ABSTRACT

Graphical Image Persistence and Code Generation For Object Oriented Databases

**by
Subrata Chatterjee**

Attached is the detailed description of the design and implementation of *graphical image persistence* and *code generation for object oriented databases.* Graphical image persistent is incorporated into a graphics editor called **OODINI**. **OODINI** creates and manipulates graphical schemas for object-oriented databases. This graphical image on secondary storage is then translated into an *abstract*, generic code for dual model databases. This abstract code, **DAL** can then be converted into different dual model database languages. We provide an example by generating code for the **VODAK** Data Modeling language. It is also possible to generate a different abstract language code, **OODAL** from a graphical schema. This language does not have any dual model database architectural dependencies.

GRAPHICAL IMAGE PERSISTENCE AND CODE GENERATION FOR OBJECT ORIENTED DATABASES

by
Subrata Chatterjee

A Thesis
Submitted to the Faculty of
New Jersey Institute of Technology
in Partial Fulfillment of the Requirements for the Degree of
Master of Science
Department of Computer and Information Science
May 1992

APPROVAL PAGE

Graphical Image Persistence and Code Generation for Object Oriented Databases

by
Subrata Chatterjee

Dr. Yehoshua Perl, Thesis Adviser
Professor of Computer and Information Science, NJIT

BIOGRAPHICAL SKETCH

Author: Subrata Chatterjee

Degree: Master of Science in Computer and Information Science

Date: May, 1992

Date of Birth:

Place of Birth:

Undergraduate and Graduate Education:

- Master of Science in Computer and Information Science, New Jersey Institute of Technology, Newark, NJ, 1992
- Bachelor of Science in Computer and Information Science, New Jersey Institute of Technology, Newark, NJ, 1987

Major: Computer and Information Science

This thesis is dedicated to
my parents and family.

ACKNOWLEDGEMENT

The author wishes to express his sincere gratitude to Professor Yehoshua Perl and Professor James Geller for their guidance, friendship, and moral support throughout this work.

The author also wishes like to acknowledge and express appreciation to the following people for their invaluable contribution to this document: Mike Halper, Ashish Mehta, Ram Mohan Madapati, Steve Beck and Walter Jones.

TABLE OF CONTENTS

Page

1 INTRODUCTION 1

1.1 Scope 1

1.2 Audience 2

1.3 Terminologies 2

2 GRAPHICAL IMAGE PERSISTENCE 7

2.1 Problem Statement 7

2.2 Design Solutions 8

2.2.1 A Traditional Approach 8

2.2.2 A Generic Approach 9

2.3 Implementation Details 13

2.4 Design Advantages 15

2.5 Performance Evaluation 17

2.6 Porting Issues 18

3. CODE GENERATION 20

3.1 Problem Statement 20

3.2 Implementation of DAL 22

3.2.1 Object Types 22

3.2.2 Attributes 23

3.2.3 Set-of and Member-of 24

Page

3.2.4 Category-of.. 25

3.2.5 Role-of.. 26

3.2.6 Part-of.. 27

3.2.7 Tuple-of.. 28

3.2.8 Ordinary Relationships.. 29

3.2.9 Essential Relationships.. 30

3.2.10 Multi-valued Relationships.. 31

3.2.11 Dependent Relationships.. 32

3.2.12 Multi-valued Essential Relationships.. 32

3.2.13 Multi-valued Dependent Relationships.. 33

3.2.14 Methods.. 34

3.3 Syntax For DAL.. 35

3.4 User API For DAL.. 35

3.5 Implementation of Structural and Semantic Hierarchy.. 38

3.6 VML Code Generation.. 39

3.6.1 Conversion Of Object Types.. 39

3.6.2 Conversion Of Object Classes.. 44

3.7 Implementation Of OODAL.. 48

3.7.1 Implementation Of Object Classes.. 49

3.7.2 Implementation Of Attributes.. 49

3.7.3 Implementation Of Relations.. 50

Page

3.7.4 Implementation Of Relationships.. 50

3.7.5 Implementation Of Methods.. 51

3.8 Syntax For OODAL... 51

3.9 User API for OODAL.. 52

4 CONCLUSIONS.. 54

APPENDIX A: MANUAL PAGES: IMAGE PERSISTENCE........................ 56

APPENDIX B: BNF DESCRIPTION FOR DAL.. 66

APPENDIX C: USER API HEADER FILE FOR DAL................................... 68

APPENDIX D: USER API LIBRARY CALL FOR DAL................................ 71

APPENDIX E: MANUAL PAGE: DAL EXECUTABLE............................... 73

APPENDIX F: MANUAL PAGE: VML EXECUTABLE.............................. 74

APPENDIX G: MANUAL PAGE: HEAP FILE COPY.................................. 76

APPENDIX H: BNF DESCRIPTION FOR OODAL...................................... 77

APPENDIX I: USER API HEADER FILE FOR OODAL.............................. 79

APPENDIX J: USER API LIBRARY CALL FOR OODAL........................... 81

APPENDIX K: MANUAL PAGE: OODAL EXECUTABLE......................... 83

APPENDIX L: SOURCE TREE AND SOURCE COMPILATION................. 84

REFERENCES.. 88

LIST OF TABLES

Table Page

1 DAL Source File Names.. 84

2 DAL Executable/Library File Names.. 85

3 OODAL Source File Names.. 85

4 OODAL Executable/Library File Names.. 85

5 VML Source File Names.. 86

6 VML Executable File Name.. 86

7 MAPAMLLOC Source File Names.. 86

8 MAPMALLOC Object File Name.. 87

9 OOCOPY Source File Names... 87

10 OOCOPY Executable File Names.. 87

LIST OF FIGURES

Figure		Page
1	Memory Mapping of Persistent Heap File	12
2	Persistent Heap File Configuration Layout	15
3	Data Flow Diagram For Code Generation	21
4	Data Flow Diagram for VML Code Generation	22
5	OODINI Attribute Graphical Notation	24
6	OODINI Setof/Memberof Graphical Notation	25
7	OODINI Categoryof Graphical Notation	26
8	OODINI Roleof Graphical Notation	27
9	OODINI Partof Graphical Notation	28
10	OODINI Tupleof Graphical Notation	29
11	OODINI Ordinary Relationship Graphical Notation	30
12	OODINI Essential Relationship Graphical Notation	30
13	OODINI Multi-valued Relationship Graphical Notation	31
14	OODINI Dependent Relationship Graphical Notation	32
15	OODINI Multi-valued Essential Relationship Graphical Notation	33
16	OODINI Multi-valued Dependent Relationship Graphical Notation	34
17	DAL User API Data Structures	37
18	VML Semantic MetaClass Keywords	47

Figure **Page**

19 OODAL User API Data Structures.. 53

20 Source Code Tree Hierarchy.. 84

CHAPTER 1
INTRODUCTION

This document describes in detail the design and implementation of *graphical image persistence* and *code generation for object oriented databases.* In order to represent large networks of classes, their attributes and relationships for an object oriented database a graphics editor called **OODINI** (**Object-Oriented Diagrams at the New Jersey Institute of Technology**) was designed and developed in the Computer Science Department at New Jersey Institute of Technology **[2]**. **OODINI**'s intent was to facilitate designers to graphically represent object oriented database schemas. A major portion of the work detailed in this document entails secondary storage representation of such graphical schemas, i.e., *graphical image persistence.* The obvious benefit derived is the iterative execution and update operations of the same graphical schema under **OODINI**. This graphical image representation on secondary storage is then translated into an *abstract*, generic code, **DAL** (Dual Model Abstract Language), for **dual model** databases. A complete description of dual model object oriented databases can be found in **[1], [4] and [5]**. Briefly, in the dual model there is a distinct separation of the structural and semantics aspects in the definition of an object class **[6]**. This abstract code, **DAL**, can then be converted into different object oriented database languages. We provide an example by generating code for the **VODAK** Data Modeling Language (**VML**). A complete description of **VML** semantics and syntax can be found in **[3]**. It is also possible to generate a different *abstract* code from a graphical schema. This code, **OODAL** (OODINI Abstract Language), does not have any dependencies on the dual model database architecture.

1.1 Scope

This document is intended for publication towards a Master's Thesis at NJIT.

1.2 Audience

This document is intended for system planners, architects, developers and testers of **Dual Model Object Oriented Database** project ongoing at New Jersey Institute of Technology. For other audiences the materials in the **Reference Section** is a mandatory prerequisite to this document.

1.3 Terminologies

API: Application programming interface.

Attribute: A structural aspect of a class that is composed of a name and a data type.

BNF: Backus-Naur Form. **BNF** is a **metalanguage** for programming languages. A metalanguage is a language that is used to describe yet another language. **BNF** is used to describe the syntax of a programming language. It uses abstractions for syntactic structures.

Category-of: A semantic relation between two classes. It relates a specialized class to a more general class where both these classes are viewed within the *same* application context.

Class: A container of objects which are similar in their structure and their semantics.

DAL: **D**ual Model **A**bstract **L**anguage. The graphical image for a database schema is first converted to this abstract language and then to other object oriented database languages. The **DAL** syntactic form closely resembles the language proposed under the Dual Model architecture.

Dependent relationship: A relationship where the existence of an object depends on the existence of yet another object. If the class **A** has a dependent relationship to class **B**, then the existence of an instance of **A** is dependent on the existence of an instance of

B. That is, if an instance **a** of **A** depends on an instance **b** of **B**, and **b** is deleted, then **a** must also be deleted.

Dual Model: A database model in which the object type description is separated from the description of object class. This model gives a clear distinction between the structural and semantic elements in the definition of an object class.

Essential Attribute: The existence of an object is conditioned on the existence of this attribute. An instance of a class can only exist if the value of its essential attributes are all different from **NIL**.

Essential Relationship: A relationship which is not permitted to have a **nil** value.

Heap: Described within the context of a programming language. It is that portion of a user process's virtual address space from which dynamic memory is allocated.

Malloc: A memory allocation routine. Allocates dynamic memory from a user process's virtual address space. This dynamic memory is *not* persistent across execution sessions. This routine is typically implemented as a library call in **UNIX** operating systems.

Member-of: A connection between two object types. Here an object type is said to *belong-to* or be a *member* of another object type - the latter object type representing a set. This is also a relation.

Method: A program segment with one required parameter of some object type, and any number of optional parameters. A method always returns a value of an object type or data type.

Mmap: Memory mapping of objects. This is an operating system terminology. It represents a system call that can map objects (e.g., files, devices, etc) into a user process's virtual address space.

Multi-valued relationship: A one-to-many relationship between two classes. It indicates that an instance of one class can be related to any number of instances of the class to which the relationship is directed. An example of this can be the relationship between the classes **section** and **student**, where a given section can have many students.

Part-of relation: A relation which is used to connect a *part* of a complex or assembled (*real-world*) object to its integral object. An example of this relation can be used to represent *parts* of a book, e.g., the classes **chapter** and **page** can be in a *part-of* relation with the class **book**.

Object: The concept of an object is universal. Literally everything, from items as simple as the integer constant 1, to a file-handling system, memory, data structures, etc., are objects. As objects, they are treated uniformly. Objects have local memory, inherent processing ability, the capability for communicating with other objects, and the ability to inherit characteristics from ancestor objects.

Object type: In order to express that all instances of a class have a common structure and behavior one can consider them to be of the same *abstract* data type. This type is called the **object type** of that class.

OODAL: OODINI Abstract **L**anguage. The graphical image for a database schema is first converted to this abstract language and then to other object oriented database languages.

OODINI: Object-Oriented Diagrams at the New Jersey Institute of Technology. A graphics editor for drawing and manipulating object oriented database schemas.

Part-of: A connection of a part of a complex or assembled (real-world) object to its integral object.

Relations: Generic or system defined connection between object types or object classes. **Set-of**, **member-of** and **subtype-of** constitutes relations between object types. **Category-of**, **role-of** and **part-of** constitutes relations between object classes.

Relationship: User defined connection between classes that can contain either structural or semantic information in the context of the application.

Role-of: A semantic relation between two classes. It relates a specialized class to a more general class, where both these classes are viewed in *different* application contexts.

Semantic aspect: An aspect of a specification is considered to be semantic if either (1) it refers to actual instances of objects in the application or (2) one cannot decide whether this aspect of information describes an object properly, based solely on its mathematical structure and without relying on an intuitive understanding of the application.

Set-of: A connection between two object types. Here an object type represents a set of other *member* object types. In a mathematical sense this is also a relation.

Structural aspect: An aspect of a specification is considered structural if either (1) it is composed of names, types and logical arithmetic operations, or (2) one can decide whether this aspect of information describes an object properly based solely on the mathematical structure of this aspect, without relying on an intuitive understanding of the application.

Tuple-of relation: A relation constructor used to gather a group of classes (*constituent classes*) into a single class (*the tuple class*) for some purpose. A concrete example of this can be the *tuple* class **shipment** which is involved in a ternary relation with its *constituent* classes **supplier**, **product** and **department**.

VML: The **VODAK** Data Modeling Language.

CHAPTER 2
GRAPHICAL IMAGE PERSISTENCE

GRAPHICAL IMAGE PERSISTENCE OODINI is a graphics editor that allows designers to graphically represent dual model database schemas. It runs under a **SUN-SPARC** work station in a **UNIX** operating system environment. The application makes use of **X-Window** and **Motif** tools for creation and manipulation of graphical images representing classes, attributes, relations, relationships, methods, etc, of a dual model database schema. This section of the document describes the problems associated with obtaining secondary storage representation of such graphical objects, describes a traditional solution approach, presents an *innovative generic* solution, its implementation details, performance statements and software porting issues.

2.1 Problem Statement

It is obvious that the graphical representation of any object *as viewed on* a screen need not be of concern. It is the *internal memory representation* of the graphical object by the application (in our case, **OODINI**) that needs to be saved on secondary storage. This is because any application will eventually invoke *primitive* library routines (e.g., **X-Window** library calls) to draw the physical image on the screen using the internal memory representation of the object. Hence with each graphical object, whether it be a line, segment, text, arc, etc., there has to be an associated internal memory representation. This internal memory representation of a graphical object also has to be a part of user's program (**OODINI**) execution image. More specifically this memory can either be a part of the user's data segment or the heap *(from which dynamic memory is obtained)*. It should now be evident that if we can achieve **data persistence** of the user data space and the heap - we will also have achieved graphical image persistence. This is the linchpin of our *proposed* design solutions.

2.2 Design Solutions

First we present a crude, traditional design approach to save graphical images generated by **OODINI** on secondary storage. Then we present a more elegant, sophisticated and *generic* design that will work not only for **OODINI** but for other graphical editors as well. The scope of the latter approach is however not restricted to graphical editors. This new design can be useful in other application environments - this will be justified appropriately.

