Supporting Office Document Architectures with Constrained Types

W B Croft
D W Stemple
Department of Computer and Information Science
University of Massachusetts, Amherst, MA 01003

Abstract

Data models have been proposed as a means of defining the objects and operations in an office information system Office documents, because of their highly variable structure and multimedia content, are a difficult class of objects to model The modeling task is further complicated by document architecture standards used for interchange between systems We present an approach to data modeling based on constrained type definitions that allows architecture standards to be defined and ensures that individual document types conform to those standards The ADABTPL model, which is used to define the schema of document types and standards, is described

1 Introduction

Office information systems, because they attempt to integrate all aspects of a business environment, must be able to deal with a large variety of information types Data that has traditionally been stored in databases such as the typical "supplier and parts" example, must be combined with office objects such as forms, documents and mail For example, an order form may contain references to suppliers and parts that have their own descriptions A number of data models have been proposed for this application [1,2,3], and the importance in the office of documents with complex structure and multimedia content means that these approaches have strong similarities to other research in multimedia databases [4], which includes applications such as CAD

As well as providing a common language for defining the structure of objects, a data model can be used to define the valid operations on objects such as documents This "object-oriented" view of data modeling has been shown to be useful in multimedia applications [4], and can be extended to describe the user operations or tasks that make up office work [5]

Document architecture standards, such as ODA [6], are also designed to be able to represent the possible structures of office documents The main aim of these architectures is to allow the interchange of documents between open systems One of the results of this emphasis is that ODA provides for the description of the *layout* structure of a document as well as the *logical* structure ODA also specifies constraints on allowable document types, rather than being used only for the specification of types Parts of ODA, therefore, have similarities to the "metaclasses" of object-oriented programming languages [7]

If a data model is used as the basis for an office information system, it must be capable of representing document architecture standards and enforcing them on document type definitions In this paper, we describe a data model, ADABTPL, that is characterized by formal underpinnings, type constructors, robust subtyping, and extensive constraint mechanisms We argue that a model with these characteristics is essential to represent document architecture standards and their relationship to particular document types That is, in addition to basic facilities for defining aggregation, generalization and instantiation, a data model should have a formal means for defining constraints on types and specifying how these constraints are inherited and specialized by subtypes The ADABTPL constraint specification language and manipulation language has a formal semantics that supports mechanical reasoning about important properties of specified systems [8,9] Specifying document architecture standards in ADABTPL gives them a formal semantics and integrates them with the rest of the office model

In the next section, we describe the ADABTPL model in more detail and show how it can be used to model simple office documents In section 3, a simplified version of the ODA standard is discussed as an example of a document architecture standard The specification of this standard in ADABTPL and its relationship to specific document types is presented in section 4 Section 5 gives an example of the operations provided in ADABTPL

2 Modeling Office Documents

The ADABTPL data model (pronounced "adaptable") forms part of a system that has been used to verify the safety of transactions for highly constrained databases [8,9] Types are constructed in ADABTPL from the primitive abstract data types tuples, lists, and finite sets, each of which has a formal semantics The type constructors available in ADABTPL include enumeration, indexing (arrays), conjunctive aggregation by use of the tuple constructor "[]", derivation of a new type from an old type by using a predicate within a *where* clause, refinement or enrichment of a new type from an old type using the *with* clause, disjunctive aggregation (generalization) using a discriminated union of types, and an encapsulation construct for defining abstract data types

[10] The operations in ADABTPL are specified at three levels the primitive abstract data types such as SET have associated operations, operations can be encapsulated with user-defined abstract data types, and database transactions are operations on the database type The ADABTPL model has similarities to both IFO [11] and GALILEO [12]

In ADABTPL, the database schema is seen as a set of type definitions culminating in a type definition of the database object itself Constraints on the database are specified in *where* clauses in both the constituent type definitions and the database type definition

