Artificial Intelligence Applications Institute

The Use of Object Oriented Database in PLINTH

Andrew Casson

AIAI-TR-145

March 1994

Artificial Intelligence Applications Institute
The University of Edinburgh
80 South Bridge
Edinburgh EH1 1HN
Scotland

Copyright © 1994 by Artificial Intelligence Applications Institute, The University of Edinburgh

The Use of Object-Oriented Databases in PLINTH

Andrew Casson
Artificial Intelligence Applications Institute
University of Edinburgh

March 1994

Abstract

This paper addresses the data storage and access problems in the current implementation of the expertext shell PLINTH. The nature of these problems and of the data that PLINTH manipulates suggest object oriented database (OODB) technology as a possible solution. OODB features are described and their appropriateness for use in PLINTH examined. Finally, we discuss the changes that need to be made to the existing system in order to accommodate an OODB storage medium.

1. Introduction

PLINTH [1,2] is a domain independent expertext shell which provides rule-based guided creation and navigation of semantically enriched hypertext networks. Its principal objective is to ensure effective and efficient authoring and use of documents, taking into account the particular needs of different classes of author and reader and the different tasks that they use the documents to support. The techniques used avoid `deep analysis' of the meaning of texts, concentrating instead on identifying and making use of their physical, logical and rhetorical *structure*.

PLINTH not only represents the contents of draft and finished documents as hypertext, but can also be used to manipulate both the *design rationale* [3] underlying the documents, and classification/indexing schemes for accessing them. As a result, PLINTH applications may generate and manipulate very large amounts of data.

All of this data is handled internally using a single underlying constrained node-and-link network representation system, called CONNEKT [2]. At present, simple file-based representations are used for the external storage both of CONNEKT nets and of the text of nodes within those nets. Nets can be loaded from files into main memory and written back again individually, even though CONNEKT allows `external' links between nodes in different nets. So for an application which uses many small nets as opposed to fewer large ones, limitations on main memory resources will not generally be a problem. However, it may not always be convenient or desirable to organise nets in this way and some of them may need to be very large indeed. Because of this, we see a need for an efficient, persistant, external data representation scheme for PLINTH, whereby nets do not have to be loaded in their entirety to be used.

Another important feature of PLINTH applications is that multiple users may need to access and update a single corpus of data concurrently. In this situation, the system should allow each user see significant changes made by others immediately, and should safeguard the data from two users trying to change a single item simultaneously or making inconsistent changes to mutually dependent items. Unfortunately, due to the limitations of its file-based external representation, PLINTH does not currently provide such facilities.

This paper proposes the use of object-oriented database technology as a solution to these problems of data volume and access control.

2. What are Object Oriented Databases?

Object-oriented databases (OODBs) represent the latest generation of database systems technology. Research and development of OODBs has been driven largely by the highperformance data storage and access requirements of design support environments, such as Computer Aided Design (CAD) and Computer-Aided Software Engineering (CASE) systems.

Atkinson et al. [4] and Zdonik and Maier [5] propose definitions of *object oriented database systems* in terms of their mandatory features, which we can divide into *object-oriented* features and *database* features, as summarised below.

2.1 Object-oriented features

These features are essentially those of object-oriented programming languages such as C++ and SmallTalk, but they are not generally found in relational database management systems (RDBMS). The three most important of these are:

complex objects:
Complex objects can be built up from simpler ones by applying *set*, *tuple* and *list* constructors which represent collections, attributes and relations, and order.

object identity:
An object in the database should have an identity independent of its value, so that there is a useful equality distinction between two identical objects which point to exactly the same data structure in memory and two objects which are merely similar in that their types and elements are equal. This provides for *object sharing*, i.e. objects can share a single element and updates to that element will be reflected in each object automatically.

encapsulation:
An object has a visible external *interface* that is independent of its hidden internal *implementation.* Only those operations specified in the interface can be performed on the object by applications. The implementation part includes both attributes representing the state of the object, and procedures which describe each operation.

Less central to the OODB concept according to Zdonik and Maier, but still seen as mandatory by Atkinson et al. are the following features:

types or classes:
Object *types* (found in C++) and *classes* (found in SmallTalk) both summarise the common features of a set of objects with the same characteristics, but they are employed in different ways. Types are not objects themselves, and are used primarily to ensure program correctness at compile time by allowing the user to declare the representation of variables and expressions the program manipulates. Classes on the other hand *are* objects, and are used at run-time to create new objects (instances), for example by applying a **new** operator to the class.