2.2.1 A Traditional Approach

A rather crude, cumbersome and traditional approach to obtain data persistence for an executable's execution environment is to write out each internal memory representation of graphical objects one at a time to a file on secondary storage. It is obvious that this should be performed prior to execution termination. The primary obstacles to this approach are outlined below:

1. **Implementation** of this design can be tedious and time consuming. It is true that it is easy to write out individual memory objects to secondary storage. But the major implementation hurdle lies in dealing with *pointers*. Data variables in memory will for the major portion use *pointer variables* to reference *other* data objects. It is easy to save, for example, the name of a class in the file. On the other hand, directly writing out the contents of pointer variables will never work. In such case the developer needs to come up with a *labeling scheme* that associates a label (usually an integer variable) with each graphical object. This requires a detailed understanding of the graphics editor and its internal data structures.

2. **Performance** of the software is also an issue. Prior to termination of the graphics editor, expensive searches on extensive data structures needs to be made and each object must be written back to secondary storage one at a time. Prior to program

execution the image has to be loaded back from secondary storage and the data structures needs to be restored to their original state. Besides such I/O considerations memory allocation/deallocation also creates performance problems. Image saving and restoring for large number of graphical objects can easily lead to *sluggish* software response.

3. **Maintenance** of this software will become *tedious*. For example assume that a new object is incorporated into the design or a new *pointer* variable is added to an existing data structure. This is typical in a development organization where several developers might attempt modifications to the same software. Every time such changes are made - a corresponding change needs to be made to the portion of the software that writes out the object to secondary storage. Hence at least one developer needs to be maintained for this software every time such modifications are anticipated. There is either a cost issue involved with maintaining a developer for this software or it requires *every* developer that makes changes to the graphics editor to have knowledge about that piece of the software that writes out images to secondary storage.

4. Furthermore this obviously is not a *generic* solution. If a new graphics editor other than **OODINI** is targeted for development, the current software that writes out images to disk will have to be changed extensively. This solution is heavily dependent on internal data structure representation.

In any case this design approach is attainable but hardly desired.

2.2.2 A Generic Approach

In this section we *propose* a more *generic* solution to our problem. Our solution attempts to achieve **user dynamic memory persistence of the heap space** of an execution image. We seek a mechanism where all updates to the heap of a *running* executable is *automatically* written to secondary storage; and to demonstrate persistence we also need a mechanism to restore the heap to its *existing/old* state prior to program execution. In order to grasp this design approach one needs to understand some **UNIX** Operating System concepts. We proceed to explain a relatively *new* feature of the **UNIX** Operating System - the **mmap()** system call.

The word **mmap** implies memory mapping. It is a **UNIX** system call that establishes a mapping between a process's address space and a virtual memory object, e.g., a device or a file. In our case the object being mapped is a flat **UNIX** file. Traditionally, in the **UNIX** world accesses to files were done using the standard *read()* and *write()* system calls. Using *mmap()* on an *existing* file allows users to manipulate the file without using *read()* and *write()* system calls. Once a file is mapped into the user virtual address space, all the process has to do to access the file is to use the data at the address range to which the file was mapped. Assume that the user requests a mapping of an existing file and the operating system maps the file at user virtual address **V**. The virtual address range accessible to the user would then be from **V** to **V + N**, where **N** is the length of the file. Then if the user process writes to location **V** this would be equivalent to writing out data in file at *logical* offset 0. If the user process reads from location **M**, where **V <= M < V+N**, this would be equivalent to reading from the file at *logical* offset **M - V**. In other words, by touching memory address from **V** to **V+N** the user process can manipulate the entire file. It should be noted that mappings to objects need not start at logical offset 0 of the object being mapped.

There are two ways of mapping an object. One can achieve a **private** mapping to a file using the **MAP_PRIVATE** flag as an argument to the *mmap()* system call. In such a case, whenever the user attempts to update data in the file by writing memory in the *mmap()*-ed address space, the user gets its own, private copy of the physical page being touched. This is analogous to how the **copy-on-write** feature works in **UNIX**. When a process creates a child process, one of them gets its own private page during the first write attempt to that page. Hence, when a process maps a file using the **MAP_PRIVATE** feature, all the pages corresponding to that file are initially marked as **read-only**. A write access on any page causes **copy-on-write** to take effect. When the operating system is low on memory, it tries to swap pages out. When **copy-on-write** occurs on a page, the page becomes *dirty* in memory. Such dirty pages are then swapped to the swap device.

Another way of mapping a file is to use the **MAP_SHARED** flag. In such a case, all processes can *share* the file - everybody has the right to read and write pages corresponding to the file. During swapping activity such *dirty* pages are written *back* into the file, i.e., these pages do not have swap-device associations. An interesting feature to note is that it is the operating system's responsibility to flush *dirty* pages for a file back to secondary storage -- it is not the user process's responsibility (this is very different from *read()/write()* system calls). So even if the user process terminates prematurely - the operating system will flush out the *dirty* pages to disk.

The following diagram summarizes the discussion of the **mmap**() system call.

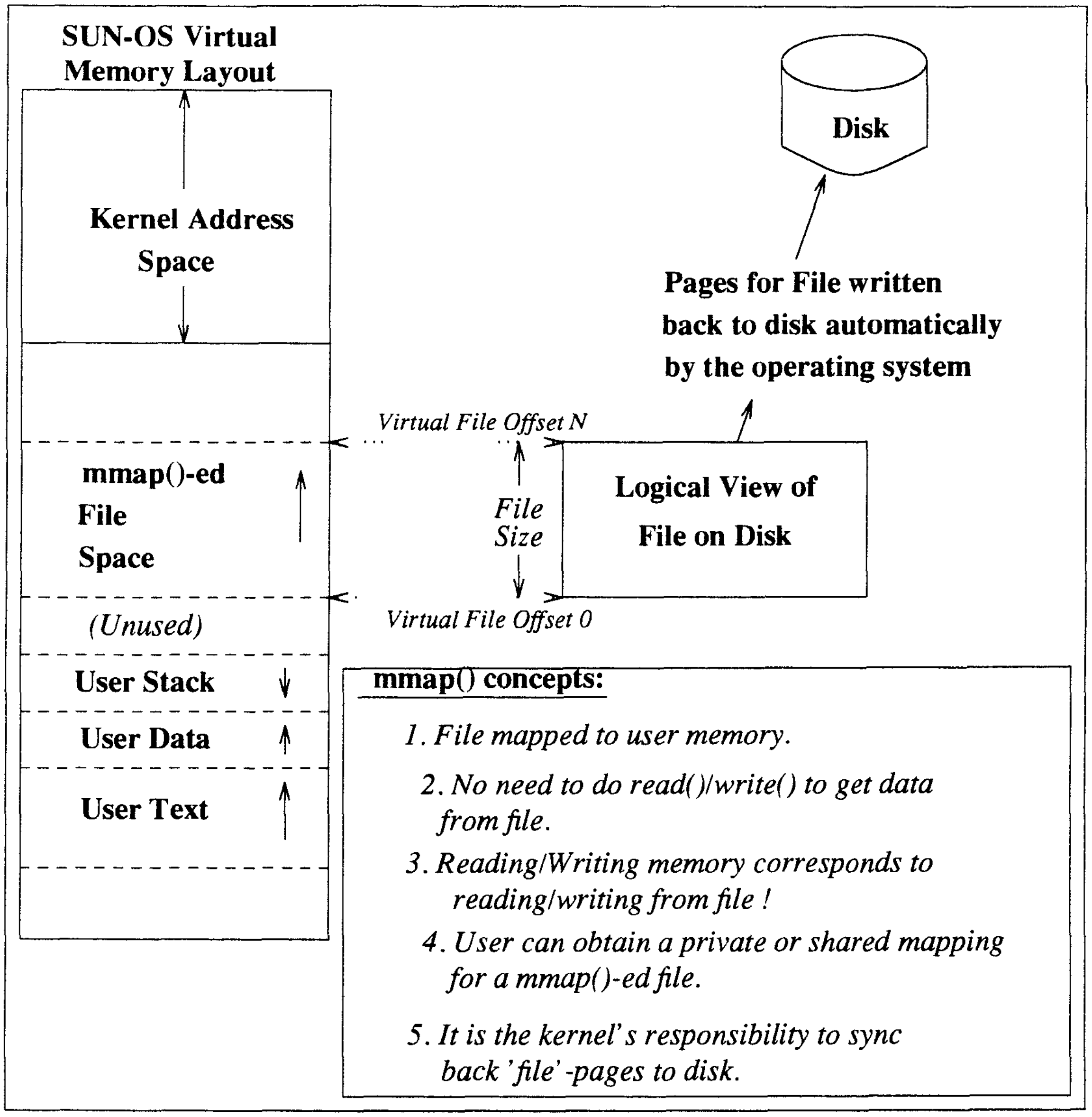
SUN-OS Virtual Memory Layout
Kernel Address Space
mmap()-ed File Space
(Unused)
User Stack
User Data
User Text
Disk
Pages for File written back to disk automatically by the operating system
Virtual File Offset N
File Size
Virtual File Offset 0
Logical View of File on Disk
mmap() concepts:
1. File mapped to user memory.
2. No need to do read()/write() to get data from file.
3. Reading/Writing memory corresponds to reading/writing from file !
4. User can obtain a private or shared mapping for a mmap()-ed file.
5. It is the kernel's responsibility to sync back 'file'-pages to disk.

Figure 1. Memory Mapping of Persistent Heap File

Our solution uses the **mmap()** system call to achieve persistent of the **heap** of any user process. During process execution, the process will allocate dynamic memory from the heap. This allocated memory would then be used to represent graphical images. Our solution to achieve persistent is extremely *simplistic* - **instead of allocating dynamic memory from the traditional heap space, we allocate memory from a mmap()-ed file**. How this is done is discussed in the next section.

2.3 Implementation Details

In the **UNIX** world dynamic memory allocation is done from the **heap** using the **malloc**() library call. Dynamic memory deallocation is done using the **free**() library call. These two routines perform necessary memory management of the user process's heap address space. Briefly, it maintains a linked list of blocks of memory that are currently allocated/used and another linked list of blocks of memory that are currently being unused or were deallocated. It maintains a header section (a block of memory) in the heap address space - that maintains pointers to the beginning of each linked (used/free) list. However note that the pages in memory corresponding to the heap address space are **not** associated with any file, these pages are associated with the swap device. During swapping activity these pages are swapped back and forth from the swap area. Upon program termination the operating system simply *throws* (returns pages to the free page pool) these pages out, and frees the swap space reserved for these pages. Such memory that have swap association are often referred to as **anonymous memory**.

Our solution implements the entire heap of a user process in the *mmap()*-ed address space. This simply means that all memory allocation and deallocation buffers, list pointers, user data, etc. - are **preserved** during process execution and even after process termination. Remember, it is the operating system's responsibility to write out pages for *mmap()*-ed address space. We use the **MAP_SHARED** flag during the *mmap()* system call, so that all updates to the heap are reflected back into a file that was mapped. We provide a *new* **library** - **libmapmalloc.a** that provides equivalent functionality of the traditional *malloc()/free* library calls. More specifically, the following library routines are critical from the implementation point of view:

1. **mapmalloc_init**() - This library routine takes in as an argument a file name. If the file does not exist it creates one. It then maps the file into user memory. The *initial* file created is of a default or user specified size.

2. **mapmalloc**() - This library routine takes in as argument the number of bytes to allocate from the heap (the *mmap()*-ed address space). If the file contents are full (*heap overflow*) this routine first *unmaps* the file from memory, grows the file by a default size and *maps* the file back into memory.

3. **mapfree**() - This library routine frees a previously allocated block of memory. The heap (*mmap()*-ed address space) is updated to reflect the change.

4. **map_set_base_addr**() - This library routine saves a pointer value in the *header* portion of the file being mapped. *Why is this needed?* It is true that we have achieved dynamic memory persistence once a program finishes execution. But when it starts execution again - how does it know which portion of the dynamic memory is the *starting point* of all its data structures? That is, the program upon re-execution needs to know the **root** pointer(s) of all its data structures. Notice that all the root pointers of multiple data structures can be in turn saved into a single data structure. This routine simply saves that root pointer in the file's header section.

5. **map_get_base_addr**() - This library routine returns to a program the saved **root** pointer of all its data structures. If a **NULL** value is returned it implies that either the user did not perform a *map_set_base_addr()* in its prior execution state or the file being mapped is in its initial state.

It is worthwhile to note that our actual implementation allows for multiple heaps (and multiple corresponding files) to be managed by the library. Multiple heaps can modularize heap management of a process. A complete description of the usage of these and other library routines are given in standard **UNIX** manual page formats in **APPENDIX A.**

We also provide a high level diagram depicting the contents of a sample memory mapped file being used as a heap.

