describing a specific configuration of a product is a specialisation of a document describing the generic product. For example the five speed manual transmission car manual is a specialisation of the manual transmission car manual which is in turn a specialisation of the ordinary car manual. Another way of looking at it is as inheritance - the more specialised document inherits all the components of the more general document, (probably) adding some of its own.

3.3. THE BUILDING BLOCK MODEL

In this section we explain more fully what a building block is and how it relates to other building blocks of the same document set.

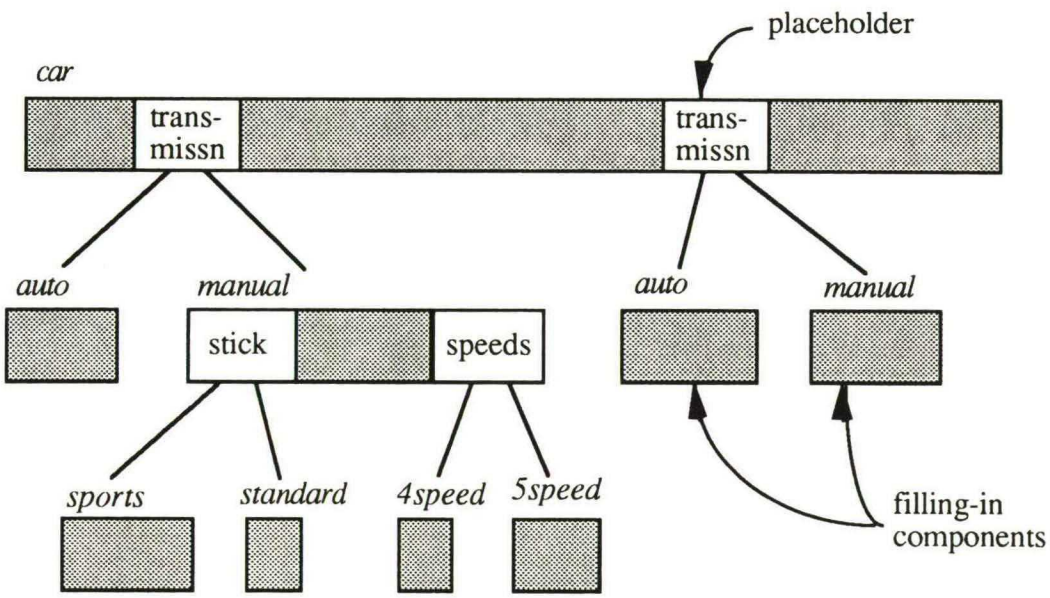

Figure 4: Placeholders and filling-in components for an example car manual

Figure 4 shows the conceptual structure of a sample manual for our car example. The top part of the diagram contains content common to all the car configuration documents. In two places final documents will contain transmission specific content. These are indicated by placeholders. Each placeholder is marked with a domain, in this case "transmission". At the next level we see the components which would replace these placeholders in final documents - one component for each placeholder for each dimension value. These are called filling-in components.

One of the manual transmission components has a speeds-specific part and a stick-specific part, each indicated by a placeholder. At the third level we see the filling-in components for these placeholders.

A building block is the collection of components for a specific configuration value. In Figure 5 we show the building blocks for the car manual document set. There are seven of them. Each one, other than the common one marked "car", is linked to a higher level building block. The links indicate which filling-in components fill which placeholders.

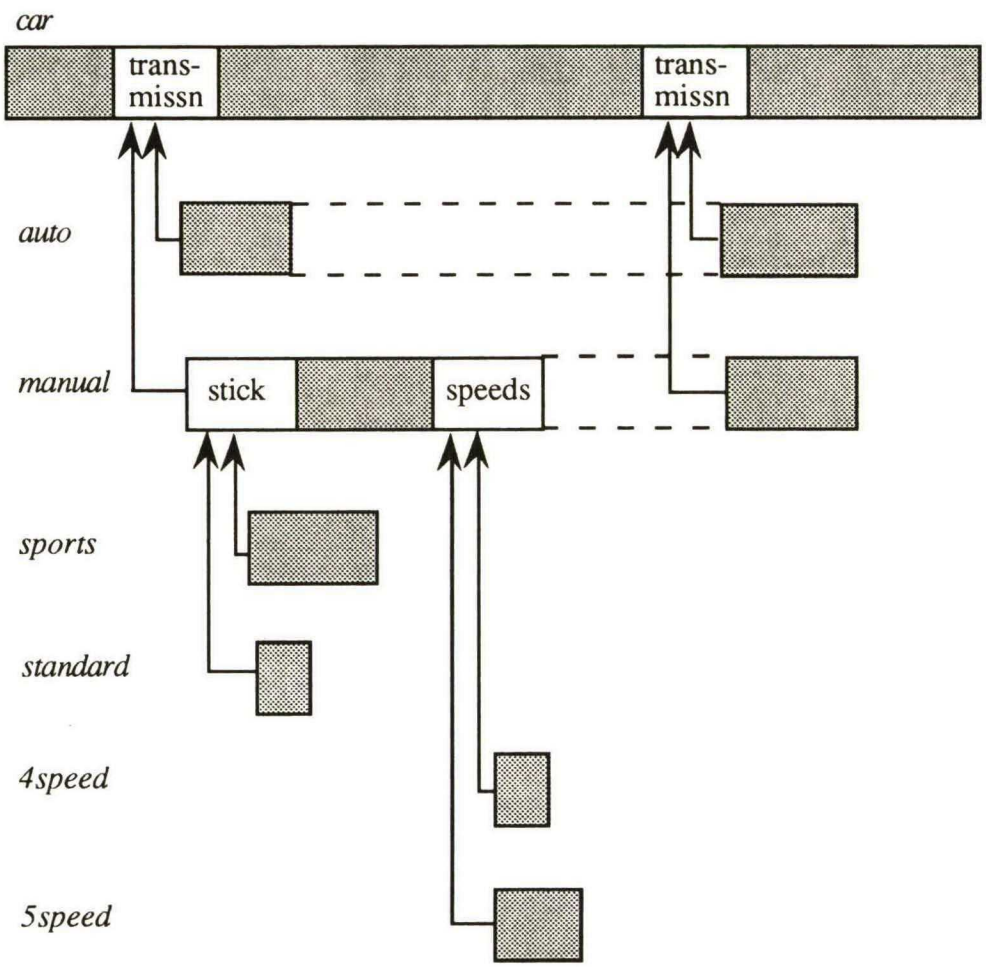

Figure 5: Seven building block documents for the example of figure 4

Final documents are produced by combining the set of building block documents corresponding to the desired configurations. Each placeholder in a higher level building block document is replaced by a filling-in component from a building block of the corresponding configuration dimension.