The following ADABTPL example defines a simple document type, each instance of which is a tuple having at least one author, and a title and body, each of which could be empty

```
simpledocument-type =
     [title  text,
      author  list of string,
      body  text]
   where notempty(author),
```

If the title and author combined forms a key for a relation containing simple documents, this is expressed as

```
simpledocs-rel =
  set of simpledocument-type
  where key(simpledocs-rel  title, author),
```

The basic ADABTPL view of a type is that it is the set of all possible members of the type and is defined by the necessary properties members must have to belong to the type The two relationships in this view of types are *type membership* or *instance of*, the relationship of an object to a type, and the *subtype/supertype* relationship ADABTPL supports the subtyping concepts of both traditional database systems and object-oriented languages That is, one view is that A is a subtype of B iff A is a subset of B The alternative view is that if all functions defined on objects of type A correctly operate on all objects of type B, then B is a subtype of A

There are two main ways of defining subtypes in ADABTPL – *derivation* and *enrichment* (other mechanisms for defining subtypes will be described later) Derivation refers to forming a subtype by adding a constraint to a type The resulting type is the subtype consisting of all instances of the first type that obey the constraint Enrichment refers to forming a new tuple type by adding a new component (or components) to an existing tuple type The resulting type is a subtype in the sense that its instances have all the properties (components) of the first type, but also have a new property (or properties) represented by the new component(s) For example, a letter document that is a subtype of the simple document with the additional constraint that there is only one author is defined as

```
letter-type = simpledocument-type
  where length(author) = 1,
```

A report document that is a subtype of simple document with an additional component that contains the date is defined as

```
report-type = simpledocument-type
  with [date  string],
```

As in the case of simpledocs-rel type above, the sets or relations of instances of object types are declared in ADABTPL as separate types

```
letter-rel = set of letter-type,
report-rel = set of report-type,
```

This allows flexibility in terms of the relationships between instances of the types and subtypes For example, if a database consists of a letter-rel set and a report-rel set, then these sets could be defined to be non-overlapping on their keys as follows

```
database-type =
  [simpledocs  simpledocs-rel,
   letters  letter-rel]
  where
  no_overlap(project(simpledocs,title,author),
             project(letters,title,author))
```

ADABTPL also provides a discriminated union type constructor As an example of its use, consider a document that consists of a title followed by a list of sections where each section may be a paragraph, figure or table The following declarations define the type of such a document

```
section-type = union [ P -> paragraph-type,
                       F -> figure-type,
                       T -> table-type ]
complexdocument-type =
  [title  text
   sections  list of section-type],
```

The symbols P, F and T are the discriminators of the type and can be used in a discriminated case statement Enrichment and union implement different forms of the IS-A subtyping construct from the knowledge representation literature

Simple models of document structure give a misleading picture of the ease of document specification Rabitti [13] has described how document structures that can readily be described in ODA are difficult to represent using the standard type definition facilities found in data models The main problems addressed are the representation of the variations that occur within instances of a document type, the revision of document instances to produce different structures, and the refinement of type descriptions to produce new type descriptions An example of the first problem is that two summary reports may have different numbers of sections and different occurrences of figures, tables and subsections within those sections This is to be contrasted with the fixed structure of an employee record from a typical database application The second problem is illustrated by a summary report with three sections containing text being edited and changed into a summary report with five sections containing tables and figures in addition to text An example of the third problem is that the summary report is a document type related to the general report type The general report type may contain only a header and a body, whereas the summary report type will define the structure in more detail by specifying, for example, that the body consists of four sections

These problems must be addressed by any data model proposed for multimedia or office applications (see, for example, [4]) In ADABTPL, the first two problems mentioned above can be handled using type definitions similar to those given in previous examples The third problem is addressed by the following (incomplete) definitions

```
general-report-type(body-type) =
  [report-id  string
   header  header-type,
   body  body-type],
```

```
summary-body-type =
  [sections  list of section-type]
  where length(sections) = 4,

summary-report-type =
  general-report-type(summary-body-type),
```