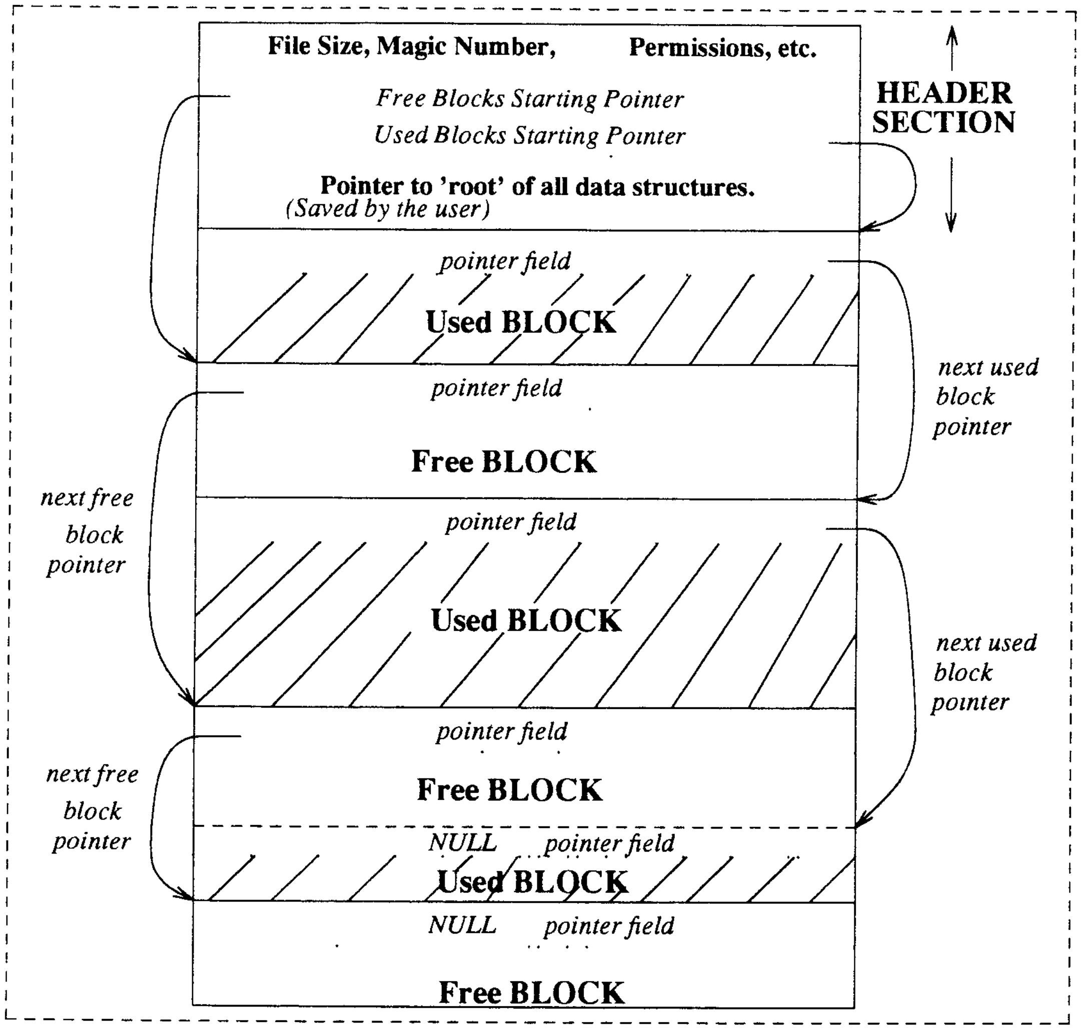

Figure 2. Persistent Heap File Configuration Layout

2.4 Design Advantages

In this section we justify why this solution is *generic* and can even be used outside our particular programming environment. Consider the following observations:

1. **Genericity:** Traditionally **UNIX** does not provide persistence of dynamic memory allocation. This is the *first time* such a solution has been concretely proposed. Dynamic memory persistence is useful in other programming environments - it is not restricted to graphical editors. Any data structure, whether a complex network of nodes or a simple linked list, stack, queue, etc. - can easily

be saved on disk - without requiring the programmer to worry about its internal representation on secondary storage.

2. **Performance:** It is proven that *mmap()* is faster than using *read()* and *write()* system calls. This is because during a *read()* system call there is an *additional* overhead of copying data from disk to the kernel buffer cache and then to the user address space. Similarly during a *write()* system call the data is first copied from the user address space to the kernel buffer cache and then to disk. It should be noted that the implementation of the *mmap()* system call in **UNIX Release 4.0** was possible as *the page pool is the buffer pool*! During *mmap()* copying of data from/to the kernel buffer cache is **not** necessary.

3. **Sharing:** When processes create other children processes - there is no easy way of allowing them to *share* a single heap. Shared memory regions are an option - but cumbersome. With our solution multiple processes can work with a single heap. Semaphore control operation features during memory allocation and deallocation are built into the library to resolve concurrency issues. This also implies that while the user is creating a graphical schema under **OODINI** we can also dynamically generate code for that schema, although **OODINI** is in execution mode.

4. **Reliability:** Using *mmap()* rather than usual *read()/write()* system calls makes the software more reliable. The user is relieved of writing persistent data to disk! The operating system guarantees all data in the *mmap()*-ed address space will *eventually* be written back to disk. It is very rational to assume that the user software or the underlying graphical application (e.g., **X-Window**) might abort execution prematurely (e.g., *core dumps*, **kill** signal, etc.) without giving the user software control to write out the data to disk. With *mmap()*, it is the kernel's responsibility to sync back last changes to the memory mapped file(s) to disk.

Finally, the operating system might crash. On some *fault tolerant* machines, e.g., **Tandem**, the kernel *attempts* to flush the buffer pool to disk before taking the hardware out of service. Since in **UNIX Release 4.0** the *page pool is the buffer pool*, all memory mapped files will be synced back to disk during kernel panics.

5. **Modularity:** Our solution also allows for creation of multiple heaps within a single process's address space. This allows modular programming and better maintenance of dynamic memory usage.

6. **Ease of Programming:** In order to lessen code impact changes during transitioning of code from traditional heap management using *malloc()/free()* library calls, we have provided two new macros, **MALLOC()/FREE()** that performs the equivalent operations of *mapmalloc()* and *mapfree()*. Changes in existing code are thus minimal. We effectively provide a *malloc/free-like* interface for minimum perturbation on existing programs.

7. **Cost Effective:** With this solution there is now no need for maintaining additional developer(s) for making changes to the portions of the software that achieves persistence. The simplicity and generic nature of the design makes this an achievable goal.

2.5 Performance Evaluation

We have mentioned earlier that using *mmap()* is faster than using *read()/write()* system calls. Since our memory allocation is also based on existing *malloc()/free()* library interface, we thus do not predict any performance degradation during allocation and deallocation. However there is an obvious extra overhead. Since we are saving data into a file (actually, the operating system does it) - file block allocation will obviously be a factor. But this is an inescapable reality - data must be preserved in a file. One can argue on file space considerations. Since the *entire* file must exist prior to the *mmap()*

system call - are we not wasting file system blocks when using a large file for a heap - while memory allocation might use only a small percentage of the actual file size? This is easily avoided since **UNIX** supports *holes* within a file. All we do during file creation is to write the first and the last byte of the file, thus allocating only 2 file system blocks - leaving a *hole* in the middle of the file. Only when memory is requested from un-allocated blocks, touching (reading/writing) memory results in block allocation by the operating system. However, space and time considerations are always in conflict. Dynamic block allocation can reduce a process's response during run time. To avoid this we provide an additional flag in *mapmalloc_init()* that pre-allocates all the disk blocks for the file being created.

2.6 Porting Issues

The **library** code, *libmapmalloc.a*, is portable across all **UNIX System V Release 4.x** versions. It has been developed on a **80386 AT&T SVR 4.2** operating system and then ported to the **SUN-OS 4.1.1** system. For earlier versions of **UNIX** operating systems the *mmap()* system call is non-existent. For such systems we provide a similar interface based on **shared memory**. There is one more issue regarding porting concerns. It should be noted that *mmap()* might map a file at *different* virtual addresses on different machines. For example, *mmap()* of a file on the **80386 AT&T machine** returns a virtual address of **0x80030000** - while on the **SUN-OS** returns **0xf7000000**. Hence it should be noted that the heap image saved on a disk on one hardware platform cannot be ported directly to yet another different hardware platform. To reduce this effect, *mmap()* does provide mapping a file at a **fixed**, user defined virtual address - this feature is preserved by our library interface.

It has also been detected that on some hardware platform(s) the virtual address returned by the underlying operating system for a memory mapped file is not *always* a constant address. There *seems* to be some relationship between the virtual address allocated and the code and the size of the executable generated by the compiler for the **OODINI** executable. In such a case, future releases of the **OODINI** software *will not* work with the persistent heap files containing graphical schemas that were created with the current version of **OODINI**. We thus allow users to specify the constant, fixed virtual address at which all heap files will be mapped. This is implemented by creating a **".config"** flat file in the user's current working directory. This file contains 2 tunable values: (i) the fixed address for memory mapped files (a **hexadecimal** value), and (ii) the maximum allowable heap file size (in **bytes**). The format of this file is illustrated with the following **".config"** file used for a **SUN-OS** operating system:

```
HEAPADDRESS=e0000000
HEAPSIZE=26214400
```

CHAPTER 3
CODE GENERATION

The graphical schema from the **OODINI** graphics editor is stored on secondary storage. This schema is *parsed* and code is generated for dual model object oriented databases. This section of the document describes the problems associated with code generation, presents implementation details of an *abstract* dual model language, **DAL** (**D**ual Model **A**bstract **L**anguage), and provides a conversion algorithm for translating the abstract language to **VML** syntax and semantics. Finally, to facilitate developers who are not familiar with the dual model architecture we present a different abstract language, **OODAL** (**OODINI** **A**bstract **L**anguage). This language can be used to generate code for object oriented databases that do not support the dual model architecture.

3.1 Problem Statement

To convert the graphical schema on secondary storage directly to different dual model object oriented database languages (e.g., **VML**) would require extraordinary implementation efforts. Developers would need to know the complex internal data structure representation of **OODINI**, its heap management techniques and the representation of graphical objects on secondary storage. This will evidently introduce redundancy in development efforts. Different parsing techniques, object representation methods, etc. will evolve when individual developers translate the graphical representation to different object oriented database languages. Furthermore, future changes to the **OODINI** software will also necessitate changes to the software that generate database code.

We thus propose the conversion of the graphical representation to a *single*, *generic* abstract language, **DAL** (**D**ual Model **A**bstract **L**anguage). We provide implementation details of **DAL** using examples and also provide its complete syntax in **BNF** format. The **DAL** code generated is then translated to other (e.g., **VML**) object oriented database languages. The following diagram provides a high level data flow diagram of the entire system.

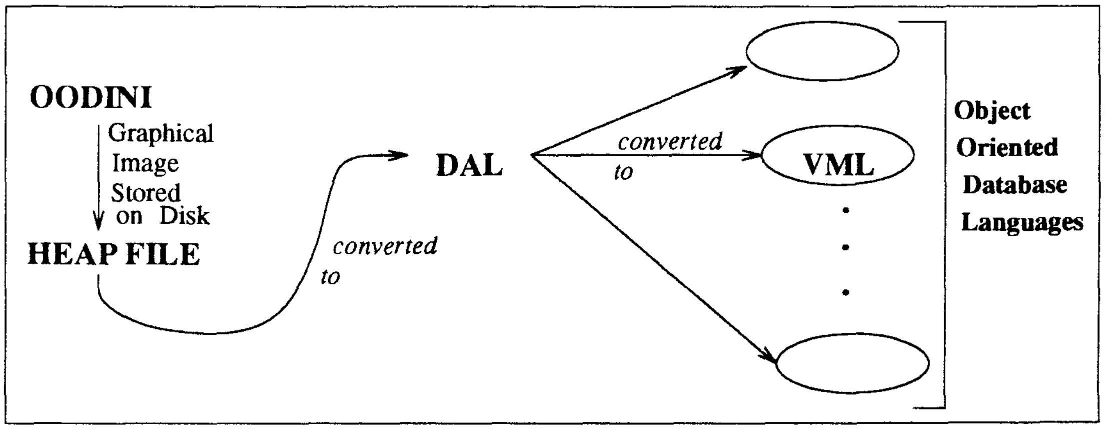
OODINI
Graphical Image Stored on Disk
HEAP FILE
converted to
DAL
converted to
VML
Object Oriented Database Languages

Figure 3. Data Flow Diagram For Code Generation

There is also a difficulty associated with generating code for an object oriented language from **DAL**. The **DAL** code generated will reside in a flat file. Developers still need to parse the file in accordance with **DAL** syntax considerations. This is not difficult but tedious as it requires developers to minimally write a token generation scanner and a parser for **DAL** code. Hence, as an alternative we also present a user application programming interface (**API**) that facilitates code generation from **DAL**. This **API** is incorporated into a library, **dallib.a**, and contains routines that returns pointers to data structures for **DAL**'s internal representation of a graphical schema. These data structures can then easily be traversed at a programming level to generate code for various object oriented database languages. The following data flow diagram summarizes this view.

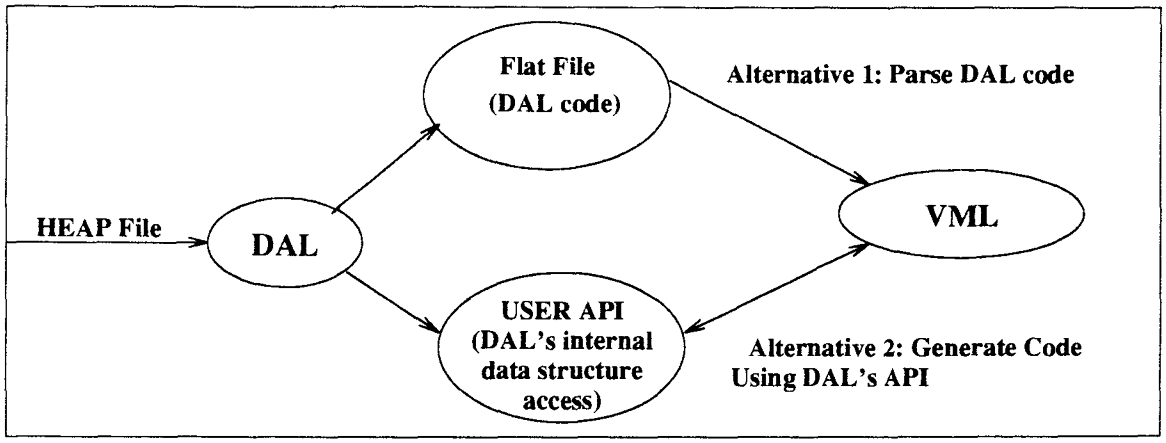

Figure 4. Data Flow Diagram for VML Code Generation

3.2 Implementation Of DAL

This section describes explicitly how each graphical object representation of **OODINI** is converted into **DAL** syntax. Rather than presenting the **DAL** syntax immediately we give examples to illustrate our implementation. The actual **DAL** syntax is described in the following section. **APPENDIX B** gives the complete **DAL** syntax in **BNF** format.

3.2.1 Object Types

Implementing object types for object classes is difficult and not straight forward. This is because **OODINI** does not allow graphical representation of object types. **OODINI** provides partial *structural hierarchy* in its dual model database schema representation. Given this limitation we proceed to create *an object type definition for each object class*. An algorithm for generating an object type specification from its corresponding object class specification has already been described in [1]. For each object class **DAL** generates the following syntactical template:

```
class   <class-name>

        objecttype          <objecttype-name>;

        <class-definition>
end;
```

The object type name is derived from the class name by suffixing the latter with the

"Type" keyword. For example, if the class name is **abc**, then its corresponding object type name is **abcType**. For each object class, **DAL** generates a corresponding object type using the following syntactical template:

```
objecttype    <class-name>Type

              <objecttype-definition>
end;
```

☞ **NOTE: In the dual model two or more object classes may share the same object type. This is not reflected in DAL. This is because it is not clearly apparent how OODINI supports this concept. In order to support this feature OODINI must be changed appropriately to incorporate this notion or partial structural integration [7, 8, 9] must be done during DAL code generation to collapse object types. This is an open issue that is currently under investigation.**

☞ **NOTE: However, we collapse two object types when the following situation arises: Object type B is a** *subtypeof* object type **A** and the *setof*, *memberof*, attribute list, relationships and methods are **not** defined for **B**. In such a case, the **NULL** object type **B** is deleted. **A** becomes the object type for the class corresponding to object type **B**.