Note that the *type/class* distinction corresponds to the difference between *static* and *dynamic* data typing in procedure-oriented programming languages such as C (static) and Lisp (dynamic), but these do not have the added benefit of encapsulation.

type or class hierarchies and inheritance:
Inheritance provides a means of avoiding the explicit and repeated storage of data which can be inferred, and helps code reusability. When two types *t1* and *t2* have features (attributes and operations) in common, those features can be abstracted out into a single subsuming type *T*, with *t1* and *t2* being declared as subtypes of *T* so that they automatically gain the common features without each having to respecify them.

overriding, overloading and late binding:
It may be desirable for a type *t* to inherit certain attributes or operations of type *T*, but not others which are under-specified or otherwise inappropriate. In this case *t* can *override* the features by defining its own local features with same name.

Having two or more types use the same name for related but distinct attributes or operations is called *overloading* the name. The system can select the appropriate binding for the name at run-time by examining type of the object to which it is being applied (this is called *late binding*).

extensibility:
Applications can add to the set of types or classes built into the OODB, and new types or classes must have the same status and usage as predefined ones.

computational completeness:
Either the database description and manipulation language should itself be able to express any computable function (not the case for the RDBMS query language SQL, for example), or it should provide an application programmer's interface (API) to a standard programming language.

2.2 Database features

These are the essential features of databases in general, including RDBMS, which are not generally found in programming languages, whether object-oriented or not.

persistance:
The data manipulated by an application, whatever its type, should survive beyond the execution of the process which runs it and be available for re-use in subsequent processes. This should happen implicitly—*save* and *load* operations etc. should not be required.

efficient secondary storage management:
Mechanisms should be provided for fast and efficient manipulation of very large databases. Allocation of disk storage and transfer of data to and from main memory should be invisible to the application.

concurrency:
Multiple users should be able to work concurrently on a database, viewing and updating shared data without affecting its integrity. The system should *lock* objects when they are read or written to prevent simultaneous access, and execute *atomic* operation sequences which cannot be interrupted leaving the database partially updated.

recovery:
In the event of disk, processor or software failure, the OODB should be able to restore the data to a coherent state.

ad hoc query language:
The system should provide the *functionality* of an efficient, high-level, application-independent, declarative *ad hoc* query language. This does not necessarily mean it should provide a textual query language *per se*; a graphic browser with equivalent access facilities is just as good. The internal data manipulation language could also be made available for this job.

3. Using OODBs in PLINTH

It is clear from the features described in section 2.2 that database technology is a good candidate for meeting PLINTH's requirements for efficient, persistent external data storage and concurrency control. What must now be established is whether a specifically *object-oriented* database system is needed. For this it is necessary to consider how PLINTH currently represents its data internally.

PLINTH is written entirely in Pop11, a procedure-oriented programming language comparable in power and basic functionality to Common Lisp, but syntactically more akin to Pascal or C. As mentioned in the introduction, the lowest layer in PLINTH is a library called CONNEKT. So when we talk of PLINTH's data storage and retrieval we are really referring to CONNEKT[1].

CONNEKT represents nets, nodes, links and their attribute slots as record classes (cf. *defstruct* in Lisp, or *struct* in C). Figure 1 shows simplified versions of the declarations

[1] Note that where there is text associated with a node, this is just represented as an ordinary attribute of the node, along with all its other application attributes. There is nothing special about text nodes as far as CONNEKT is concerned, and how the text fragment itself is actually represented is not important here. However, the issue of text format and representation is very important in other respects to PLINTH, and is discussed in detail elsewhere [6].