In this example, general-report-type is a *parameterized type* That is, the type is defined using a type variable in place of a type name The summary-report-type instantiates the value of this type variable and thereby defines the specific form of body-type that is required It will be shown that the flexibility of subtype definition in ADABTPL is an essential feature for representing document architecture standards

3 The ODA Document Architecture

The main components of the ODA standard are described here as an example of a document architecture ODA is part of the standards for document interchange being developed by ISO (International Standardization Organization) and ECMA (European Computer Manufacturers Association) In the following discussion, we shall concentrate on those aspects of ODA that have the most impact from the data modeling perspective It should be kept in mind that the terminology of ODA, particularly with regard to types, is not the same as ADABTPL and is somewhat confused A more detailed description of ODA appears in [6]

As mentioned previously, ODA distinguishes between the logical and layout structure of a document The logical structure is made up of a hierarchy of *logical objects* Similarly, the layout structure is a hierarchy of *layout objects* The content of the document (text, image, etc) is associated with both the logical and layout structures Logical and layout objects are classified according to their "type" The logical object types are *composite* or *basic* Layout object types include *page set*, *composite page*, *basic page*, *frame* and *block* *Basic objects* (basic logical objects, basic pages or blocks) are associated with *content portions*, which contain the content of the document A document object consists of one or more composite logical objects or basic logical objects The document may contain only composite logical objects, or only basic logical objects, or both The document must contain at least one composite logical object or basic logical object Composite logical objects consist of other composite logical objects or basic logical objects Figure 1 gives a simplified form of the hierarchical logical structure of an ODA document It also shows the layout structure relationships It should be emphasized that these standards are constraints on the allowable structure of specific document types To enforce these constraints, a data model has to be able to represent the document type as some form of subtype of the standard

Objects with similar properties are defined in ODA as *object classes* Examples of basic logical object classes are Paragraph and Date Examples of the layout block class are ParagraphBlock and DateBlock Figure 2 shows part of the type hierarchy of objects specified in ODA with the corresponding ODA names The links in this figure represent subtype relationships

Documents with similar characteristics are defined in ODA as *document classes* In data modeling terms, these are the document types Examples of document classes are Report, SalesReport and BusinessLetter ODA does not explicitly allow document classes to be related through specialization

ODA also contains *layout directives* that are used to create layout structures and associate content portions with basic layout objects After laying out an instance of a document class, the ODA logical and layout structures are related through common content portions An example of how this may look for a document of type SalesMemo is shown in Figure 3

4 Enforcing Standards using ADABTPL

The schema and transaction definition facilities provided in ADABTPL can be used to describe a document architecture standard, relate particular document types to that standard, and create instances of the documents Document types are created as subtypes of an ODA type We shall demonstrate this with the simplified form of ODA and the Sales Memo document type described in the last section The first part of the schema describes the main components of any ODA type

```
oda-type =
  [logical  composite-logical-type
   layout  composite-layout-type,
   mapping  map-rel]
  where
  project(all x in logical
            where basic-logical-type-p(x)),
          b-log-id)
  = project(mapping  b-log-id) and
  project(all y in layout
            where basic-layout-type-p(y)),
          b-lay-id)
  = project(mapping, b-lay-id),

{This constraint states that the mapping
 maps all the basic logical objects to all
 the basic layout objects  The key
 constraint on the map-rel type (below) is
 inherited to constrain this mapping to be
 many-to-one }

map-type =
  [b-log-id  number,  b-lay-id  number],

map-rel = set of map-type
  where key(map-rel,b-log-id),
```

This defines an oda-type as consisting of a logical part, a layout part, and a mapping In our view of the simple ODA architecture, the content of the document will be stored in the basic logical objects defined in the logical component of the oda-type, not in separate "content portions" The mapping relation records how basic logical objects are mapped to basic layout objects when the layout process is done This is done using unique identifiers for each basic object Each logical object (and its content) is assumed to be mapped to one basic layout object A basic layout object can contain a number of basic logical objects This restriction (purely for puposes of illustration) is captured in the key constraint in map-rel The constraints in the oda-type definition ensure that the mapping is total (an instance of referential integrity) The predicates `basic-logical-type-p` and `basic-layout-type-p` test if a component is of the specified type These predicates can be defined for union types using the discriminators in a simple case statement