3.2.2 Attributes

OODINI allows for attribute names only. The attribute's data type is not represented under **OODINI**. Hence when **DAL** code is generated for an attribute listing, the keyword **unknown_type** is used as a place holder. This information eventually needs to be provided by the user prior to conversion of the **DAL** code to other object oriented languages (e.g., **VML**).

Essential attributes provide semantic information and they are listed in the **<class-definition>** body. All attributes are listed in the **<objecttype-definition>** body. The following example illustrates code generation from a graphical representation of attribute listing under **OODINI** to **DAL** syntactical template:

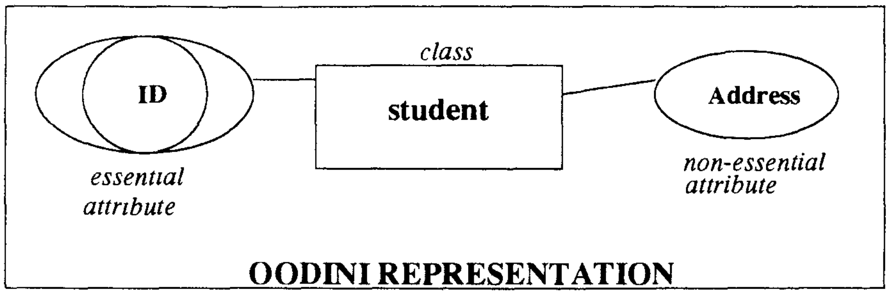

Figure 5. OODINI Attribute Graphical Notation

objecttype	studentType	**class**	student
			objecttype : studentType;
	attributes		**attributes**
	id : **unknown_type;**		id;
	address : **unknown_type;**		
	endattributes;;		**endattributes;**
end;		**end;**	

3.2.3 Set-of And Member-of

Set-of and **member-of** are relations that connect object types into object type hierarchies. Hence they represent structural information. All structural informations are incorporated in the **<objecttype-definition>** body. The following example illustrates code generation from graphical representation to **DAL** syntactical template:

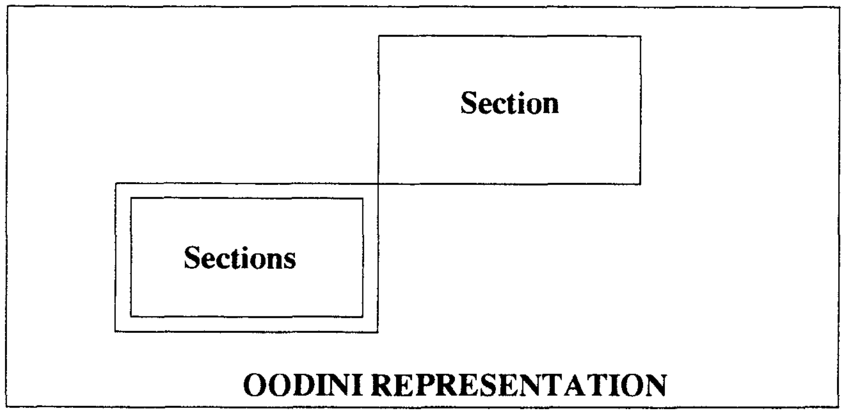

Figure 6. OODINI Setof/Memberof Graphical Notation

objecttype	sectionType	**objecttype**	sectionsType
	memberof : sectionsType;		**setof** : sectionType;
end;		**end;**	

3.2.4 Category-of

The **category-of** relation connects a specialized class to a more general class where both are viewed in the *same* application context. Since the **category-of** relation embodies a semantic relation - this information is incorporated into the <**class-definition**> body. However in [4] we find a formal proof that whenever a class **A** is a **category-of** another class **B** then the corresponding object type of **A** must be a **subtype-of** of the corresponding object type of **B**. We thus make appropriate code changes within the <**objecttype-definition**> body to reflect **subtype** inheritance. Code generation from graphical representation to **DAL** syntactical templates is illustrated below:

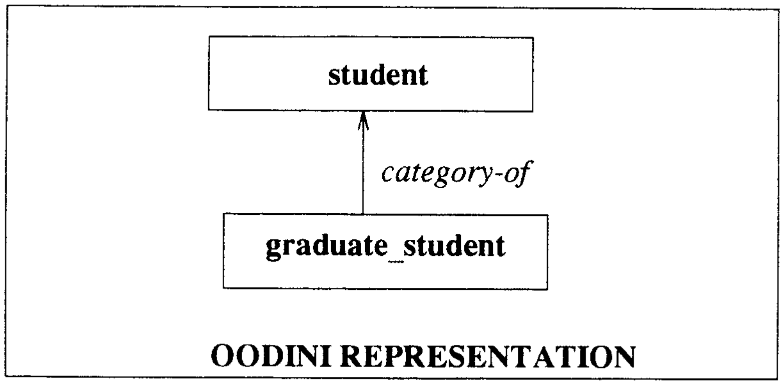

Figure 7. OODINI Categoryof Graphical Notation

objecttype graduate_studentType	**class** graduate_student
	objecttype : graduate_studentType;
subtypeof : studentType;	**categoryof :** student;
end;	**end;**

3.2.5 Role-of

The **role-of** relation connects a specialized class to a more general class, where the two classes are seen in *different* contexts of the application. Since this relation conveys semantic behavior - this information is incorporated into the **<class-definition>** body. The syntactical template generated by **DAL** from a graphical representation is illustrated below:

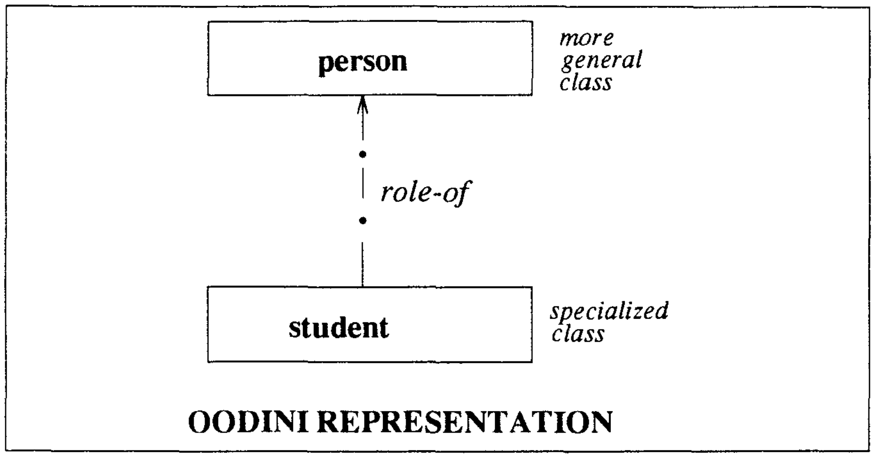

Figure 8. OODINI Roleof Graphical Notation

```
class   student

        objecttype :   studentType;
        roleof :       person;

end;
```

3.2.6 Part-of

The **part-of** relation is used to connect a *part* of a complex or assembled (real-world) objects to its integral object. This connection information is reflected both in the **<class-definition>** body and the **<objecttype-definition>** body. A syntactical template for a graphical view follows:

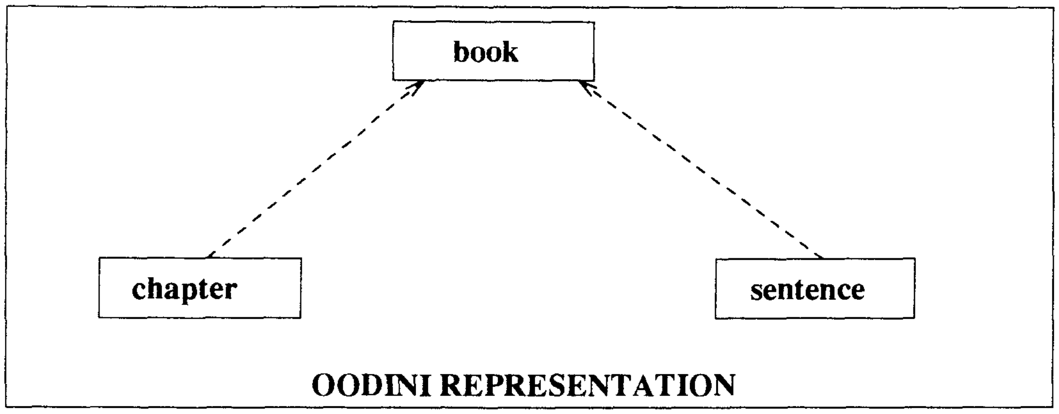

Figure 9. OODINI Partof Graphical Notation

class chapter
 	
objecttype : chapterType;
partof : book;
end;
	class sentence
	
	objecttype : sentenceType;
	partof : book;
	end;

objecttype chapterType
 	
partof : bookType;
end;
	objecttype sentenceType
	
	partof : bookType;
	end;

3.2.7 Tuple-of

The **tuple-of** relation is used to connect a *tuple* class to its constituent classes. The **tuple connector** definition is reflected in both the **<objecttype-definition>** as well as the **<class-definition>** body. A syntactical template for a graphical view follows:

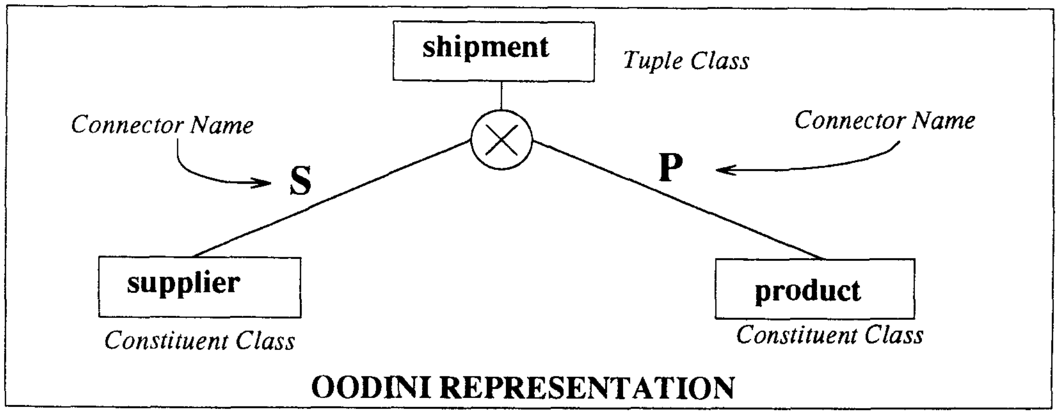

Figure 10. OODINI Tupleof Graphical Notation

objecttype shipmentType
 	
tupleof : < P : productType,
S : supplierType >;
end;
	class shipment
	
	tupleof : < P : product,
	S : supplier >;
	end;

☞ Note that in <**objecttype-definition**> body the connector names (e.g., **P** and **S**) refers to object types, while in the <**class-definition**> body the reference is to object classes. If in the **OODINI** representation, a connector name was not provided by the user then connector name defaults to the name of the corresponding constituent class.

3.2.8 Ordinary Relationships

Relationships, unlike relations, are user defined connections between classes. A relationship can convey either a structural or a semantic connection. All user defined relationships are incorporated both in the <**class-definition**> and the <**objecttype-definition**> body. In the former, the relationship reference is to an object class and in the latter the relationship reference is to the corresponding object type. The **DAL** syntactical template for its corresponding graphical representation is illustrated below:

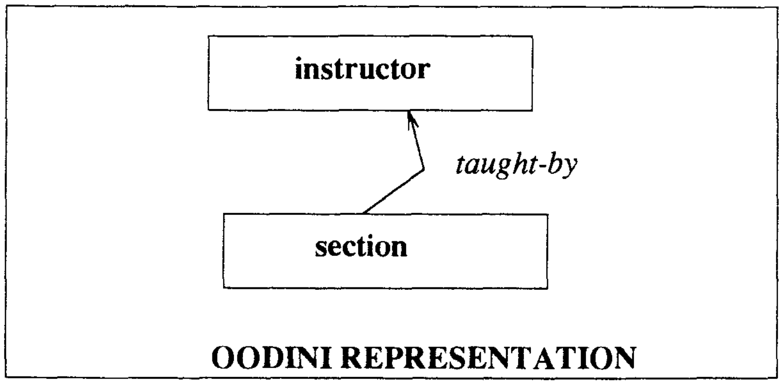

Figure 11. OODINI Ordinary Relationship Graphical Notation

objecttype sectionType	**class** section
	objecttype : sectionType;
relationships	**relationships**
taught-by : instructorType;	taught-by : instructor;
endrelationships;	**endrelationships;**
end;	**end;**

3.2.9 Essential Relationships

Essential relationships are incorporated both into the **<class-definition>** and the **<objecttype-definition>** bodies. This is illustrated below:

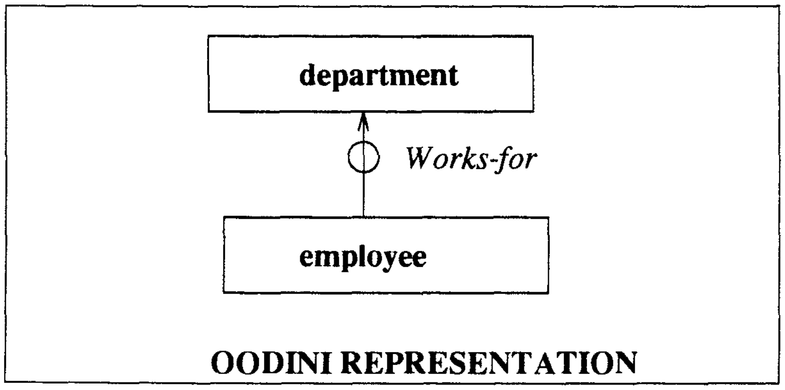

Figure 12. OODINI Essential Relationship Graphical Notation

objecttype employeeType	**class** employee
	objecttype : employeeType;
relationships	**relationships**
Works-for : departmentType;	Works-for :+ department;
endrelationships;	**endrelationships;**
end;	**end;**

☞ NOTE: The :+ separator is used to distinguish essential relationships from ordinary relationships in the **<class-definition>** body. Since essentiality is a semantic property such a separator is *not* reflected in the corresponding **<objecttype-definition>** body.