3.4. COMBINING CHECKPOINTS AND CONFIGURATIONS

As stated above, building blocks have their own checkpoint history. If we consider all checkpoints of building blocks, the configuration tree, as exemplified in Figure 5, becomes more complicated. For example, we may have three checkpoints of the 4speed building block, and two of the manual transmission building block. Furthermore, it may be the case that only the first checkpoint of the 4speed building block is compatible with the first checkpoint of the manual transmission building block, while the second and third checkpoints are compatible with the second. "Compatible" means here that for each filling-in component there exists one placeholder component in the higher level building block. The compatibility relationship is explicitly recorded and automatically updated by the system. In this way, the user can see immediately how a certain building block can be combined. The system also ensures that for each lower level building block at least one compatible higher level building block exists. If this integrity constraint were not enforced, we could get building blocks that could never be edited or printed, since editing and printing requires a context that specifies, for each filling-in component, its position in the logical structure of the document.

Usually, the user is not interested in all checkpoints and configurations but only in one particular checkpoint of each configuration. It is possible to keep record of the "currently active configurations" in a so called "composed document". This composed document is just a set of identifiers representing compatible building blocks. This composed document can be opened by the user when starting the editor. Composed document objects can be regarded as offering a simple idea of *context.*

3.5. COMPARISON WITH OTHER APPROACHES

Distinctive features of the SPRITE version model are:

* versioning is defined at the conceptual level;
* a clean distinction is made between historical versions and configurations;
* the version model is supported by an efficient storing mechanism;

As for the first feature, SPRITE differs for example from EXODUS (Carey et al, 1986) and the general version model presented in (Klahold et al, 1986). EXODUS supports basic versioning operations on the internal level, but leaves the conceptual level open. (Klahold et al, 1986) represent versions by means of version graphs and partitions. The user has the possibility to define several version graphs, and to insert versions into these graphs. To make one object a version of another object, the user must define an edge between the two nodes. Several version graphs can coexist, so that for example our version model could be implemented by means of a combination of a historical graph and a configuration graph. Partitions are used to class nodes of a graph together; for example, our notion of editions corresponds to a partition of the historical version graph. Although this mechanism could be used to implement our version model, it does not have the attached semantics.

A difference between configurations and histories is made in the VISION object-oriented

database system (Caruso & Sciore, 1988). The difference is here between TimeSeries and versions. TimeSeries are ordered and accessed on date; when an object (function) is declared as a TimeSeries, automatically all historial versions are stored (no explicit freezing is necessary). Versions are derived from objects by means of a "newVersion" operation, and versions can be frozen and reactivated. VISION also implements an interesting *context* mechanism; however, it is not a multimedia database and apparantly has no data sharing at the internal level.

4. Conclusion

In this paper, we described the SPRITE techical documentation system and in particular its versioning mechanism. SPRITE makes a distinction between historical versions and configurations. A special feature of this versioning mechanism compared with other approaches is that versions are defined at the conceptual level. It allows for efficient data sharing,but this occurs as a result of the user's modelling the document rather than by direct manipulation on data structure level. We compared SPRITE in this respect with a couple of other multi-media and/or object-oriented databases.

One limitation of the SPRITE version mechanism is that configurations must be fitted into a hierarchy. Multiple inheritance is not possible. Taking up the example of section 3, there is no place to put content that is specific to "5 gear, sport stick". This would require a building block that is both a subconfiguration of "5 gear" and of "sport stick". We have prohibited such situations in order to keep the system simple.

One interesting point of future research is to transfer this versioning mechanism from the particular case of the document to the general level of "object". For example, for a "person" we might also keep a history record, as well as different configurations. John may be both a teacher and a researcher; as a teacher, he earns 30K and as a researcher he earns 40K. Although for these cases the implementation issue of data sharing is less relevant (because the data items are not very big), there is of course the question of how to handle these versions cleanly on the conceptual level.

Acknowledgements

This work has been carried out under ESPRIT contract 2001 (SPRITE). The authors are grateful to all participants in this project, Oce-Nederland B.V., Alcatel TITN, AEG Electrokom Gmbh, Tilburg University and Trinity College Dublin. Special acknowledgements are due to Hans Daanen and Olga de Troyer for their significant contributions to the version model. However, nothing here should be taken to commit any partner, and the opinions and any errors are our own.

References

Andrews, T., C. Harris, K. Sinkel, 1989. *The ONTOS Object Database*. Ontologic.

Carey, M. et al, 1986. Object and File Management in the EXODUS Extensible Database system. *Proc. Int. Conf on VLDB*, Kyoto.

Caruso, M., E. Sciore, 1988. Contexts and MetaMessages in Object-Oriented Programming Language Design. *Proc. ACM SIGMOD*.

Christodoulakis, S. et al, 1986. Multimedia Document Presentation, Information Extraction, and Document Formation in MINOS: A Model and a System. *ACM Trans. on Office Information Systems*, 4(4).

Kent, W., 1989. An overview of the versioning problem. Panel Contribution. *Proc. ACM SIGMOD*.

Kim, W. et al, 1989. Composite Objects revisited. *Proc. ACM SIGMOD*.

Klahold, P., G. Schlageter, W. Wilkes, 1986. A general model for version management in databases. *Proc. Int. Conf. on VLDB,* Kyoto

Krönert, G., 1987. Standardized Interchange Formats for Documents. *ESPRIT'87*, North-Holland.

Walter, M.E., 1988. *Technical Documentation Systems*. Seybold Publications.

AN OBJECT-ORIENTED APPROACH IN A MULTIMEDIA DATABASE PROJECT

Hans Weigand
INFOLAB, Tilburg University
P.O.Box 90153
5000 LE Tilburg
The Netherlands
email: weigand@kub.nl

Abstract

On the basis of practical experience in a project on multimedia databases, we propose an object-oriented methodology for the conceptual database design. The conceptual specification language includes the notions of generalization, aggregation, versions and methods. The semantics of the language is described in dynamic logic.

Keywords: design methodology, versions, multimedia databases, dynamic logic

1. INTRODUCTION

Object-oriented data models (Banerjee et al, 1987) allow the definition of complex objects (Khoshafian & Copeland, 1986). An object has a number of attributes; the value of an attribute itself is an object. An object is an instance of a class; a class may be a primitive class without any attributes (e.g. integer, string), or may have any number of attributes. Object-oriented models recapitulate semantic data models used in databases. Primitive classes correspond to Lexical Object Types (LOT's) in the NIAM model (Nijssen, 1976). Other objects are called Non-lexical Object Types (NOLOT's). Peculiar features of object-oriented models are primarily the possibility to define complex objects, and the specification of operations in the form of methods.