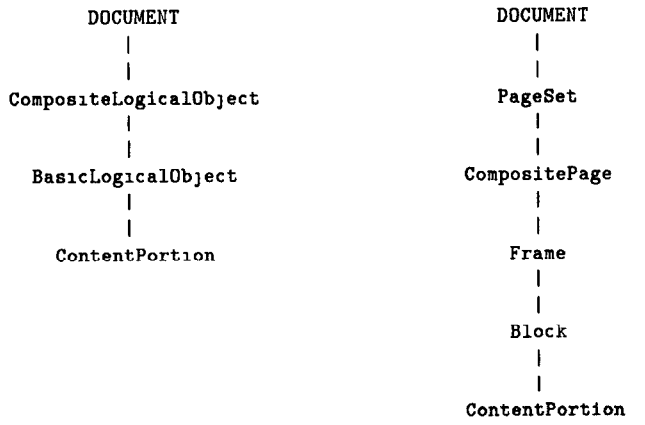

Figure 1 Simplified ODA Document Structure

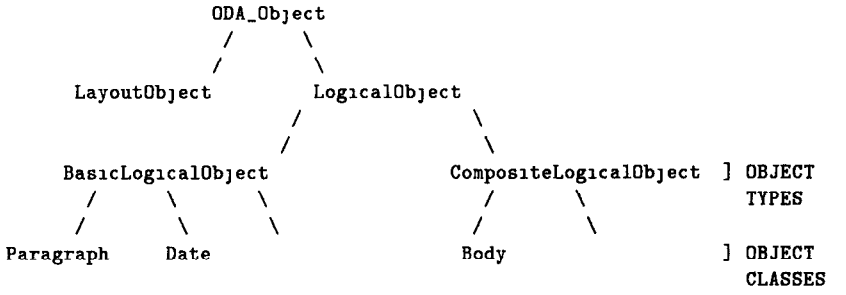

Figure 2 A type hierarchy of ODA objects

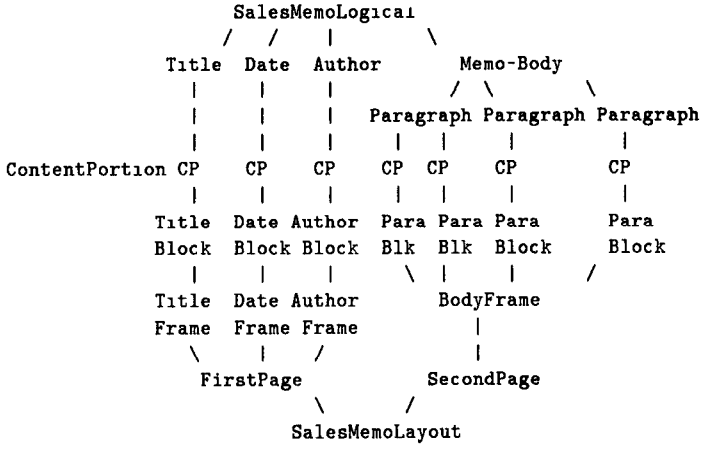

Figure 3 An instance of an ODA document class SalesMemo

The next part of the schema defines the logical part of an ODA document in more detail and defines the logical part of the Sales Memo document In general, types such as composite-logical-type in the ODA architecture will have a number of other components that are not shown here We have included one such attribute (example-feature) to show how these components would be handled

```
composite-logical-type =
  GENPLEX
   [basic-logical-type ->  basic_type
                with [b-log-id  number],
    comp-log-type ->
      [example-feature  example_type,
       components
         list of composite-logical-type]
      where notempty(components) ],


basic-type =  union [ P -> paragraph-type,
                      D -> date-type,
                      A -> author-type,
                      T -> title-type ],

paragraph-type = text, {for example}
```