3.2.10 Multi-valued Relationships

Multi-valued relationships are incorporated both into the **<class-definition>** and the **<objecttype-definition>** bodies. This is illustrated below:

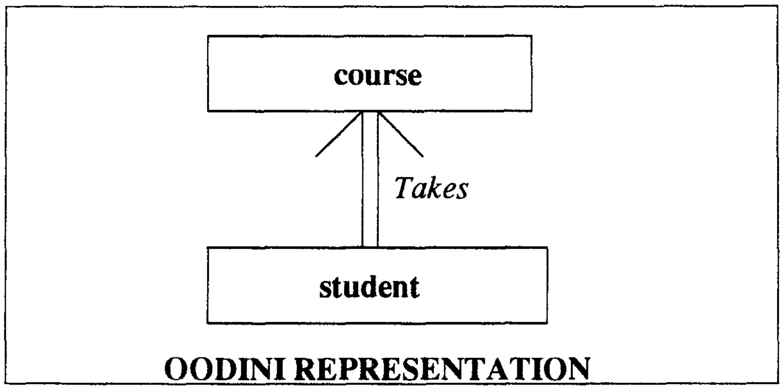

Figure 13. OODINI Multi-valued Relationship Graphical Notation

objecttype studentType	**class** student
	objecttype : studentType;
relationships	**relationships**
Takes :: courseType;	Takes :: course;
endrelationships;	**endrelationships;**
end;	**end;**

☞ NOTE: The :: separator is used to distinguish multi-valued relationships from other relationships.

3.2.11 Dependent Relationships

Dependent relationships are also incorporated both into the **<class-definition>** and the **<objecttype-definition>** bodies. This is illustrated below:

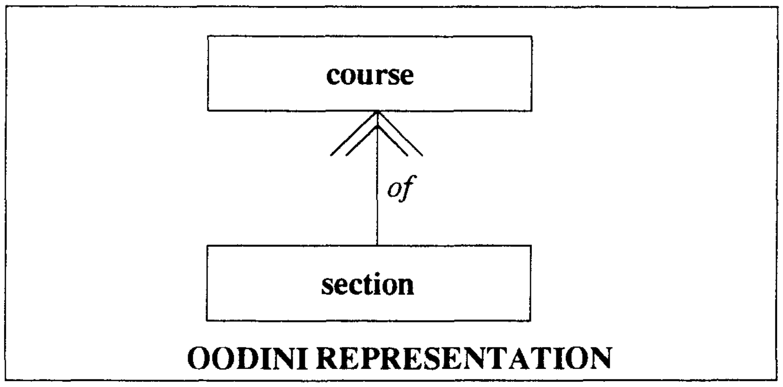

Figure 14. OODINI Dependent Relationship Graphical Notation

objecttype	sectionType	**class**	section
			objecttype : sectionType;
	relationships		**relationships**
	of : courseType;		of :> course;
	endrelationships;		**endrelationships;**
end;		**end;**	

☞ **NOTE:** The :> separator is used to distinguish dependent relationships from other relationships in the **<class-definition>** body. Since dependent relationships convey semantic information this separator is **not** reflected in the **<objecttype-definition>** body.

3.2.12 Multi-valued Essential Relationships

Multi-valued essential relationships are incorporated both into the **<class-definition>** and the **<objecttype-definition>** bodies. This is illustrated below:

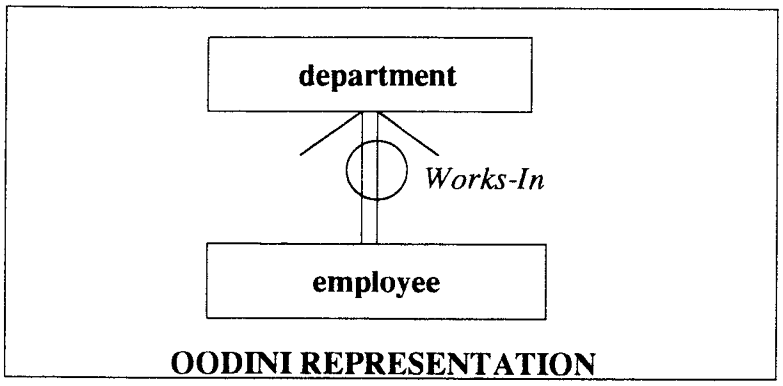

Figure 15. OODINI Multi-valued Essential Relationship Graphical Notation

objecttype employeeType
 	
relationships
Works-In :: departmentType;
endrelationships;
 	
end;
	class employee
	
	objecttype : employeeType;
	
	relationships
	Works-In ::+ department;
	endrelationships;
	
	end;

☞ NOTE: The ::+ separator is used to distinguish multi-valued essential relationships from other relationships in the **<class-definition>** body. In the **<objecttype-definition>** body the multi-valued aspect of the relationship is denoted by the :: separator since essentiality is a semantic concept.

3.2.13 Multi-valued Dependent Relationships

Multi-valued dependent relationships are incorporated both into the **<class-definition>** and **<objecttype-definition>** bodies. This is illustrated below:

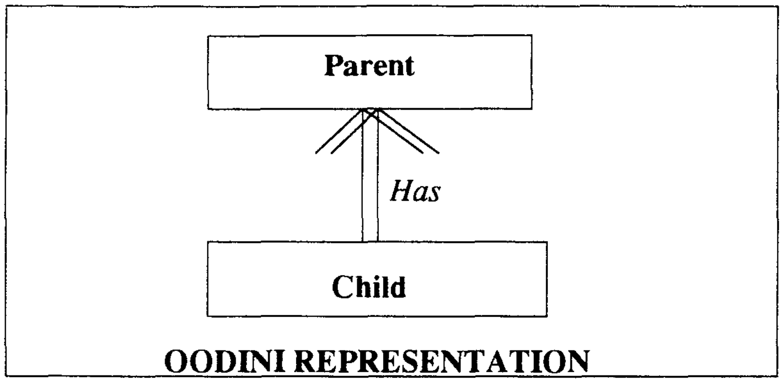

Figure 16. OODINI Multi-valued Dependent Relationship Graphical Notation

objecttype	ChildType	**class**	Child
			objecttype : ChildType;
	relationships		**relationships**
	Has :: ParentType;		has ::> Parent;
	endrelationships;		**endrelationships;**
end;		**end;**	

☞ **NOTE:** The ::> separator is used to distinguish multi-valued dependent relationships from other relationships in the <**class-definition**> body. The :: separator is used in the <**objecttype-definition**> body to convey only the multi-valued aspect of the relationship since dependency is a semantic concept.

3.2.14 Methods

For each object class, only the names of all the methods are listed within the corresponding object type's <**objecttype-definition**> body. Methods are implemented in **OODINI** as **derived attributes**. Pictorially, they are depicted as *dashed* ellipses, very much like attributes. The following syntactical template is generated in **DAL** for methods:

```
objecttype    <objecttype-name>

              methods
                                    method-name-1();
                                    method-name-2();
              endmethods;
end;
```

3.3 Syntax For DAL

The syntax for **DAL** is simple and concise. Most of the syntax has followed the guidelines outlined in [1] and [4]. The code generated in the file is *line-oriented.* Every line can be individually read and easily parsed. Every line contains a **single** syntactical definition, e.g, relation, relationships, class name, object type name, etc. The following set of **keywords** are valid for **DAL**:

class	**end**	**objecttype**	**relationships**
endrelationships	**methods**	**endmethods**	**attributes**
endattributes	**memberof**	**setof**	**categoryof**
subtypeof	**roleof**	**partof**	**unknown_type**
tupleof			

APPENDIX B gives a complete description of the **DAL** language in **BNF** format.

3.4 User API For DAL

The **BNF** specification of **DAL** provides programmers a framework for converting **DAL** into other object oriented programming languages. As mentioned before this entails knowledge of token generation, parsing and syntax validation. Depending on the programming environment this might be a tedious, if not difficult, task. As such we provide a description of a user application programming interface (**API**) that allows programmers to access the internal data structure layout of **DAL**. This **API** is provided in the form of a library routine, **dallib.a**. A single invocation to the library routine, *dal()*, returns two pointer values: (i) a pointer to the list of existing object types, and (ii) a pointer to the list of existing object classes. A complete description of the *dal()* routine is given in standard **UNIX/C** manual page form in **APPENDIX D**.

Associated with this library is also a header file, **dal.h**. This header file describes the internal data structure layout for using the **API**. **APPENDIX C** lists the contents of this header file. Understanding this header file is **critical** to programmers generating code for object oriented languages.

Briefly, we describe in this header file **3** records or **C** language structures and their relationships to each other. The first structure is the **basestruct** record that represents different types of relations, relationships, subtypes, attributes, etc. This structure contains pointers to other classes or object types. This record is analogous to *arcs* connecting two graphical objects along with relevant informations. The next structure represents an **objecttype**. This structure contains the object type name, a pointer to the class it was derived from, along with pointers for attribute lists, member-of, set-of and subtype-of relation list pointers. Finally, the **oclass** structure contains relevant informations about an object class. This includes the class name, a pointer to its object type, pointers for all essential attribute lists, role-of, category-of, part-of, tuple-of as well as all other user defined relationship pointers.

The following diagram depicts the relationships between these 3 structures for a simple graphical model provided by **OODINI**.

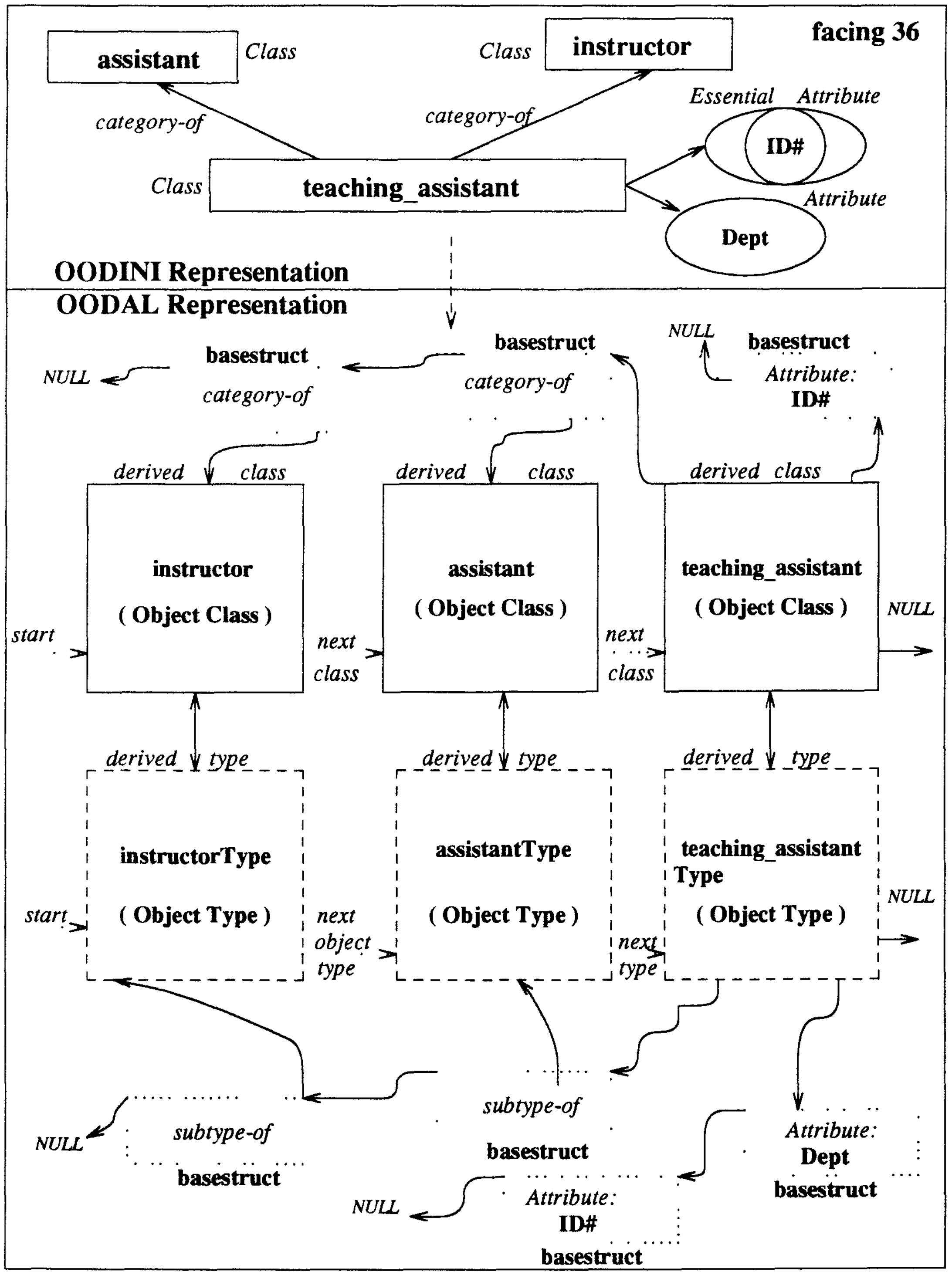

Figure 17. DAL User API Data Structures

3.5 Implementation Of Structural And Semantic Hierarchy

According to the dual model architecture the structural hierarchy for object types (defined by the **subtypeof**, **setof** and **memberof** connections between object types) and the semantic hierarchy (defined by the **roleof**, **categoryof** and **partof** connections between object classes) must be in the form of Directed Acyclic Graphs (**DAG**s). This hierarchy constraint can be maintained by the user implementing a schema under the **OODINI** software.

This constraint is sometimes necessary for generating code. Some languages might have the restriction that **an object type must be explicitly declared prior to its reference.** Similarly, a super class must be declared prior to its subclass declaration. It should be noted that this is a restriction of a programming language and not the dual model. Thus we sometimes have to take this restriction into account prior to code generation from **DAL** to other object oriented languages.

It should be obvious that if we **sort** the **DAL** internal data structure lists for object types and object classes - we can achieve our goal. Sorting in this context implies associating with each object type or class a **level number**. This number is analogous to the level of occurrence of an object type or class in its **DAG** representation. Top level classes or object types in the **DAG** gets lower level numbers. The leaf **nodes** (object types or object classes) in the **DAG** gets higher level numbers. The lists of object classes and object types are then *sorted* based on the level number. This is done prior to code generation from **DAL** to an object oriented database language. Sorting nodes in a network to generate level numbers can easily be done using a **depth first search** algorithm which produces an algorithm of order **O(N)**.

3.6 VML Code Generation

In this section we provide algorithms for converting the **DAL** representation of a dual model database schema to **VML**. Object types and object classes are converted to their equivalent representation in **VML**. *It should be noted that sharing of object types by object classes are still under investigation.* We describe our algorithmic steps using examples.