This report is based on research currently performed on the design and implementation of a document processor aimed at technical documentation. Some special features of this document processor are:

- the use of historical versions of documents
- the integration of different media (text, graphics, raster)
- the possibility to define variants of documents
- the availability of links for hyper-text applications
- a screen-oriented browser
- integrated access to databases and CAD/CAM files

The implementation of the Document Processor is split up over three levels to enhance data-independence. The bottom layer is responsible for the storage of content and structure of the documents (for the structural information, a commercial relational database system is used, for content an Optical Disk Server). The middle layer provides a Multimedia Database (MMD). The MMD is based on a conceptual model of the document and the document space, and supports conceptually meaningful operations, such as CREATE document, INSERT document (in folder), SELECT (search condition) etc. On top of the MMD, a couple of applications is defined. These applications allow for, among others, editing the documents, browsing in the document space, printing and document acquisition (scanning).

In this paper, we will concentrate on the design of the MMD, which is based on a conceptual

model. For this model, a combination has been made between the NIAM methodology (Nijssen, 1976) and an object- oriented approach. NIAM is a semantic data model and as far as data structures are concerned, close to an object-oriented model. It has a graphical formalism which makes it easier to communicate the model between various partners of the project. Moreover, since we used a relational database for the physical layer, we could make profitable use of the existence of a database generation tool that maps NIAM schemas to table definitions (with all the necessary constraints - see DeTroyer, 1989).

We extended NIAM in an object-oriented way for the following reasons. First, for the design of the MMD interface we needed also to define the dynamics of the model. The object-oriented methodology allows a clean description of the dynamics by means of methods attached to the object types which we will describe shortly. Secondly, adopting an object-oriented methodology is helpful for attaining extensibility, since it makes the dynamic addition of new object types and methods easier. Thirdly, the objects we are dealing with show features that tend to be built-in in object-oriented database systems:

* the central object type (the document) is a complex object consisting of several text and picture components. For the reasons described above, sharing of parts of this component structure is very important because these parts may be very large. NIAM itself does not incorporate an aggregation concept yet explicitly. The aggregation hierarchy is not only useful for sharing, but also as a unit of clustering, as a unit for retrieval, as a unit for authorization, and as a unit for locking in the context of concurrent access (cf. Kim, 1989);
* documents may have versions, which is also a feature of advanced object-oriented systems;
* the document processor allows also the creation of new document types. Examples are "manual", "letter" etc. When a document is created as instance of a certain type, it inherits the style, and maybe some content (for example, a front page).

An object-oriented model was also adopted by the MINOS system (Christodoulakis et al, 1986), a prototype MMD system developed at the Universities of Toronto and Waterloo.

On the other hand, our database (design) differs from what is sold nowadays as "object-oriented databases" at least in the following two respects:

(i) we use an RDBMS, extended with an Optical Disk server, for physical storage. That is, a document, as a typically aggregated object, is not *stored* "object-oriented". We do make use of the aggregation concept for clustering, but not as radical as this is done in "object-oriented databases". This is not only because an RDBMS is more readily availabe, but also because it is not clear whether an object-oriented *storage strategy* gives an overall better performance.

(ii) instead of using an (imperative) OO programming language for the specificaiton of dynamic behaviour, we tried to develop a *declarative* style based on dynamic logic.

This paper is organized as follows. In section 2, we describe the MMD specification language with a special emphasis on the operations dealing with versions. In section 3, we describe the semantics by means of an interpretation into dynamic logic. Section 4 contains some conclusions and topics for future research.

2. OVERVIEW OF THE MMD SPECIFICATION

The MMD is a well-defined level of the Document Processor. For the design, we have developed a specification formalism from which the following language is derived. The specification now exists only on paper and is used as the basis for the implementation. We intend to automate the implementation process later. Then we would have a self-contained application-independent Object Oriented Multimedia Database Layer built around a relational DBMS. In the mean time, the formalism serves well as an interface between the programmers and the application developers.

2.1 Static part

The static part of the specification consists of class definitions. We restrict ourselves here to the written form, ignoring the NIAM pictures which have been developed in parallel. A class definition contains an object type name, a list of attributes, and generalization information. The lexical object types are supposed to be predefined. Some examples:

```
Class document space object
Attributes
        name: string;
        owner: user;
Subtypes
        dso_type: {folder, document, document_list}
end;
Class folder
isa     document space object
Attributes
        content: set_of document space object independent
Functions
        size: integer
                = COUNT(content($i_self))
end;

Class user
Attributes
        name: string;
        real_name: string;
        password: string;
        home-folder: folder
Keys
        {name}
end;

Class group
Attributes
        name: string;
        content: set_of user independent
end;
```

Folder is defined as a subclass of "document space object", and so it inherits the attributes "name" and "owner". A folder has an attribute "content". In our language, this is a special attribute with special semantics, used for the aggregation abstraction. The special semantics include:

* the optional use of keywords *independent* and *exclusive* (cf. Kim, 1989). When the composite reference is dependent, this means that the deletion of the complex object causes the deletion of the component if this component is not contained in any other complex object of the same type. It also implies that the object can only be updated in context. A reference is exclusive if a component may occur in at most one complex object. By default, the component reference is dependent and not exclusive.

* the system-defined operations INSERT, REMOVE, and MOVE immediately apply.

In contrast to database models, it is not necessary to specify the key, or the candidate keys, because each non-lexical object is uniquely identified by an object id (unless it is an aggregation - see below). The definition of keys is optional and imposes an extra integrity constraint on the

model. For example, names of users must be unique within the system. Keys can be used for quick reference. If "hansw" is the name of a user, then @hansw is the user with name "hansw".

Functions can be defined on classes behaving like derived attributes. They can be accessed in the same way as ordinary attributes but they cannot be updated. When subtypes are defined for an object type, the subtype group, or criterion, should be given which also can be queried as an attribute. An object type may have more than one subtype group.

Sometimes, we need complex objects that are existence-dependent on their components. We call such objects *aggregations*. For example, in our model we also have a class "document list", which can be used to group several documents together, for example, the respective editions of a manual. The elements of these list are not just documents, because we want to add some additional information to them, in particular, some edition number and a date of insertion. Therefore we define the class "document entry". A document list is a list of document entries. A document entry can be defined as:

Class document entry
Aggregation_of
document;
label: string
Attributes
inserted_at: date
end;

This means that document entries can only exist as combinations of document and label. When the document is deleted, the document entry is deleted as well (but not the other way round).