of these records in Pop11, with comments showing the type and contents of each field. It should be noted (following the distinction described in section 2.1) that Pop11 records define dynamic **classes** of which instances are constructed at run-time, rather than static **types** used for compile time checking.

```
defclass net {
        net_file,       ;;; string: name of external storage file
        net_name,       ;;; string: name of net
        net_class,      ;;; word (= LISP atom): class of net
        net_nodes,      ;;; list of node records in the net
        net_slots       ;;; list of slot records representing attributes
};

defclass node {
        n_net,          ;;; net record containing node
        n_class,        ;;; word: class of node
        n_name,         ;;; string: name of node
        n_linksfrom,    ;;; list of link records connecting from node
        n_linksto,      ;;; list of link records connecting to node
        n_slots         ;;; list of slot records
};

defclass link {
        l_class,        ;;; word: class of link
        l_fromnode,     ;;; node that this link connects from
        l_tonode,       ;;; node that this link connects to
        l_slots         ;;; list of slot records
};

defclass slot {
        slot_name,      ;;; word: name of attribute
        slot_value      ;;; any: value of attribute
};
```

Figure 1: CONNEKT type declarations for nets, nodes, links and slots

We can see at once from the record class definitions that CONNEKT data items are indeed **complex objects**: instances of these records could be regarded as ordered lists of name/value tuples (although their actual representation is of course more efficient than that).
There is also *object sharing*. For example, each *node* structure can be referred to both by its *net*'s list of nodes, and by each *link* connecting to or from it. This is straightforward in Pop11 since all non-numeric data items are implicitly treated as *pointers* to structures (in C terms) and therefore they have an **object identity** distinct from their value.

CONNEKT includes **extensible** node and link **class hierarchies** and **inheritance**, but because Pop11 records do not at present support subclassing directly, the node or link class has to be represented as ordinary field in the record, and the CONNEKT system has to represent the hierarchies and manage inheritance itself. However, a new *objectclass* facility is about to be introduced into Pop11, which extends *defclass* to provide subclassing, inheritance with **overriding**, **overloading**, and **late binding**, and

also data **encapsulation**. PLINTH will very probably make internal use of these *objectclass* facilities, so its external data storage should ideally provide them too.

In summary then, although PLINTH is not written in a conventional object-oriented language, it most certainly is an object-oriented application, which makes use of complex objects, object identity, extensible class hierarchies and inheritance, and which would benefit from the other objectoriented features described in section 2.1. Object-oriented database technology would therefore seem to provide the ideal answer to PLINTH's external data storage and access requirements. On reflection, given that OODBs were originally developed for use in design support environments, of which computer assisted authoring tools are another good example, along with CAD and CASE, this should come as no great surprise.

4. OODB features required by PLINTH

In this section, we will discuss the specific features of OODB systems that are needed by PLINTH, and those that are not, in order to have a basis on which to select a suitable candidate (see the Appendix). Not all available systems calling themselves OODBs have all the features described above, and some provide many more. Conversely, some of the features are not actually essential for PLINTH, though they would all be useful if available.

4.1 Essential features

4.1.1 Object-orientation

Of the `mandatory' object-oriented features listed in section 2, only **complex objects**, **object identity**, and **computational completeness** (in the sense that an API is provided) are absolutely essential for PLINTH, since these are the only features provided directly by Pop11 in the current implementation, as we saw in section 3. **Extensible class hierarchies** with **inheritance** and **overriding** can be simulated by the application using ordinary object attributes to represent classes, and **encapsulation** can be provided by procedural wrappers if required. The API provided by the OODB must use the C language, as we shall explain.

Although an OODB should be computationally complete, this is not the same as being *resource complete*, i.e. having full access to the machine's input and output devices. The OODB may not therefore be able to express a complete application. However, this is generally quite acceptable, since a welldesigned system should keep its user-interface and data storage modules distinct, allowing interaction only through a tightly specified interface (see section 5 below).

PLINTH's own graphical user interface (GUI) is very complex, and is written in Pop11 using its X Toolkit libraries. This need not change when CONNEKT is moved onto an OODB platform if a Pop11 API to the OODB can be provided. No existing OODB provides a Pop11 API directly, but Pop11's facility for externally loading C

library functions can be used to write one, since many OODBs provide a C/API (Pop11 does not, unfortunately, provide a C++ interface, which is even more common in OODBs).

Choosing a C/API makes sense for another reason. Although Pop11 provides an excellent environment for prototyping and development, it does not run on a wide range of platforms, and it is expected that PLINTH will be eventually have be ported to a more widely used language. Since the X Toolkit is also written in C, this language would seem to be a good candidate. Moreover, Pop11's C interface means that small PLINTH modules can be converted into C independently, so that we can gradually migrate the system over time from Pop11 to C, rather than having to start from scratch.