3.6.1 Conversion Of Object Types

We provide the following object type description under **DAL** and convert it to its equivalent representation under **VML**:

```
objecttype  oType

            subtypeof :     aType;
            subtypeof :     bType;

            setof :         cType;
            memberof :      dType;

            attributes
            abc :           unknown_type;
            xyz :           unknown_type;
            endattributes;

            relationships
            is :            dType;
            of :+           eType;
            as :>           fType;
            has ::          gType;
            has-a ::+       hType;
            is-a ::>
            endrelationships;

            methods
                            methodA();
            endmethods;

end;
```

We now present the following algorithm for converting the **DAL** code to its equivalent **VML** representation:

1. **Replace** all occurrences of the **objecttype** keyword with **OBJECTTYPE**; the **end** keyword with **END** and the **attributes** keyword with **PROPERTIES**. Also **replace** the multi-valued relationship syntactical structure:

   <relationshipname> :: <objecttype>

   with

   <relationshipname> : { <objecttype> }

   Also replace the "::" separator in all multi-valued relationships with the ":" separator. Also remove essentiality and dependent separators from all relationships. The *new* **VML** representation is shown below:

```
OBJECTTYPE    oType

              subtypeof :         aType;
              subtypeof :         bType;

              setof :             cType;
              memberof :          dType;

              PROPERTIES
              abc :               unknown_type;
              xyz :               unknown_type;
              endattributes;

              relationships
              is :                dType;
              of :                eType;
              as :                fType;
              has :               { gType } ;
              has-a :             { hType } ;
              is-a :              { iType } ;
              endrelationships;

              methods
                                  methodA();
              endmethods;
END;
```

2. **Delete** all occurrences of the keywords **endattributes**, **endrelationships** and **relationships**. Also **delete** the **partof** and **tupleof** syntactical templates since **VML** do not have equivalent representation for these. For the **setof** template *create* a *new* multi-valued relationship called **setof**. Translate the **memberof**

relation into a relationship. For multiple **memberof** relations we also create *new* variables with an unique suffix identifier, e.g., **memberof1**, **memberof2**, ..., **memberofn**. The *new* **VML** code is given below:

```
OBJECTTYPE   oType

             subtypeof :      aType;
             subtypeof :      bType;

             PROPERTIES
             abc :            unknown_type;
             xyz :            unknown_type;

             setof :          { cType } ;
             memberof :       dType ;
             is :             dType;
             of :             eType;
             as :             fType;
             has :            { gType } ;
             has-a :          { hType } ;
             is-a :           { iType } ;

             methods
                              methodA();
             endmethods;
END;
```

3. For each **subtypeof : <objecttype>** syntactical template **delete** the occurrences and **collapse** all the **subtypes** in the header declaration "**OBJECTTYPE <objectname> SUBTYPEOF <type-1> <type-2> ...**". If the **subtypeof** relation do not exists for an object type, then we use the predefined **VML** keywords: **SUBTYPEOF MetaClass_InstType**. The *new* **VML** code generated is as follows:

```
OBJECTTYPE    oType              SUBTYPEOF aType, bType

              PROPERTIES
              abc :              unknown_type;
              xyz :              unknown_type;

              setof :            { cType } ;
              memberof :         dType ;
              is :               dType;
              of :               eType;
              as :               fType;
              has :              { gType } ;
              has-a :            { hType } ;
              is-a :             { iType } ;

              methods
                                 methodA();
              endmethods;

END;
```

4. For each relationship create **parametrized** object types in the **OBJECTTYPE** **<objectname>** header declaration. Note that under our **DAL** representation each objecttype **abcType** has a corresponding class, literally written as **abc**. Then each parameter takes on the form of : "**abcClass : abcType**", where we added the suffix "**Class**" to the original class name **abc**. Also in each relationship declaration replace the **<objecttype>** syntactical form by its corresponding parameter name. Create a parameter for the object type itself (in this case create the parameter **oclass** that refers to the objecttype **oType**). Also if the object type is a **subtype** of other object types, then each parameter of the super types must be *suffixed* in the parameter list of this subtype. For example, in this case **oType** has 2 super types, **aType** and **bType**. Assume that the parameter list for **aType** consists only of "**aClass : aType**" and that for **bType** is "**bClass : bType**". Then incorporate these two parameters in the parameter list for **oType**. The corresponding **VML** code for a **DAL** object type representation is as follows:

```
OBJECTTYPE   oType
                            [ aClass : aType, bClass : bType, oClass : oType,
                              cClass : cType, dClass : dType, eClass : eType,
                              fClass : fType, gClass : gType, hClass : hType,
                              iClass : iType ]
                            SUBTYPEOF aType [ aClass, bClass ], bType

             PROPERTIES
             abc :          unknown_type;
             xyz :          unknown_type;

             setof :        { cClass } ;
             memberof :     dClass ;
             is :           dClass;
             of :           eClass;
             as :           fClass;
             has :          { gClass } ;
             has-a :        { hClass } ;
             is-a :         { iClass } ;

             methods
                            methodA();
             endmethods;
END;
```

5. Finally, for each method defined in the **DAL** object type definition, we add the **VML** keywords **METHODS** and **IMPLEMENTATION**, and change each **DAL** **<method-name>();** syntactic template to its corresponding **VML** template: **<method-name() : READONLY { };**. The formal parameters of the method along with the actual code needs to be added by the user. With these changes the final **VML** code for an object type representation is as follows:

```
OBJECTTYPE    oType
                                [ aClass : aType, bClass : bType, oClass : oType,
                                  cClass : cType, dClass : dType, eClass : eType,
                                  fClass : fType, gClass : gType, hClass : hType,
                                  iClass : iType ]
                                SUBTYPEOF aType [ aClass, bClass ], bType
              PROPERTIES
              abc :             unknown_type;
              xyz :             unknown_type;
              setof :           { cClass } ;
              memberof :        dClass ;
              is :              dClass;
              of :              eClass;
              as :              fClass;
              has :             { gClass } ;
              has-a :           { hClass } ;
              is-a :            { iClass } ;
              IMPLEMENTATION
              METHODS
              methodA() :       READONLY
              {
              };
END;
```

3.6.2 Conversion Of Object Classes

We provide the following object class description under **DAL** and convert it to its equivalent representation under **VML**:

```
class   oclass

        objecttype :        oType;
        roleof :            class-1;
        categoryof :        class-2;

        attributes
        id :                unknown_type;
        endattributes;

        relationships
        is :                d;
        of :                e;
        as :                f;
        has :               g;
        has-a :             h;
        is-a :              i;
        endrelationships;

end;
```

We now present the following algorithm for converting the **DAL** code to its equivalent

VML representation:

1. **Delete** the *entire* **relationships** ... **endrelationships** block. The relations are already listed using parameters in the corresponding object type description block. Also delete the *entire* **attributes ...**

   endattributes block, since the attributes are already listed in the object type for the object class. Furthermore, **VML** has no concept of essentiality of attributes. Making these changes we have a *new* **VML** representation:

```
class   oclass

        objecttype :    oType;
        roleof :        class-1;
        categoryof :    class-2;

end;
```

2. **Replace** the keyword **class** with **CLASS** and the keyword **end;** with **END**. Also replace the syntactical template **objecttype : <objectname>;** with the new syntactical representation **INSTTYPE <objectname>**. Making these changes we have the *new* **VML** format:

```
CLASS   oclass

        INSTTYPE :     oType
        roleof :       class-1;
        categoryof :   class-2;

END;
```

3. **Parametrize** the **INSTTYPE <objectname>** declaration. For example, if the corresponding object type header declaration has the following parameter list:

   OBJECTTYPE otype [AClass : AType, BClass : BType]

   Then after parametrization the class **INSTTYPE**'s declaration is changed as follows:

   INSTTYPE otype [A, B]

   ☞ **NOTE:** This is possible because under the **DAL** representation if a class name is **A**, then its corresponding object type name is **AType**, and when used as a parameter it is **AClass**, literally. Making these changes we have the following **VML** view:

```
CLASS   oclass

        INSTTYPE :     oType [ a, b, o, c, d, e, f, g, h, i ];
        roleof :       class-1;
        categoryof :   class-2;

END
```

4. If there exists a **roleof** and/or **category** of declaration within a class definition we have to add the following syntactical template in the class declaration header statement of the form:

   CLASS <class-name> **METACLASS** <semantic-keyword>

   This definition must be used in both the **specialized** and the **generalized** classes. In our example, the specialized class is **otype** while the classes being generalized are **class-1** and **class-2**. The **<semantic-keyword>** to be used depends on what *combination* of semantic relations applies to the class. For example, the class could be a specialized class (via a **roleof** connection) and at the same time a generalized class (via a **categoryof** connection), etc. However, there are only some distinct possibilities and the semantic keyword for each possibility is depicted in the following figure.

Roles played by a class				
ROLE GENERAL-IZATION	**ROLE SPECIAL-IZATION**	**CATEGORY GENERAL-IZATION**	**CATEGORY SPECIAL-IZATION**	*<semantic-keyword> to use*
			✓	**CATEGORY-SPECIALIZATION-CLASS**
		✓		**CATEGORY-GENERALIZATION-CLASS**
		✓	✓	**C-GEN-C-SPEC-CLASS**
	✓			**ROLE-SPECIALIZATION-CLASS**
	✓		✓	**R-SPEC-C-SPEC-CLASS**
	✓	✓		**R-SPEC-C-GEN-CLASS**
	✓	✓	✓	**R-SPEC-C-GEN-C-SPEC-CLASS**
✓				**ROLE-GENERALIZATION-CLASS**
✓			✓	**R-GEN-C-SPEC-CLASS**
✓		✓		**R-GEN-C-GEN-CLASS**
✓		✓	✓	**R-GEN-C-GEN-C-SPEC-CLASS**
✓	✓			**R-GEN-R-SPEC-CLASS**
✓	✓		✓	**R-GEN-R-SPEC-C-SPEC-CLASS**
✓	✓	✓		**R-GEN-R-SPEC-C-GEN-CLASS**
✓	✓	✓	✓	**R-GEN-R-SPEC-C-GEN-C-SPEC-CLASS**

Figure 18. VML Semantic MetaClass Keywords

Using the previous table we find out the **<semantic-keyword>** to use in the class declaration header. In our example, the object class **oclass** has both a **roleof** and a **category** of semantic connection. Hence the **<semantic-keyword>** to use is **R-SPEC-C-SPEC-CLASS** - as this class plays a *specialized* role in both the semantic relations. Finally, for role specialized class **VML** needs the following syntactical definition:

INIT : SELF->defRoleOf({ *<role-generalized-class-name>* });

Similarly, for category specialized classes **VML** needs the following definition:

INIT : SELF->defCategoryOf({ *<category-generalized-class-name>* });

The final **VML** object class representation is then as follows:

```
CLASS   oclass          METACLASS R-SPEC-C-SPEC-CLASS

        INSTTYPE :      otype [ d, e, f, g, h, i ]

        INIT:           SELF->defRoleOf( { class-1 } );
                        SELF->defCategoryOf( { class-2 } );

END
```

☞ Note that the **INIT** clause is used to invoke the predefined method(s), e.g., **defRoleof** and/or **defCategoryOf** of the metaclass, e.g., **R-SPEC-C-SPEC-CLASS**. Also note that the actual argument in these method calls are treated as a list of classes (by using braces around the arguments). This is how semantic relations, e.g., **roleof** and **categoryof**, between a single specialized class and *multiple* generalized classes are handled. More specifically, for example, if the object class **oclass** has multiple **roleof** semantic relations to object classes **class-1**, **class-2**, ..., **class-n**, then we would have generated the following syntactical format:

SELF->defRoleOf({ class-1, class-2, ..., class-n });

☞ Note that in the semantic relation **roleof** the object class **class-1** plays a more *generalized* role. Hence this object class must have a header declaration of the following format:

CLASS class-1 **METACLASS** ROLE-GENERALIZATION-CLASS

☞ Similarly the object class **class-2** is the more *generalized* class with regards to the **categoryof** semantic connection. This object class must also then have a header declaration of a similar format:

CLASS class-2 **METACLASS** CATEGORY-GENERALIZATION-CLASS

3.7 Implementation Of OODAL

Converting a **OODINI** graphical schema to **DAL** code often requires an understanding of the Dual model architecture. Most object oriented database languages do not adhere to this architecture. As such the **DAL** code generated might impose difficulty for developers generating object oriented database code. For this purpose, we propose a *new* and *different abstract* language, **OODAL** (OODINI Abstract Language), that removes any dependencies on the dual model architecture. However to maintain consistency between these abstract languages, the **OODAL** syntax closely resembles that described for **DAL**. **OODAL** is almost identical to **DAL**, except that the **<object-type definition>** is completely removed from this language. Only code for object classes are generated. All necessary information for an object class, e.g., relations, relationships, attributes and methods, are represented entirely in the **<object-class definition>** syntactical template. Unlike **DAL**, in **OODAL** there is no reference to a corresponding object type from an object class.

3.7.1 Implementation Of Object Classes

For each **OODINI** object class, **OODAL** generates the following syntactical template:

```
class    <class-name>

         <class-definition>
end;
```

3.7.2 Implementation Of Attributes

The attributes for an object class are listed within the **<class-definition>** body. Since an attribute's data type or object type information is not represented under **OODINI**, the keyword **unknown_type** is used as a place holder. The attributes are listed using the following syntactical template:

```
class    <class-name>

         attributes
                           <attribute-name> : unknown_type;
                           <essential-attribute-name> :+ unknown_type;
         endattributes;
end;
```

☞ **NOTE:** The usage of the :+ separator to distinguish essential attribute declaration from non-essential ones.