2.2. Dynamic part

For each object type declared, the functions CREATE and DELETE are predefined. For sets and lists, we also have the functions INSERT and REMOVE. For example, if "owner" of "document space object" would have been defined as a set of users, the operation (message)

[dsoY INSERT owner userX]

inserts the user userX in the set of owners of instance dsoY. As we noted above, the "content" attribute has a special treatment. In that case, we do not need to give an attribute:

[folderY INSERT dsoX]

This could be viewed as a shorthand of:

[folderY INSERT content dsoX]

2.2.1 Versions

Special semantics are included for versioned objects. These must have been defined as such. For example:

Class document
isa document space object **versionable**
Attributes
title: string;
style: StyleObject;
content: composite component **exclusive**;
authors: **set_of** user
end;

Class component
isa object **versionable**

Attributes

Subtypes
{component_type: composite component, basic component}
end;

Class composite component
isa component
Attributes
content: component **dependent;**
title: string **optional;**
end;

Class basic component
isa component
Attributes
data: **set_of** character
end;

When an object is versionable, the following operations are provided:

FREEZE marks the object as immutable. No changes are possible anymore (but the object can still be deleted).

NEW creates a new "checkpoint" of the object. The new checkpoint is a duplicate of the parameter object. Versioned objects inherit two system-defined attributes: "generic_id" and "version_nr". These two attributes together form a key of the object.

If d1 is a document, then the operation

[d1 NEW]

where generic_id(d1) = x and version_nr(d1) = n, results in a new document, d2 let`s say, with generic_id(d2) = generic_id(d1) = x, and with a different version_nr (the version_nr is not necessarily n+1, since the versions can form a tree - it is possible to make new versions from any old version). The id of the new object, which is a duplicate of d1, is returned. Versioned objects do also inherit a system-defined attribute "previous_version", which for d2 in this case is set to d1.

The operation FREEZE is used for making the object ineditable. In the Document Processing system, a NEW always triggers a FREEZE of the old version, but this is not necessary. To know whether an object is frozen or not, it is possible to ask for the "edition_status", which is either "revisable" or "final". Again, this attribute is system-defined. Any version mechanism must also specify whether new versions of components are made when a new version of the complex object is generated. Obviously, when the composite reference is exclusive, new versions must be made. In the case of non-exclusive composite reference, we have decided to defer the generation of new component versions. Since documents may be very large, it is necessary to share content between versions as much as possible. We use the mechanism of "deferred copy" to achieve this. This means that the two versions share their content. Now if the old version is frozen and the user starts editing the new version, new versions must be made from the components that get changed, and from their parents (up to the root). The updates are performed only on these new components. Components that are not affected by the update remain unchanged (and shared). The mechanism used is essentially the same as in the EXODUS system (Carey et al, 1988).

Note that in the specification above, both documents and components are defined as versionable objects. If components were not versionable, the mechanism described above should make copies of the components to be edited.

The individuation of versions has been brought forward before as a problem (Kent, 1989). Is each document version a separate document, or is there actually one document? In our present conceptual model, each version is considered to be a document subsuming one generic document. However, we would not like to treat all versioned objects in this way (for example, persons). In general, the individuation question depends on the entity type. In (Weigand, 1990) we made a distinction between first-order entities, such as persons, second-order entities, such as States of Affairs, third-order entities, such as propositions, and fourth-order entities, such as sentences and texts. Roughly said, it is only for the third- and fourth-order entities that each version is to be taken as an individual object. For the other entities, we count only one object. In some cases, a predicate is ambiguous: for example, we must distinguish the predicate "book" as a physical entity (first-order) from the predicate "book" as text (fourth-order entity).

2.2.2 Attribute update operations

The language includes a general SET command which can be used to update the value of an attribute. Examples:

[x1 SET owner @hansw]
[d1 [c1 SET title "introduction"]]

The first message is a request to object x1 to set the owner attribute to (the user with name) hansw. The second message shows the use of aggregation. It is a request to document d1 to request to component c1 to set the title to "introduction". This is the way to update some object in the context of a complex object. We recall that if the the attribute is set-valued, the operations INSERT and REMOVE are available as well.

2.2.3 Schema evolution

The language allows the creation of new types by means of the command CREATE_TYPE. The new type can be created as a subtype of an existing type, in which case attributes are inherited. The new type can be edited. If it is versioned, it can also be frozen, and new versions can be made from it. We allow the creation of instances only for frozen or non-versionable types. In the Document Processor, the CREATE_TYPE is used for defining document types such as manuals, or memo's. The document type specifies, among others, the possible styles of the document (page size, font, font size etc).

2.2.4 Variants

The MMD also allows variants of documents to be created, by means of the command MARK_CONFIGURATION, which takes as arguments a document, a configuration dimension (for example, "terminal") and a domain (for example, vt110, vt220). This means that the document exists in two variants, one for "terminal=vt110" and one for "terminal=vt220". The user should indicate which components of the document are shared, and which ones are configuration-dependent. We intend to present the variant mechanism in a separate paper (see Hederman & Weigand, 1990).

2.2.5 User-defined dynamic constraints

The operations CREATE, INSERT, FREEZE etc do already provide the user with a powerful general operation language. Application-dependent dynamic constraints can be defined in methods. We distinguish preconditions, postconditions and triggers. Preconditions are conditions that must be fulfilled before the operation can be executed. For example, a component can be removed from a document only if the document has editionstatus "revisable". Postconditions are conditions that are true immediately after the operation. For example, when a document is created, the owner of the document is the current user, and the document must have at least one component. Triggers are operations that should occur afterwards. For example, when a document is deleted, all links with other documents must be deleted as well. We also allow the definition of "before-triggers". These triggers are executed before the operation. In general, the methods are defined by forms with the following structure:

```
NAME
        CREATE
CLASS
        document
PARAMETERS
        $i_name: string
        $i_folder: folder
RETURN
        $o_id: document
DESCRIPTION
        Create a document with name $i_name in folder.

PRECONDITIONS
C1      NOTEXIST $I_doc contained_in $i_folder
        WHERE name($i_doc) = $i_name
C2      ...

LOCAL CONDITIONS
L1      ...
L2

POSTCONDITIONS
P1      EXIST $o_id
P2      name($o_id) = $i_name
P2      owner($o_id) = $g_user

TRIGGER BEFORE
D1      ...

TRIGGER AFTER
T1      ...

ERRORS
        EC1     Name not unique within folder
        EC2     ...

REMARKS
        ...
END
```

The meaning of these specifications will be described precisely in section 3. Intuitively, PRECONDITIONS contains necessary preconditions of the method, and POSTCONDITIONS describes the effect. Both are in the form of first-order formulae, possibly open in the parameter variables. For each precondition, we have one corresponding error specification. If the precondition is not met, the corresponding error message is returned. The TRIGGER BEFORE and TRIGGER AFTER are calls to other methods, to be executed before or after the operation. Finally, the LOCAL CONDITION part can be used to set the value of local variables to be used in the postconditions. For example, consider a method for BIRTHDAY:

```
NAME
        BIRTHDAY
CLASS
```

```
        person
PARAMETERS

DESCRIPTION
        BIRTHDAY increases the age of the person.

LOCAL CONDITIONS
L1      age($i_self) = $l_n

POSTCONDITION
P1      age($i_self) = $l_n + 1

END
```

This operation specifies that if the age of person is n, then after BIRTHDAY, the age is n+1 (note that the input parameters are prefixed with $i, and the local variables with $l - note also that the object of the class to which the message is sent, need not be mentioned explicitly in the parameter list). The two examples, CREATE_DOCUMENT and BIRTHDAY exemplify two kinds of specifications. CREATE_DOCUMENT is an example of a method that refines and replaces an already defined method (the standard CREATE). BIRTHDAY is a method defined independently.