4.1.2 Database

Of the database features listed all but the **ad hoc query language** (whether textual or graphical) are necessary. As we have seen, PLINTH will take care of user-interface issues itself, accessing the data through the OODB's API, and providing display and manipulation facilities using the same window-based graphical toolkit as it currently uses.

4.1.3 General

PLINTH currently runs on Sun SPARCStations, and a suitable OODB would have to do the same. However, in order to maximise the scope for porting PLINTH to new platforms in future, the more machines and operating systems the OODB runs on now, the better.

4.2 Non-essential but desirable features

4.2.1 Schema evolution

`Schema evolution' refers to the ability to modify the class hierarchy for an existing database, and to update at runtime the classes of the objects that have already been created. Many databases do not provide such facilities; the class hierarchy is defined and fixed before the database is first opened, and the class of each object when it is created, and these may not change subsequently. In PLINTH, however, the user is allowed to change the class hierarchy for a net (defined in an NCM—see [2]) and the classes of individual nodes and links at any time, provided that the changes do not lead to inconsistencies. Therefore, for PLINTH to make use of an OODB's class system, schema evolution must be supported.

4.2.2 Versioning mechanism

A versioning mechanism records the history of objects, or allows different views of them. With such a mechanism, each object is a list or set of instances representing consecutive or alternative states of the object (note that applications can use ordinary relations between objects to achieve much the same effect). PLINTH does not at

present support historical versions, but applications may need to deal with several drafts of a document, stages in the design rationale process, etc. On the other hand, PLINTH does provide multiple views of nets, consisting of alternative sets of attributes for net, node and link objects. A versioning mechanism could therefore be useful.

4.2.3 Convenient and efficient text storage

The database should not limit the size of object attributes to the extent that the ability to associate very long text fragments with PLINTH nodes is affected. Maximum sizes of the order of several megabytes are quite acceptable.

4.2.4 Low cost

There are a number of public domain OODBs around. For experimental purposes at least, PLINTH should use one of these if at all possible.

4.2.5 Widespread use

There are two reasons for selecting a popular OODB. Firstly, more widely used it is, the better its quality is likely to be. Secondly, we might want to make PLINTH data available to other applications, for example statistical and data profiling tools. The more popular the OODB is, the more application packages we are likely to find.

4.3 Summary

An OODB which can be used by PLINTH should possess the following features:

Essential features
Complex objects and object identity
Persistance, efficient secondary storage management, concurrency and recovery
C language API
Runs on SPARCstations

Desirable features
Extensible class hierarchies with overriding, overloading and late binding
Schema evolution
Encapsulation
Versioning
Low cost
Widespread use
Runs on a wide range of platforms

5. How must PLINTH be changed to accommodate OODBs?

The main work that has to be done to allow PLINTH to use an OODB is the reimplementation of the CONNEKT subsystem to make it create and manipulate

external database objects instead of the records shown in figure 1. In order for this to be done with minimum disruption to the rest of the system code, the programmatic interface to CONNEKT should stay essentially the same.

However, at the moment CONNEKT data is not sufficiently well hidden—other PLINTH modules can directly access the fields in the net, node and link record structures in order to get information about the objects. The first stage of the process is therefore to **encapsulate** the CONNEKT subsystem fully, to formally define and fix its own API (this has only been done informally up to now), and to ensure that rest of the PLINTH system code adheres strictly to the defined interface. Once this has been done, we will be free to reimplement either CONNEKT or the GUI independently of each other, whenever it becomes necessary to port PLINTH to a new OODB or a new window system. Given that the CONNEKT code is already quite modular in most repects, this will not be a major task.

This definitive CONNEKT API should of course take into account the special requirements of a possible OODB implementation, even though this will not necessarily be transparent to the application program (i.e. the rest of PLINTH). For example, consider the **n_linksfrom** and **n_linksto** fields of the *node* record in figure 1. The contents of these fields are Pop11 lists of *link* instances. However, in an external OODB with a C interface, we might well have definitions similar to those in figure 2.