3.7.3 Implementation Of Relations

All relations for an object class are listed within the **<class-definition>** body. The **setof, memberof, categoryof, roleof, partof** and **tupleof** relations are generated using the following syntactical template:

```
class   <class-name>

        setof : <class-name>;
        memberof : <comma separated class-name list>;
        categoryof : <comma separated class-name list>;
        roleof : <comma separated class-name list>;
        partof : <comma separated class-name list>;
        tupleof : <comma separated <connector : class-name> list>;
end;
```

3.7.4 Implementation Of Relationships

All relationships for an object class are listed within the **<class-definition>** body. **OODAL** makes a distinction between ordinary, essential, dependent, multi-valued, multi-valued essential and multi-valued dependent relationships by using the same **separators** that were used for **DAL**. More specifically, the **:** separator is used to list ordinary relationships. The **:+** separator is used for essential relationships. The **:>** separator is used to distinguish dependent relationships. The **::** separator is used for multi-valued relationships. The **::+** separator is used for multi-valued essential relationships. The **::>** separator is used to distinguish multi-valued dependent relationships. Relationships are generated using the following syntactical template:

```
class   <class-name>

        relationships
                         ordinary-relationship-name : <class-name>;
                         essential-relationship-name :+ <class-name>;
                         dependent-relationship-name :> <class-name>;
                         multivalued-relationship-name :: <class-name>;
                         multivalued-essential-relationship ::+ <class-name>;
                         multivalued-dependent-relationship ::> <class-name>;
        endrelationships;
end;
```

3.7.5 Implementation Of Methods

All methods for an object class are listed within the **<class-definition>** body. Only method names are generated using the following syntactical template:

```
class   <class-name>

        methods
                     method-name-1();
                     method-name-2();
        endmethods;
end;
```

3.8 Syntax For OODAL

The syntax for **OODAL** closely resembles the one described for **DAL**, except that the **<object-type definition>** is not present in **OODAL**. The code is generated in a flat file and is *line-oriented.* Individual lines can easily be read and parsed for further code generation. The following set of keywords are valid for OODAL:

class	**end**	**relationships**	**endrelationships**
methods	**endmethods**	**attributes**	**endattributes**
memberof	**setof**	**categoryof**	**roleof**
partof	**unknown_type**	**tupleof**	

APPENDIX H gives a complete description of the **OODAL** language in **BNF** format.

3.9 User API For OODAL

The **BNF** description of **OODAL** allows programmers for converting **OODAL** into other object oriented database languages. However this would entail parsing a flat file containing **OODAL** code, often a tedious task. As such, like in **DAL**, we provide a description of an **API** that allows programmers to access the internal data structure layout of **OODAL**. This **API** is provided in the form of a library routine, **oodallib.a**. An invocation to the library routine, *oodal()*, returns a pointer to a list of existing object classes. A complete description of the *oodal()* library routine is given in standard **UNIX/C** manual page form in **APPENDIX J**.

Associated with the library is also a header file, **oodal.h**. This header file describes the internal data structure layout for using the **API**. **APPENDIX I** gives the content of this header file. The data structures described in this header file closely resembles the one described for **DAL**. Like in **DAL**, we describe only **2** records or **C** language structures and their relationships to each other. The first structure is the **basestruct** record which is almost identical to the one described for **DAL**. This structure contains pointers to another structure, the object **class** structure (**oclass**), which also resembles closely the structure described for **DAL**. This **class** structure contains the class name and pointers to list of attributes, relations, relationships and method names.

The following diagram depicts the relationships between these 2 structures for a simple graphical model implemented under **OODINI**.

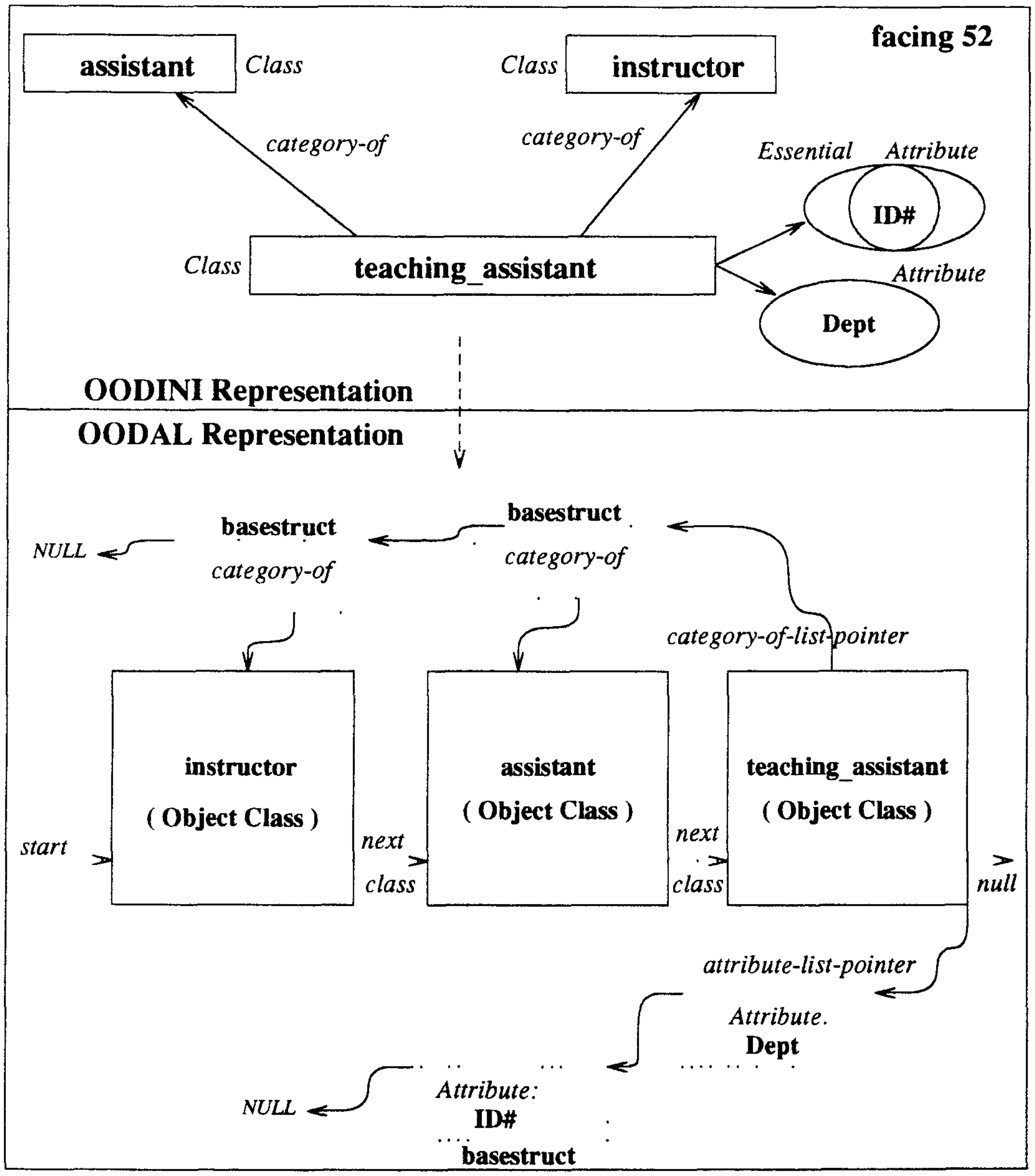

Figure 19. OODAL User API Data Structures

CHAPTER 4
CONCLUSIONS

In this paper we have provided a generic solution for storing graphical images for an object oriented database graphics editor on secondary storage. Our solution achieves dynamic data persistence for the heap of a process's address space. This is done using memory mapped files. Memory for all graphical images are allocated from the heap portion of the virtual address space of the graphics editor. This solution is generic, innovative, reliable, easily programmable, performance conscious and even cost effective in the sense that it removes the necessity for having developers to maintain the software that saves graphical images on secondary storage.

We also present a formal definition of a dual model object oriented abstract language, **DAL**. The graphical image representation on secondary storage is converted to **DAL**. We provide concrete examples of each graphical image representation and its corresponding syntactical form in **DAL**. The entire graphical database schema is converted into object classes and corresponding object types. To facilitate code generation from **DAL** we also provide a user application programming interface.

We also provide an example of code generation into an object oriented database language. The **DAL** code generated for a graphical schema is converted to its equivalent representation for the **VODAK** data modeling language (**VML**). Using concrete algorithms and examples we demonstrate this conversion.

Finally, we present a formal definition of yet another object oriented abstract language, **OODAL**. This language removes dependencies on the dual model architecture. This will help developers to generate object oriented database code without having them to know about the *relatively new* dual model schema representation. It should be noted that both these abstract languages (**DAL** and **OODAL**) will help developers in generating object oriented database code. These developers need not have knowledge about **OODINI**'s complex data structures or operating system concepts, e.g., *mmap()*. Furthermore, by establishing a static syntax for these languages, any future changes to the **OODINI** software will minimize code generation efforts that might affect these developers.

In the future releases of this software we are targeting our efforts on the following specificities:

- Although we collapse object types, in certain cases we still need a mechanism to show that different object classes share the same object type. Two future looking work can be done in this respect. One possibility is to represent this graphically in the graphics editor itself. Another possibility is to use partial/structural integration prior to **DAL** code generation.
- From the graphical schema it is not clearly evident whether user defined relationships are structural or semantic in nature. Structural relationships should only be incorporated into object types, while semantic relationships should only be declared in object classes. Clearly, changes to the graphics editor must be made if this kind of information needs to be extracted. The same concern holds for method definitions.
- Given the **DAL** or **OODAL** representation we would like to generate the graphical schema. This will immensely facilitate schema modifications.

APPENDIX A: MANUAL PAGES: IMAGE PERSISTENCE

This section contains **UNIX** style manual page for the **libmapmalloc.a** library.

NAME

mapmalloc_init - **mapmalloc** initialization operation.

C SYNOPSIS

```
#include <sys/types.h>
#include "mapmalloc.h"

int     _mapmalloc_init(fixed_addr, size, flags, filename, offset)

char    *fixed_addr;
size_t  size;
int     flags;
char    *filename;
off_t   offset;
```

DESCRIPTION

This routine initializes a memory mapped region for a file. This memory mapped region can then be used for dynamic memory allocation and deallocation. Data allocated from this region is always persistent across program execution sessions.

ARGUMENTS

The **fixed_addr** argument specifies the virtual address in the user address space where the file should be mapped from. If the user does not want a fixed mapping, then this argument should be 0.

The **size** argument is the length of the file being mapped. If the file specified by the **filename** argument does not exist, the file is created. In such a case if the **size** argument is 0, the file is created with an initial size of **64 K** bytes. If the file exists and the **size** argument is 0 - then the entire length of the file is mapped. It should be noted that specifying a **size** value other than **0** will not cause the file to be grown when its contents are full (*heap overflow*).

The **flags** argument could be a combination of any one of the following values defined in the **mapmalloc.h** header file:

MAPMALLOC_READ_ONLY - the file is mapped as read-only into the users's address space.

MAPMALLOC_FORCE_INIT - the original contents of the file being mapped are ignored, and the file is re-initialized.

MAPMALLOC_NO_CREATE - No new file is created. The file must exist. If it does not then *mapmalloc_init()* fails.

MAPMALLOC_PRE_ALLOC_DISK - Forces block allocation for the entire file during this initialization process.

No *holes* are left in the file.

MAPMALLOC_NO_PRE_ALLOC_DISK - Not *all* the blocks are allocated for the file during initialization. Only the first and the last blocks are created leaving a *hole* in between.

MAPMALLOC_EXCLUSIVE - Locks the file exclusively for the first process that gains access through the library. If the file is already locked, the invocation fails and **errno** is set to **EAGAIN**. However, if the **MAPMALLOC_WAIT_EXCLUSIVE** flag is set (see below) - the process blocks on the file lock.

MAPMALLOC_LOCK_SHARED - Causes file locking between *cooperating* processes sharing the file during memory allocation and deallocation. This prevents simultaneous updates to the file being mapped when processes share the file.

MAPMALLOC_NO_GROW - When the file contents are full and no further memory can be allocated (*heap overflow*) - usage of this flag prevents the file from growing.

MAPMALLOC_WAIT_EXCLUSIVE - If the file has already been locked by another process, this flag causes the process to block on the lock.

MAPMALLOC_MK_MMAP_FILE - Tells this initialization routine to create an *mmap()* based memory region.

MAPMALLOC_MK_SHM_FILE - Tells this initialization routine to create a shared memory region instead of a *mmap()* based file. If this flag is used along with **MAPMALLOC_MK_MMAP_FILE** then this flag takes precedence.

MAPMALLOC_BEST_FIT - During memory allocation a *best-fit* strategy is to be used.

MAPMALLOC_FIRST_FIT - During memory allocation a *first-fit* strategy is to be used.

FREE_COALESCE - During memory deallocation coalesce as many free blocks as possible. Coalescing is done only on the free blocks occurring on the *left* side of the memory block being deallocated.

FREE_COALESCE_RIGHT - During memory deallocation coalesce as many free blocks as possible. Coalescing is done only on the free blocks occurring on the *right* side of the memory block being deallocated.

FREE_NO_COALESCE - During memory deallocation do not do any coalescing.

0 - If a value of 0 is passed in for this argument, then by default, the following flag value combinations are *or*-ed in: **FREE_NO_COALESCE, MAPMALLOC_PRE_ALLOC_DISK,** and **MAPMALLOC_MK_SHM_FILE.**

The **filename** argument specifies the name of the file being mapped into memory. If the file does not exist on secondary storage, the file is created. If the file exists the file is simply opened; the contents of the file are not changed.

The **offset** argument specifies the offset into the file from which the mapping should be started. The offset value should be greater than or equal to **0** and less than or equal to the entire length of the file being mapped.

Upon invocation this routine initializes a memory mapped region based on the file named by the argument **filename**. The file is assumed not yet to be opened. If the named file already exists, then the contents of the file are not initialized.

RESULTS

On success **mapmalloc_init**() returns the virtual address where the file is mapped by the operating system. On failure it returns a value of **(char *) -1**.

EXAMPLE

Sample code to initialize a memory mapped file:

```
char    *vaddr;

/* Map a 1 Megabyte file at logical file offset 0 */

if (mapmalloc_init(0, 1024 * 1024, 0, "/tmp/heapfile", 0L) ==
(char *) -1) {

        /* init failed -- execute your error processing code */
}
```

WARNINGS

If shared memory region is being used then it should be noted that the file corresponding to the shared memory cannot be *grown* if the file contents are *full*. Also in such cases, the operating system does not automatically flush the pages for the shared memory to the file. The user needs to invoke **mapregion_sync**() prior to process termination - to force memory contents to be synced back to secondary storage.

SEE ALSO

mapmalloc(), mapfree(), map_set_base_user_addr(), map_get_base_user_addr(), mapregion_sync(), mapclose().

NAME

mapmalloc - Memory allocation routine.

C SYNOPSIS

```
#include <sys/types.h>
#include "mapmalloc.h"

char *  mapmalloc(region_addr, size, strategy)

char *  mapcalloc(region_addr, nelem, size, strategy)

char *  maprealloc(region_addr, nelem, newsize, strategy)

char *  MALLOC(size)

char *  CALLOC(nelem, size)

char *  REALLOC(addr, size)


char    *region_addr;
size_t  size;
int     strategy;
size_t  nelem;
size_t  newsize;
char    *addr;
```

DESCRIPTION

All of the above routines provide a simple general-purpose memory allocation package based on memory mapped files or shared memory regions.