In this definition, composite-logical-type is a recursive type Theoretically there is no barrier to the definition and use of recursive type definitions in ADABTPL For practical reasons they are limited to those that can be defined as a combination of recursive tupling and discriminated union (GENPLEX) The discriminators of a genplex type become globally known types The basic logical type has an identifier and the content of the document is contained in objects of type basic-type Only one of the basic types is defined as an example

```
sales-memo-body = comp-log-type
  where
    for all a in sales-memo-body components
        paragraph-type-p(a)

sales-memo-logical =
  comp-log-type where
  length(sales-memo-logical components)
        = 4 and
  title-type-p(first
    (sales-memo-logical components)) and
  date-type-p(second
    (sales-memo-logical components)) and
  author-type-p(third
    (sales-memo-logical components)) and
  sales-memo-body-p(fourth
    (sales-memo-logical components)),
```

The logical components of the Sales Memo document type are defined as subtypes of the ODA standard types using constraint predicates The elements of the composite-logical-type list are referenced using predefined functions first, second, etc , that are defined in terms of CAR and CDR, the basic list operations The naming of the components of sales-memo-logical could be made more convenient by defining functions such as title() and date() that would be identical to first() and second() The abstract data type facilities in ADABTPL provide a robust ability to define and encapsulate operations on types, but are beyond the scope of this paper [10]

The layout architecture is defined in a similar way The main difference is that the types defined in the architecture will, in general be the same as those used in specific document layout types For example, objects of type frame will serve as title frames, date frames, author frames, and body frames for the Sales Memo layout Once again, we emphasise that these definitions leave out many components of the layout objects that would be necessary in an operational system

```
composite-layout-type =
  GENPLEX
  [comp-lay-type ->
     parts  list of composite-layout-type
            where notempty(parts),
   comp-page-or-frame-type ->
     GENPLEX
     [comp-pg-fr-type ->
        [ex-feature-2  ex-type-2,
         rest list of comp-page-or-frame-type]
        where notempty(rest)
      bl-type -> block-type ]]

{Example components are used to illustrate that
 a full specification of these types would
 contain many more components}

basic-layout-type =
  [b-lay-id  number, ex-feature-3  ex-type-3]

block-type = basic-layout-type
  with [block-component  component-type],

{block-type is enriched subtype of
 basic-layout-type with extra component,
 basic-page (not described) is also a
 subtype of basic-layout-type}

sales-memo-layout =  comp-pg-fr-type
```

This specifies that the sales memo layout simply consists of a list of composite-page or frame types The next part of the schema defines the sales memo document type as a subtype of the ODA type

```
sales-memo-type = oda-type
  where
    sales-memo-logical-p(logical) and
    sales-memo-layout-p(layout),

{also define relation of sales memo objects}

sales-memo-rel = set of sales-memo-type
```

5 Creating documents

Instances of sales memo documents are created with the insert, update and delete operations provided in ADABTPL transactions The first part of creating an instance of any ODA document is to insert the logical (content) part The layout structure and mapping components are created when the document is "layed out" into pages If we assume that the example-feature

of a composite logical type is an identifier, then a typical insertion may be as follows,

```
insert [ sysid(),
         (['title',sysid()]
          [ 11/4/87 ,sysid()],
          ['Jones',sysid()],
          [sysid(),
           ([ First line of memo ',sysid()]
            nil)]) ]
into sales-memos,
```

The function sysid() assigns a unique identifier to the composite and basic logical objects We have assumed that the database type contains a component `sales-memos` of type `sales-memo-rel` If the memo is later expanded to include a second paragraph, this could be done using the following UPDATE operation

```
update s in sales-memos
  where s logical example-feature = doc-id
  by [second(s logical body components)  =
        ['The second memo entry ',sysid()],
      third(s logical body components)  = nil],
```