2.3 Retrieval

To achieve full data-independence, a query language is defined that hides the relational table definitions, and that also supports the generalization and aggregation abstraction. For practical purposes, we have chosen for a format that remains close to SQL. The following syntax defines the first version of this query language:

```
-----------------------------------------------------------------
<S>   ->    SELECT <A>* FROM <V> <T> WHERE <C>
<A>   ->    <ATTRIBUTE>
<V>   ->    [A-Z][0-9]*
 <T>  ->    <TYPE>
<C>   ->    ( <C1> (AND | OR ) <C> | <C1> )
<C1>  ->    ( "(" <C> ")" | <E> )
<E>   ->     <L> ( "=" | ">" | "<" | "!=" | IN | CONTAINED_IN ) <L>
<L>   ->    ( <CONSTANT> | <V> | <M> | <F> "(" <L> ")" )
<F>   ->    ( <ATTRIBUTE> | <FUNCTION> )
<M>   ->    "(" <Q> <V> <T> WHERE <C> ")"
<Q>   ->    ( SOME | THE )
-----------------------------------------------------------------
```

- The query takes the form of a SELECT statement, where <T> specifies the type domain, <C> the search condition, and <A> the attributes that must be retrieved. A variable <V> must be supplied so that it can be used in the search condition. Example:

```
SELECT real_name
FROM X user
WHERE name(X) = "maria"
```

which returns the real_name of the user(s) with name "maria". Now a more complicated example:

```
SELECT      name title
FROM        D document
WHERE       owner(D) = (THE X WHERE name(X) = "maria") AND
```

(SOME C figure WHERE caption(C) = "NIAM picture 1")
CONTAINED_IN D

This query returns the name and title of documents owned by "maria" and containing some figure with caption "NIAM picture 1". The search condition contains two terms (type <M>). One is a definite term with variable X, which refers to user "maria", and the other one is an indefinite term with variable C, which refers to a figure (subtype of component) in the document. A term corresponds to a subquery in SQL. Note the use of the CONTAINED_IN, which takes the transitive closure of the function content (content() + content(content()) + ...). This is a useful construct to support the aggregation abstraction.

- Some integrity constraints on the syntax must be supplied. The most important one is that each atomic part of the condition must contain at least one reference to the search variable. The same condition holds for the condition of the embedded terms.

- The negation is present only in the "!=" comparison operator. It is required that at least one side of the inequation is a constant. This is too restricted, but it makes the interpretation much easier.

- The present version of the query language does not support yet the upwared inheritance of attributes along the aggregation dimension. In several object-oriented query languages, this is supported by a dot notation. For example, ..font can be used as an attribute of document when it is in fact an attribute of the style of the document. We hope to include such semantics later.

3. SEMANTICS

The semantics of the retrieval language is not worked out here. For the semantics of the specification language, we make use of dynamic logic. Dynamic logic has been used before for the definition of integrity constraints (Khosla et al, 1986; Wieringa, Meyer, Weigand, 1989; Wieringa et al, 1989). We briefly review the main concepts. The syntax of the specification language, with AND, OR etc, is already logic-based. The interpretation of this part of the syntax is therefore omitted. The same holds for complex term expressions, for example, expressions with arithmetical operators. We concentrate on those points of the language that are really different.

3.1 Dynamic logic

Dynamic logic is an extension of first-order logic that can be used profitably for integrity constraint specification. For static constraints, we can simply use the first-order part. An example is the constraint that birds are warm-blooded. For dynamic constraints, the language is extended as follows. We assume a fixed set A of atomic actions, and then define the language L-Act of actions as elements of A, the non-deterministic choice of two actions, represented as a1 | a2, the parallel execution of two actions, represented as a1 & a2, the non-performance of the action a, represented as NEG a, and the special constants ANY (denoting the unspecified action) and FAIL (denoting the empty action). For the semantics, we assume an S5 Herbrand-Kripke structure PW which can be viewed as a collection of Herbrand structures which are called worlds or states. The semantics of actions is that they are functions on PW. Note that if an action changes a world, it does so instantaneously, i.e. there are no intermediate worlds during the execution of an action. A sequence of action steps is called a transaction. Now the extension of first-order logic L-Dyn is basically the addition of formulas of the form

[t] f

where t is a transaction and f some well-formed formula, with the intended meaning that f is true AFTER t. An example of a dynamic logic expression is

FORALL e: NOT employee(e)
=> [hire(e)] employee(e)

(we use the verbalizations FORALL and NOT for the corresponding logical operators; "hire()" is supposed to be an atomic action, and employee a predicate; for the use of parameters in actions, see Dignum, 1989). The meaning of this formula is that for all e, if e is not an

employee then he is an employee AFTER the performance of the action "hire". Dynamic logic can be used again to implement a variant of deontic logic. The interested reader is referred to the afore-mentioned publication, and to (Fiadeiro et al, this volume) for application in software specification. We specify here only the representation of objective modalities:

```
POS(a) <=> NOT [a] false     /* possible
NEC(a) <=> [NEG a] false     /* necessary
DIS(a) <=> NOT NEC(a)        /* discretionary
IMP(a) <=> NOT POS(a)        /* impossible
```

Moreover, if a1 and a2 are actions, then a1 >> a2, pronounced as: "action a1 implies action a2" is defined as:

a1 >> a2 <=> IMP(a1&NEGa2)

That is, a1 cannot be executed if a2 is not executed in parallel.