When the Pop11 application program needs to know the links from or to a particular node, we do not want to have to create a new Pop11 list every time to shadow the external *linklist* structure, as this will cause garbage problems. Neither do we want to simply pass the Pop11 program an external pointer to the C structure and expect it to know how to traverse it, as this will break the data encapsulation. The solution is not to have the API provide an access procedure that returns a list of links (as it does currently), but rather a procedure that returns the next link each time it is called, until there are no more (i.e. in Pop11 terms we need a `link repeater').

```
typedef struct {
        net             net;            ;;; net containing this node
        int             class;          ;;; class id
        char            *name;          ;;; name of node
        linklist        linksfrom;      ;;; links connecting from node
        linklist        linksto;        ;;; links connecting to node
        slotlist        slots;          ;;; node slots
} *node;

typedef struct {<fields>} *link;

typedef struct {
        link            *links;         ;;; array of links
        unsigned int    length;         ;;; number of links
} linklist;
```

Figure 2: C definitions of PLINTH data

It is also likely that some existing CONNEKT procedures and structure fields will become redundant when an OODB implementation is used, for example the **loadnet** and **savenet** procedures (though the latter might be changed to call a `commit' function in an OODB with atomic operation sequences as described in 2.2 above).

6. Conclusions

Object-oriented database technology can be used as the basis for future versions of PLINTH that provide efficient management of very large document corpora with safe, concurrent access by multiple users. Preparing the existing system for this development is a relatively straightforward task, which is largely independent of the actual database eventually chosen. The appendix to this paper gives details of some of the available OODB systems that (more or less) meet the specific requirements described above. It will be continually updated and revised as details of suitable systems become available.

References

1. Casson, A. (1993). "PLINTH: Integrating Hypertext, Semantic Nets and Rule-Based Systems in an Expertext Shell for Authors and Readers of Regulatory Documents", in J. Mayfield and C. Nicholas (eds.), *Proceedings of Workshop on Intelligent Hypertext CIKM '93*, Arlington VA, November 1993.

2. Casson, A. (1994 forthcoming). "Technical overview of PLINTH". AIAI-TR-???, University of Edinburgh.

3. Lee, J. amd K.-Y. Lai (1991). "A Comparative Analysis of Design Rationale Representations". Center for Coordination Science TR #121, MIT, May 1991.

4. Atkinson, M., D. F. Bancilhon, D. DeWitt, K. Dittrich, D. Maier and S. Zdonik (1990). "The Object Oriented Database Manifesto".

5. Zdonik, S.B. and D. Maier (1990). *Readings in Object-Oriented Database Systems*. Morgan Kaufmann, Inc.

6. Casson, A. (1994). "PLINTH, HTML and the World Wide Web". AIAI-TR-154, University of Edinburgh, July 1994.

7. Knight, S. (1993). `The Design of ObjectClass, a Seamless Object-Oriented Extension of Pop11", in *Applications and Innovations in Expert Systems, Proceedings of Expert Systems 93*, Cambridge, December 1993.

APPENDIX A: Candidate OODBs

A.1 GRAS

The GRAS system, developed at the Technical University of Aachen, is a public domain OODB which uses a graph oriented data model. It provides:

- typed, attributed nodes;
- typed, directed edges;
- temporary/volatile generic sets, relations and lists;
- declaration of *graph schemes* and incremental evaluation of derived attributes;
- nested transactions with commit, undo and redo operations;
- concurrency control and error recovery;
- version control;
- C language API.

The *graph schemes* are similar to CONNEKT's network consistency models (NCMs), in specifying inheritance hierarchies and the syntactic properties of graphs, such as which types of nodes can be connected by each type of edge.

All these features make GRAS almost perfectly suited for use by CONNEKT in PLINTH, but unfortunately it has one limitation: edges are not first-class objects in GRAS, and it is not possible to attach attributes to them, as CONNEKT requires. However, attributed links can be simulated by using nodes to represent links, *i.e.* instead of **node—edge—node** chains with a variety of edge types, we would have **node—***from***—linknode—***to***—node** where *from* and *to* are the only edge types allowed. The only drawback with this solution is that the built-in graph scheme syntax facilties become much less directly useful to CONNEKT. However, since most OODBs don't even use a graph model at all, GRAS is still a very good candidate for use in PLINTH, and some preliminary experiments with providing a Pop11 API to it have been done already.

Reference:

Kiesel, N., A. Schürr and B. Westfechtel (1992). "Design and Evaluation of GRAS, a Graph-Oriented Database System for Engineering Applications". Lehrstuhl für Informatik III, Technical University of Aachen.