These operations are used to define transactions The transactions can then be checked for safety by the associated reasoning system For example, if author had been defined as unique for each sales memo, the following transaction would be rejected by the system at compile-time

```
transaction change-author
  (doc-id  number, new-author  author-type)
begin
  update s in sales-memos
    where s logical example-feature = doc-id
    by [ s logical author  = new-author ]
end,
```

The transaction is rejected because the system cannot prove that it will not violate the uniqueness constraint for author In this case, the ADABTPL system will recommend that a precondition

```
new-author not-in sales-memos logical author
```

be added to the transaction The automatic verification of more complex transactions is described in [9]

6 Conclusion

ADABTPL is an example of a data model that can be used to specify document architecture standards and enforce these standards on specific document types Document types are seen to be subtypes of types defined in the standard Capturing document architectures in the same language as is used to describe other parts of the office information system has obvious advantages in terms of integration and sharing of data, but it requires robust structuring and subtyping capabilities To represent ODA, we used subtypes defined by derivation, enrichment, disjunctive union, and the GENPLEX constructor The constraint specification language must be powerful, and to ensure both standardization and correct implementation, should have totally formal semantics Ideally, this will enable robust mechanical reasoning about important properties of specified systems The ADABTPL model has these characteristics and a transaction verification system using ADABTPL has been implemented in FranzLisp We are currently working on an implementation of a complete ADABTPL system

Acknowledgments

This research was supported by a contract with Ing C Olivetti & C and by NSF grants DCR-8503613 and IST-8606424 Additional support was provided by University College Dublin, where the first author was on leave David Harper, of Glasgow University, suggested using data models for this application Discussions with T Bogh, B Anker-Moeller and other members of ESPRIT project 59, and with Tim Sheard of the University of Massachusetts, Amherst were particularly valuable

References

[1] S J Gibbs Conceptual modeling and office information systems, in *Office Automation*, ed D Tsichritzis, Springer-Verlag, Berlin, 193-225 (1985)

[2] W Lamersdorf, G Muller and J W Schmidt Language support for office modelling *Proceedings of VLDB 10*, Singapore, (1984)

[3] D J Harper, J Dunnion, M Sherwood-Smith and C J Van Rijsbergen Minstrel-ODM A basic office data model *Information Processing and Management 22*, 83-107 (1986)

[4] D Woelk, W Kim and W Luther An object-oriented approach to multimedia databases *Proceedings of SIGMOD '86*, 311-325, (1986)

[5] W B Croft Task management for an intelligent interface *IEEE Bulletin on Database Engineering*, Vol 8, 8-13, (December 1985)

[6] W Horak Office document architecture and office document interchange formats - Current status of international standardization *IEEE Computer 18*, 50-60 (October 1985)

[7] M Stefic and D G Bobrow Object-oriented programming Themes and variations *AI Magazine 6*, 40-62 (1986)

[8] T Sheard, D Stemple Coping with Complexity in Automated Reasoning about Database Systems *Proceedings of VLDB 11*, 426-435, (1985)

[9] T Sheard, D Stemple Automatic Verification of Database Transaction Safety Technical Report 86-30, Computer and Information Science Department, University of Massachusetts, Amherst, (1986)

[10] T Sheard, D Stemple Type Specification in ADABTPL Technical Report, Computer and Information Science Department, University of Massachusetts, Amherst, (1987)

[11] S Abiteboul, R Hull IFO A Formal Semantic Database Model *Proceedings of the Third ACM Symposium on Principles of Database Systems*, 119-132, (1984)

[12] A Albano, L Cardelli, R Orsini Galileo A Strongly-Typed, Interactive Conceptual Language *ACM Transactions on Database Systems, 10* 230-260, (1985)

[13] F Rabitti A model for multimedia documents, in *Office Automation*, ed D Tsichritzis, Springer-Verlag, Berlin, 227-250 (1985)