In (Wieringa et al, 1989), special attention was given to the inheritance of dynamic and deontic integrity constraints. For this purpose, the language was endowed with two special predicates. First, the unary predicate E for existence, indicating which entities exist in the world in question. Second, the binary predicate TYPE, where the second argument can only be filled with a type name (of set T), such as "bird" or "employee". If t1 and t2 are type names, we use the abbrevation

t1 < t2

to say that t1 is a subtype of t2, that is,

FORALL x: TYPE(x,t1) => TYPE(x,t2)

3.2 Further extensions to the dynamic logic

In order to be able to give compositional semantics from our object-oriented specification language to dynamic logic, we define the following useful structures. We define a special ternary predicate ATT, where the first and third argument is always a type name, and the second argument an attribute name (of set AT). Moreover, a ternary predicate VALUE, where the second argument is always an attribute name. In this way, we arrive at a semi-second order logic where we can quantify over attributes. Some examples of well-formed atomic formulae:

```
ATT(document, title, string)
ATT(document, author, person)
ATT(person, age, integer)
VALUE(c1, title, "An object-oriented approach in an MMD project")
VALUE(c1, author, c2)
VALUE(c2, age, 30)
```

where

AT = {title, author, age, owner, ...}

The following axioms must be defined in L-Dyn:

```
A1   FORALL e1,a,x: (TYPE(x,e1) AND AT(a) AND EXIST y: VALUE(x,a,y))
     => (EXIST e2: ATT(e1,a,e2))
A2   FORALL e1,e2,x,a,y: (AT(a) AND T(e1) AND T(e2) AND ATT(e1,a,e2) AND
                          TYPE(x,e1) AND VALUE(x,a,y))
     => TYPE(y,e2)
```

The first axiom says that if some object has some value for some attribute, then this attribute must have been defined for the object type. A2 requires that the attribute value of an object falls within the domain of the attribute. Additionally, we need axioms to inherit attributes from supertypes to subtypes. These are not specified here.

3.3 Semantics of class definitions

With the TYPE and ATT predicates defined above, the translation of class definitions to dynamic logic is straightforward. Each class name is mapped by the interpretation function I to one type name in T. Each attribute name is mapped to one attribute name in AT. If c1 is a subclass of c2, then I(c1) < I(c2), that is, they are subtypes. If c1 has attribute a1 with domain c2, then ATT(I(c1),I(a),I(c2)). The "content" attribute is mapped in a special way (see 3.6). The definition of TYPE and ATT is such that the inheritance of attributes follows as a theorem. Let us define the predicate POSS_ATT(,) for specifying the possible attributes of an object. That is,

```
FORALL x,a: POSS_ATT(x,a) <=>
        (EXIST t,c: T(t) AND TYPE(x,t) AND ATT(t,a,c))
```

Then, we can easily prove that:

```
T1    FORALL x,t1, t2: (T(t1) AND T(t2) AND t1 < t2 AND TYPE(x,t1))
      => (FORALL a,c: (ATT(t2,a,c) => POSS_ATT(x,a)))
```

3.4. Semantics of objects and versions

For the representation of objects and versions, we assume that our language L-Dyn contains the natural numbers as constants (cf. Wieringa, Meyer, Weigand, 1989). We define the function symbols id() and vid(,). By taking the natural numbers as arguments, we have infinitely many terms id(1), id(2), .. . We keep a pre-defined "counter" predicate "last_id". Now we assume the atomic action CREATE'() in our action set A with the following properties:

```
A3    FORALL n,m: (last_id(n) AND m > n) => NOT E((id(m))
A4    FORALL n,t: last_id(n) AND T(t)
      => [CREATE'(t)] (last_id(n+1) AND E(id(n+1)) AND TYPE(id(n+1),t) )
A5    FORALL n,m: (last_id(n) AND last_id(m)) => n=m
A6    FORALL x,n: (x = id(n)) <=> generic_id(x)=n)
```

A3 says that objects beyond the last_id do not exist yet (in this world). Note that the extension of last_id may differ from one world to another. A4 says that the effect of the CREATE' (with argument "type name") is that the "last_id" is incremented with one, that there exists now an object identified by id(n+1) of the specified type. A6 defines the system-attribute "generic_id" which for non-versioned objects is just the inverse of id(). Similarly, we have an atomic action DELETE'() with properties:

```
A7    FORALL x: E(x) => [DELETE'(x)] NOT E(x)
A8    FORALL x: NOT E(x) => IMP ( DELETE'(x) )
```

A6 simply states that an existing object does not exist anymore after the DELETE. A7 states that it is impossible to delete a non-existing object. For versions, we use the same mechanism with function v_id(,). This function takes two arguments, the first being a generic id and the second being a version number. In analogy with the predicate last_id, we define a function last_version() which returns the last version number of the object. We assume an atomic action NEW'() in A as follows:

```
A9    FORALL x,m,t: ((TYPE(x,t) AND E(x) AND last_version(x) = m)
      => [NEW'(x)] (last_version(x) = m+1 AND E(v_id(generic_id(x),m+1))
                    AND TYPE(v_id(generic_id(x),m+1),t)))
```

In words: after the NEW' operation, there exists one more version of the object, of the same type. In fact, this is still to weak: what we want in addition is the following. For convenience,

we define the term "new(x)" as an equivalent of "v_id(generic_id(x), last_version(generic_id(x)))".

A10 FORALL x,t,a,v: ((E(x) AND TYPE(x,t)) AND AT(a) AND VALUE(x,a,v))
=> [NEW'(x)] (VALUE(new(x),a,v) AND previous_version(new(x),x)))

This says that the new version of the object has the same attribute values as the object itself, and the previous_version of this new version is the object. Previous_version(,) is another system-defined attribute, which just like "generic_id" and "version_nr" is not treated as the normal user-defined attributes. The use of the two functions id and v_id(,) needs some justification. In (Wieringa, Meyer, Weigand, 1989) functions were restricted to so-called transparant functions, which roughly means that the output of the function must be an already existing constant. In this way, no new ground terms have to be added in the construction of the Herbrand Universe. So the functions id() and v_id(,) must be transparant too. To show that it is possible to define them so, it is sufficient to show how they could be mapped to the set of natural numbers. A numerical solution could be to map id(n) to the n-th prime number, and use the exponentials of these primes for the versions v_id(n,m).

3.5 Semantics of methods

The semantics of methods is given as follows. First, CREATE and DELETE can be mapped to the atomic actions CREATE' and DELETE' above. Similarly, for NEW we can use the NEW' action. The attribute update operation (2.2.2)

[$i_object SET $i_att $i_val]

is interpreted by a function SET'(I($i_object,I($i_att), I($i_val) given by:

A11 FORALL x,a,v: [SET'(x,a,v)] VALUE(x,a,v)

(the meaning of I($i_val) should be specified again compositionally, if $i_val is a complex expression - see DeBakker, 1980). For the user-defined methods, we must make a distinction between methods that refine system-defined methods, such as CREATE_DOCUMENT in 2.2.5, and independent ones, such as BIRTHDAY. For independent methods, the meaning is as follows. Let m be the method, then I(m) is an atomic action in L-Dyn, with the following axioms (in fact, axiom schemata):

A12 FORALL m(p): POS(m(p)) => I(PRECONDITIONS(m(p)))
A13 FORALL x,p: I(LOCAL CONDITIONS) => [m(p)] I(POSTCONDITIONS(p+x))

Axiom schema A12 (we need one for each method m) specifies that the preconditions of m are necessary preconditions, that is, the action is only possible if the preconditions are met. The "p" in this formula should be read as a vector of variables, one for each input parameter of the method. Similarly, "x" in A13 is a vector of local variables. A13 specifies that after the execution of the action, the postconditions hold. For convenience, we take the parameter definitions and the preconditions together. For example, for the BIRTHDAY method we get:

(12) FORALL $i_p: POS(BIRTHDAY($i_p)) => TYPE($i_p,person)
(13) FORALL $l_n, $i_p: VALUE($i_p,age,$l_n)
=> [BIRTHDAY($i_p)] VALUE($i_p,age,$l_n +1)

In (12), the PRECONDITIONS part contains only the parameter specification that the input parameter must be of type person. In (13), the LOCAL CONDITION is put at the left-hand side, and the postcondition (with [m]) at the right-hand side of the implication. In the case of refined operations, triggers can be defined. Let m1 be some primitive operation (system-defined or user-defined), mapped to atomic action I(m1) in A. Let m2 be a refinement of m1, and let the TRIGGER BEFORE of m2 be "b" and the TRIGGER AFTER be "a" (both a and b may be empty). Then the interpretation of m2 is the transaction tm, defined as:

I(m2) = tm = I(b); I(m1); I(a)

that is, the action I(m1) preceded by the action I(b) and followed by action I(a). The interpretation of the pre- and postconditions for this method is the same as for primitive operations. The only difference is that I(m2) is a transaction rather than a primitive action. If a is empty, then I(a) is the identity action, the action that maps each world to itself. This interpretation of refined methods does justice to the intuition that the refinement must "include" the original. However, in general there will be no simple relationship between the postconditions of the original and the refinement. This is because the TRIGGER AFTER (c.q. I(a)) may partly undo the effect of I(m1). So it is not always the case that the postconditions of m1 are still valid after the transaction tm.

3.6 Semantics of aggregation

For the aggregation abstraction, some special semantics are needed. In the first place, we have the primitive operations INSERT and REMOVE that work on sets. We assume corresponding atomic actions INSERT' AND REMOVE' defined by:

A14 FORALL x,a,v: [INSERT'(x,a,v)] VALUE(x,a,v)
A15 FORALL x,a,v: [REMOVE'(x,a,v)] NOT VALUE(x,a,v)
A16 FORALL x,a,v: POS(REMOVE'(x,a,v)) => VALUE(x,a,v)

The last axiom says that REMOVE' is only possible if the element is first included in the set. Note that in L-Dyn, attributes may be multi- valued. If an attribute can only have one value, we must specify

A17 FORALL a,t: (AT(a) AND T(t) AND Single_valued(t,a)) <=>
(FORALL x,v1,v2: (TYPE(x,t) AND VALUE(x,a,v1) AND VALUE(x,a,v2))
=> v1=v2)

and mark all single-valued attributes as "Single_valued". The content attribute has a special meaning in our language, since it signals a composite reference. The composite reference can be exclusive and/or dependent. The meaning of these is as follows. We do not translate this attribute to some attribute name in AT, but to a special binary predicate CONTAIN(,). At the type level, we introduce the predicate COMP_REF(,,), where the first argument is some object type (the composite object), the second argument is some object type (the component object) and the third argument indicates the kind of composite reference, of the set {excl, dep, non-excl, indep}. In this way we can interpret the composite reference between "document" and "component" (2.2.1) as:

COMP_REF(I(document),I(component),dep)

(if the reference was exclusive also, we have to add a second clause). If document d1 contains component c1, this is represented as:

CONTAIN(d1,c1)

To get the complete content of a complex object, we need the transitive closure of CONTAIN, as follows:

A18 FORALL x,y: REC_CONTAIN(x,y) <=>
(CONTAIN(x,y) OR (EXIST z: CONTAIN(x,z) AND REC_CONTAIN(z,y))

The predicate REC_CONTAIN corresponds to (the inverse of) the operator CONTAINED_IN defined in our specification language. When a composite reference is exclusive, components can occur in at most one complex object:

A19 FORALL x,y: (TYPE(x,e1) AND TYPE(y,e2) AND COMP_REF(e1,e2,excl)
AND CONTAIN(x,y))
=> (NOT EXIST z: CONTAIN(z,y) AND z != x)

When the composite reference is dependent, the component must be contained in at least one complex object. It is deleted if it does not occur in any complex object. This is accomplished by

the following extra axiom for DELETE'

A20 FORALL x,y,e1,e2: (CONTAIN(x,y) AND TYPE(x,e1) AND TYPE(y,e2) AND
(NOT EXIST z: CONTAIN(z,y) AND z!=x)
AND COMP_REF(e1,e2,dep)
=> (DELETE'(x) >> DELETE'(y))

and of course we also have:

A21 FORALL x,y: CONTAIN(x,y) => [DELETE'(x)] (NOT CONTAIN(x,y))

The effect of A20 for exclusive composite references is that the component is always deleted when the containing object is deleted since there cannot exist another object containing the component. For non-exclusive composite references, it depends. Note the use of the imply symbol for actions ">>". This formalation is preferable over a specification where the conseqent is:

=> [DELETE'(x)] (NOT E(y))

since other axioms may add additional semantics to the DELETE'(y) which would be missed otherwise.

Axiom A21 is simple: if x does not exist anymore after the operation, of course it cannot contain any object. Such existence axioms are needed for all predicates, not just for CONTAIN. When the composite reference is exclusive, and the object is versionable, new versions of components must be made when the complex object gets a new version:

A22 FORALL x,y,e1,e2: (TYPE(x,e1) AND TYPE(y,e2) AND
COMP_REF(e1,e2,excl) AND CONTAIN(x,y))
=> [NEW'(x)] (E(new(x) AND E(new(y) AND CONTAIN(new(x),new(y)))

When the composite reference is non-exclusive, and the object is versionable, the new version shares content with its predecessor.

A23 FORALL x,y,e1,e2: (TYPE(x,e1) AND TYPE(y,e2) AND
COMP_REF(e1,e2,non-excl) AND CONTAIN(x,y))
=> [NEW'(x)] CONTAIN(new(x),y)

This is the analogon of A10, which specified that attribute values are the same for the new version. However, when the two versions share content, new versions of components must be made as soon as some update operation is performed on the component. Such an update is made in the context of the complex object. The semantics of such contextualized updates becomes rather complex and we give only the rough outline here. The basic idea is that, if

m = [x [y <op>]]

is some contextualized update, where <op> is some operation, and y is in the content of x (dependent composite reference), then the interpretation of m is a transaction tm in which first a new version of the object y is made before the operation is performed (on the new version). That is, if y is a shared component,

tm = (NEW'(y); I(op){new(y)/y})

where s{a/b} should be read as: "s in which all references to b are replaced by references to a". If y is not a shared component, the interpretation is of course the same as for not contextualized updates. The general interpretation is the disjunction of these two cases, where each case starts with a condition. We cannot express that in the L-Dyn as we introduced it above; see (Harel, 1984) and (Meyer, 1988) for conditional actions in dynamic logic.

Example. Let m be the operation

[d1 [c1 [SET title "Introduction"]]

Then the interpretation becomes, if c1 is a shared component,

(NEW'(c1); SET'(new(c1),title,"Introduction")

However, this is still not correct, since we must make a new version of the component c1 ONLY in the document d1. Hence we also need a revised version of the NEW'(). In the above, we did not yet specify the effect of this operation for component objects. For that purpose, we must extend the NEW'() to a two-place operator NEW_DEP'(,), defined as follows:

A24 FORALL x,y: CONTAIN(x,y)
=> [NEW_DEP'(x,y)] (E(new(y) AND NOT CONTAIN(x,y) AND
CONTAIN(x,new(y)))

In other words, the effect of the NEW_DEP' is the same as for NEW' (we should repeat A9 here completely), but at the same time it replaces the occurrence of y in complex object x by its last version new(y). So the interpretation of contextualized operations should be:

tm = (NEW_DEP'(x,y); I(op){new(y)/y})

Now the interpretation works well on one level of the aggregation hierarchy. We refrain here from generalizing it to arbitrary levels. The operation NEW_DEP is not possible for exclusive references:

A25 FORALL x,y,e1,e2: (TYPE(x,e1) AND TYPE(y,e2) AND CONTAIN(x,y)
AND COMP_REF(x,y,excl))
=> NOT POS(NEW_DEP(x,y))

Finally, we must specify the semantics for the aggregation. If object c is an aggregation of objects x1,x2 .. xn, then it is automatically deleted when one of the objects xi is deleted. We interpret aggregation attributes in the same way as ordinary attributes (the set AT in L- Dyn), but we record the fact that the attribute is an aggregation attribute with the predicate Aggr(,). For example, Aggr(document entry, document) specifies that "document" is an aggregation attribute of object type "document entry". The semantics is expressed in A26:

A26 FORALL x,y,e1,a: (TYPE(x,e1) AND VALUE(x,a,y) AND Aggr(e1,a))
=> (DELETE'(y) >> DELETE'(x))

That is, the effect of DELETE'(y) is not only that y no longer exists, but aggregate object x exists no longer as well.

4. CONCLUSION

In this paper, we have introduced a specification language for both the static and the dynamic part of a conceptual model. The language has been used in an object-oriented design method. We described the main semantics in the form of dynamic logic, which has been used before successfully for the semantics of programming languages and for the specification of integrity constraints in conceptual models. In comparison with the MINOS system (Christodoulakis et al, 1986), which is also object-oriented, our system has a more refined versioning mechanism (we distinguish historical versions from co-occurring variants), and it facilitates object-oriented specification of operations. Combined with the three-level architecture of the system, this may make our system more extensible.

Some integrity constraints can be expressed in first-order logic as well as in dynamic logic. An example is the dependency relationship between composite objects and components. The advantage of the dynamic logic formulation is that it gives a better clue for the interpretation. A difference can be made between forbidding some action that would cause an inconsistent state and allowing it, then "repairing" the inconsistency immediately. The dynamic logic formulation can be mapped easily on an implementation; for the static integrity constraint, additional choices would have to be made during the mapping.

The variant mechanism has not been considered in this paper. We intend to work out the semantics of these separately.

In the short future, we plan to implement a Data Dictionary facility in which the OO specifications can be stored and which supports the interpretation of GQL queries. It will be integrated with the RIDL tools so that database tabe, simple constrants and atomic procedures can be generated automatically. In the remote future, we plan to work on the compilation of the method specifications. At that stage, we also have to consider what restrictions must be set on the formats of the postconditions in order to guarantee automatic compilation. If the postcondition is too complex, the designer must specify an implementation method for it.

5. Acknowledgements

Research for this report was sponsored by the European Community under ESPRIT project 2001 (SPRITE). The other participants of this project are: Océ-Nederland (NL), AEG Electrocom (FRG), Alcatel TITN (F) and Trinity College Dublin (IR). I am grateful to Liu Ling and Egon Verharen for useful comments on a first version of this paper.

References

Banerjee, J. et al, 1987. Data Model issues for Object-Oriented Applications. *ACM Trans on Office Information Systems*, 5(1).

Carey, M.J., DeWitt, D.J. and Vandenberg, S.L., 1988. A data model and query language for EXODUS. *Proc. ADM SIGMOD*, pp.413-423.

Christodoulakis et al, 1986. Multimedia Document Presentation, Information Extraction, and Document Formation in MINOS: A Model and a System. *ACM Trans on Office Information Systems*, 4(4).

DeBakker, J.W., 1980. *Mathematical Theory of Program Correctness*, Prentice-Hall, London.

DeTroyer, O., 1989. RIDL*: A Tool for the Computer-assisted engineering of large databases in the presence of integrity constraints. *Proc ACM SIGMOD.*

Dignum, F., 1989. A Language for Modelling Knowledge Bases based on Linguistics and Founded in Logic. Ph.D. Thesis, VU Amsterdam.

Harel, D., 1984. Dynamic Logic. In: D.M. Gabbay & F. Guenther (eds), *Handbook of Philosophical Logic, Vol 2*, Reidel, Dordrecht.

Khoshafian. S & G. Copeland, 1986. Object identity. *PROC ACM OOPSLA* 1, Portland.

Khosla, S. et al, 1986. Database Specification. In: T.B. Steel & R. Meersman (eds), *Database Semantics (DS-1)*, North-Holland, pp141-158.

Kent, W., 1989. An overview of the versioning problem. Panel Contribution, *Proc. ACM SIGMOD.*

Kim, W. et al, 1989. Composite Objects Revisited. *Proc. ACM SIGMOD.*

Meyer, J.-J.Ch, 1988. A Different Approach to Deontic Logic: Deontic Logic Viewed as a Variant of Dynamic Logic. *Notre Dame Journal of Formal Logic, 19(1).*

Nijssen, G.M., 1976. A Gross Architecture for the next generation database management systems. In: G.M.Nijssen (ed), *Modelling in database management systems*, Proc. IFIP TC-2, North-Holland.

Weigand, H., 1990. *Linguistically motivated principles of knowledge base systems.* Foris, Dordrecht.

Wieringa, R.J., J.-J.Ch. Meyer, H. Weigand, 1989. Specifying Dynamic and Deontic Integrity Constraints. *Data and Knowledge Engineering, 4(3)*, pp.157-189.

Wieringa, R.J., H. Weigand, J.-J.Ch. Meyer, F. Dignum, 1989. The inheritance of Dynamic and Deontic Integrity Constraints. *Annals of Mathematics and Artificial Intelligence*, Oct 1990.

ITK: P.O. BOX 90153 5000 LE TILBURG THE NETHERLANDS