ARGUMENTS

The **region_addr** argument is *one of the* memory mapped regions in the user address space. A user can map as many files as possible (limited by operating system restrictions) into its address space. Memory is then allocated from the region specified by address returned during the *mapmalloc_init()* invocation. If this value is 0 then memory allocation is done on the *first* region that is memory mapped in the user's address space.

The **size** argument is a non-negative integer value greater than 0 specifying the number of bytes to allocate.

The **strategy** argument can take in two values, **MALLOC_BEST_FIT** and **MALLOC_FIRST_FIT**. If **MALLOC_BEST_FIT** is used then a *best-fit* search strategy is used to find a piece of available memory. If **MALLOC_FIRST_FIT** is used then a *first-fit* search strategy is used to find a piece of available memory. If this value is 0 then it defaults to **MALLOC_BEST_FIT**.

The **nelem** argument is a value greater than 0 specifying that **n** elements of size **size** are to be allocated contiguously in the virtual address space.

The **newsize** argument is a value greater than 0 specifying the *new* number of bytes to be allocated.

The **addr** argument value is an address that was returned from a previous invocation to any one of these routines.

mapmalloc() allocates the requested number of bytes using the strategy option (*best/first fit*) requested by the user from the memory region corresponding to the file that was mapped.

maprealloc() changes the size of the previously allocated memory block pointed to by *addr* to *newsize* bytes and returns a pointer to the (possibly moved) memory block. The contents will be unchanged up to the lesser of the old and *newsize* values. If no free block of size bytes is available then **maprealloc**() will ask **mapmalloc**() to enlarge the arena by *newsize* bytes and will then move the data to the new space.

mapcalloc() allocates space for an array of *nelem* elements of size *size*. The space is initialized to zeroes.

MALLOC() invokes **mapmalloc**() with the strategy argument set to **MALLOC_BEST_FIT**. The *region_addr* value passed in is 0. Hence memory allocation is done from the first region mapped in.

REALLOC() invokes *maprealloc()* with *region_addr* set to 0 and strategy set to **MALLOC_BEST_FIT**.

CALLOC() invokes *mapcalloc()* with *region_addr* set to 0 and strategy set to **MALLOC_BEST_FIT**.

RESULTS

On success all of the memory allocation routines return a valid virtual address of the memory block(s) allocated. On failure a value of **(char *) NULL** is returned.

It should also be noted that when the contents of the file from which memory allocation takes place are full (*heap overflow*) - the file is automatically grown by the library. However, if a **size** value other than **0** was specified during the *mapmalloc_init()* call, the file is not grown - and memory allocation will fail. Note that except for the super user a file cannot be grown beyond the **ulimit** value specified for a user.

Another interesting case occurs when the **MAPMALLOC_NO_PRE_ALLOC_DISK** flag is used in the *mapmalloc_init()* call. In such a case, file block allocation can be done by the operating system when the library attempts to allocate memory for which file blocks never existed before. It just might happen that during this memory allocation, the operating system runs out of available file system blocks or a user quota is exceeded. In such a case the kernel sends a **SIGBUS** signal to that process - the effect will be an unfortunate **core dump**. To avoid this from happening the memory allocation routines performs appropriate signal handling (by doing *setjmp()/longjmp()*) and does return back a **NULL** pointer to the user.

EXAMPLE

Sample code to allocate 100 bytes of memory.

```
char    *vaddr;

if ((vaddr = MALLOC(100)) == (char *) NULL) {

        /* Error processing code */
}
```

WARNINGS

Unpredictable results may occur if the user corrupts the heap space that is being memory mapped.

SEE ALSO

mapmalloc_init(), mapfree(), map_set_base_user_addr(), map_get_base_user_addr(), mapregion_sync(), mapclose().

NAME

mapfree - Memory deallocation routine.

C SYNOPSIS

```
#include <sys/types.h>
#include "mapmalloc.h"

void    mapfree(addr, strategy)

void    FREE(addr)

char    *addr;
int     strategy;
```

DESCRIPTION

mapfree() deallocates the memory associated with the value of *addr*.

ARGUMENTS

The *addr* argument must be a valid pointer obtained from a previous invocation to any one of the memory allocation routines.

The *strategy* argument values can be any one of the following:

> **FREE_COALESCE** - Here coalescing is attempted on all of the free blocks that are *left* adjacent to the memory block being freed.
>
> **FREE_COALESCE_RIGHT** - Here coalescing is attempted on all of the free blocks that are *right* adjacent to the memory block being freed.
>
> **FREE_NO_COALESCE** - Here no coalescing of existing free memory blocks is attempted when a memory block is being deallocated. This is the default value used.

Upon invocation the previously allocated block of memory is freed to the appropriate region that it was allocated from. The **FREE**() *macro* invokes **mapfree**() with **FREE_NO_COALESCE** flag.

RESULTS

On success the memory block is deallocated and returned to the free buffer pool. If the address specified is an invalid one - no action is taken.

WARNINGS

Undefined results, e.g. heap corruption, will occur if an allocated freed memory block is freed a second time.

SEE ALSO

mapmalloc_init(), mapmalloc(), map_set_base_user_addr(), map_get_base_user_addr(), mapregion_sync(), mapclose().

NAME

map_set_base_user_addr/map_get_base-user_addr - Save/retrieve pointer value.

C SYNOPSIS

```
#include <sys/types.h>
#include "mapmalloc.h"

void    map_set_base_user_addr(region_addr, user_addr)

char *  map_get_base_user_addr(region_addr)


char    *region_addr;
char    *user_addr;
```

DESCRIPTION

These routines allows programmers to store and retrieve in the header section of a given memory mapped region a pointer value usually pointing to the base user information. Such a pointer value usually reflects the **root** pointer value for user data structures, so that between each program execution sessions the user will have a known point of reference for the data structures it is using.

ARGUMENTS

The **region-addr** argument is the starting address of the memory mapped region.

The *user_addr* is a pointer variable value that is provided by the user.

RESULTS

map_set_base_user_addr() simply stores a pointer value without validating the value. **map_get_base_user_addr**() returns a pointer variable value to the user. This value will be **0** if the user did not previously do a **map_set_base_user_addr**().

SEE ALSO

mapmalloc_init(), **mapmalloc**(), **mapfree**(), **mapregion_sync**(), **mapclose**().

NAME

mapregion_sync - Sync back memory pages for memory mapped region to file.

C SYNOPSIS

```
#include <sys/types.h>
#include "mapmalloc.h"

int     mapregion_sync(region_addr, flags)

char    *region_addr;
int     flags;
```

DESCRIPTION

This routine forces the contents of an entire memory mapped region to be written out to its corresponding file storage. This routine **must** be used prior to process termination if shared memory region has been selected during *mapmalloc_init()* instead of *mmap()*.

ARGUMENTS

The *region_addr* argument is the virtual address where the memory mapped region is attached to the user's address space.

The *flag* argument could take in one of the following values:

MS_SYNC - Perform synchronous writes to disk. For shared memory region based heaps this flag is the only option available.

MS_ASYNC - Perform asynchronous writes to disk. This flag can only be used when *mmap()* has been used to memory map the file.

If a flag value of **0** is used then it defaults to **MS_SYNC**.

RESULTS

On success **mapregion_sync**() returns a value of 0. On failure it returns the value of -1.

SEE ALSO

mapmalloc_init(), **mapmalloc**(), **mapfree**(), **map_set_base_user_addr**(), **map_get_base_user_addr**(), **mapclose**().

NAME

mapclose - Unmap entire memory mapped region for a file.

C SYNOPSIS

```
#include <sys/types.h>
#include "mapmalloc.h"

int     mapclose(region_addr)

char    *region_addr;
```

DESCRIPTION

This routine simply unmaps and closes the file associated with **region_addr**. The address space that was previously memory mapped is now inaccessible.

ARGUMENTS

The *region_addr* argument is the virtual address where the memory mapped region is attached to the user's address space.

RESULTS

On success **mapclose**() returns a value of 0. On failure it returns the value of -1.

SEE ALSO

mapmalloc_init(), mapmalloc(), mapfree(), map_set_base_user_addr(), map_get_base_user_addr(), mapregion_sync().

APPENDIX B: BNF DESCRIPTION FOR DAL

This section gives the **BNF** description of the **DAL** language.

<Start> --> <objecttype> <class>

<objecttype> --> <objectdefinition> <objecttype> | <objectdefinition>

<objectdefinition> --> **objecttype** <objectname> <objectbody> **end;**

<objectname> --> <classname>**Type**

<objectbody> --> <subtypeof> <setof> <memberof> <opartof> <otupleof> <attributes>
<orelationships> <methods>

<subtypeof> --> **subtypeof** : <objectname-list> ; | <NULL>

<setof> --> **setof** : <objectname-list> ; | <NULL>

<memberof> --> **memberof** : <objectname-list> ; | <NULL>

<opartof> --> **partof** : <objectname-list> ; | <NULL>

<otupleof> --> <otuple> ; <otupleof> | <otuple> ; | <NULL>
<otuple> --> **tupleof** : < oconnector-list> ;
<oconnector-list> --> < <connector-name> : <objectname> <more-oconnector>
<more-oconnector> --> > | , <connector-name> : <objectname> <more-oconnector>

<attributes> --> **attributes** <attrs> **endattributes;** | <NULL>
<attrs> --> <attr> ; <attrs> | <attr> ;
<attr> --> <attribute-name> : **unknown_type**

<orelationships> --> **relationships** <orelationbody> **endrelationships;** | <NULL>

<orelationbody> --> <relationshipname> <typeoforelation> <objectname> ;
<orelationbody> | <NULL>

<typeoforelation> --> : | ::

<methods> --> **methods** <method-body> **endmethods;** | <NULL>
<method-body> --> <method-name> (); <method-body> | <method-name> ();

<class> --> <classbody> <class> | <classbody>

<classbody> --> **class** <classname> <classdefinition> **end** ;

<classdefinition> --> <relations> <eattributes> <relationships> <methods>

<relations> --> <classtype> <categoryof> <roleof> <partof> <tupleof>

<classtype> --> **objecttype** : <objectname> ;

<categoryof> --> **categoryof** : <class-name-list> ; | <NULL>

<roleof> --> **roleof** : <class-name-list> ; | <NULL>

<partof> --> **partof** : <class-name-list> ; | <NULL>

<tupleof> -> **tupleof** : <connect-list> ; <tupleof> | <NULL>
<connect-list> --> < <connector-name> : <classname> <more-connect>
<more-connect> --> > | , <connector-name> : <classname> <more-connect>

<eattributes> --> **attributes** <eattrs> **endattributes;** | <NULL>
<eattrs> --> <attributename> ; <eattrs> | <attributename> ;

<relationships> --> **relationships** <relationbody> **endrelationships;** | <NULL>

<relationbody> --> <relationshipname> <typeofrelation> <classname>
<relationbody> ; |
<relationshipname> <typeofrelation> <classname> ;
<relationbody>

<typeofrelation> --> : | :: | :+ | :> | ::+ | ::>

<classname> --> *Character string constituting a class name*

<class-name-list> --> <classname> , <class-name-list> | <classname>

<objectname-list> --> <objectname> , <objectname-list> | <objectname>

<attributename> --> *Character string constituting an attribute name*

<relationshipname> --> *Character string constituting a relationship name*

<connector-name> --> *Character string, a connector name in a tuple-of relation*

<method-name> --> *Character string constituting a method name*

<NULL> --> ""

APPENDIX C: USER API HEADER FILE FOR DAL

This section lists the contents of the user **API** header file, **dal.h**.

```
typedef struct basestruct {

        struct basestruct    *next;                          /* Next list element pointer */

        int                  flag;                           /* What kind of arc */
        char                 *name;                          /* Name field */

        union {                                              /* Variant */

                             struct oclass  *classp;         /* Class pointer */
                             struct objecttype  *objectp;    /* or objecttype pointer */
        } un;

        char                 *foruser[2];                    /* 2 private slots for users */
        char                 *future[4];                     /* Future expansion slots */

} basestruct_t;


typedef   basestruct_t   orelation_t;     /* Ordinary relationships */
typedef   basestruct_t   erelation_t;     /* Essential relationships */
typedef   basestruct_t   drelation_t;     /* Dependent relationships */
typedef   basestruct_t   mvrelation_t;    /* Multi-valued relationships */
typedef   basestruct_t   mverelation_t;   /* Multi-valued Essential */
typedef   basestruct_t   mvdrealtion_t;   /* Multi-valued Dependent */
typedef   basestruct_t   roleof_t;        /* Role-of connection */
typedef   basestruct_t   categoryof_t;    /* Category-of connection */
typedef   basestruct_t   partof_t;        /* Part-of connection */
typedef   basestruct_t   setof_t;         /* Set-of connection */
typedef   basestruct_t   memberof_t;      /* Member-of connection */
typedef   basestruct_t   tupleof_t;       /* Tuple-of connection */
typedef   basestruct_t   subtypeof_t;     /* Subtype-of connection */
typedef   basestruct_t   attributeof_t;   /* Attribute list */
typedef   basestruct_t   essential_t;     /* Essential attribute list */
typedef   basestruct_t   method_t;        /* Method name list */
typedef   basestruct_t   oconnection_t;   /* List of classes connected */
typedef struct otype {

        struct objecttype *next;                                    /* Next object type */

        long            flag;                                       /* Flags */

        char            *name;                                      /* Name of object type */
        int             level;                                      /* DAG level number */
        struct oclass   *derivedclass;                              /* Corresponding class */
        oconnection_t   *connectionlist;                            /* List of classes */
        attribute_t     *attributelist;                             /* Attribute list */
        subtype_t       *subtypeoflist;                             /* Subtype-of list */
        setof_t         *setoflist;                                 /* Set-of list */
        memberof_t      *memberoflist;                              /* Member-of list */
        memberof_t      *tupleoflist;                               /* Tuple-of list */
        orelation_t     *orelationlist;                             /* ordinary relationships */
        mvrelation_t    *mvrelationlist;                            /* Multi-valued */
        method_t        *methodlist;                                /* Method name list */
        long            attributecnt;                               /* Attribute count */
        long            supertypecnt;                               /* Super-Type count */
        long            subtypeofcnt;                               /* Sub-Type count */
        long            memberofcnt;                                /* Memberof count */
        long            setofcnt;                                   /* Setof count */
        long            relationshipcnt;                            /* Relationship count */
        long            methodcnt;                                  /* Method count */

        char            *foruser[2];                                /* 2 private slots for users */
        char            *future[8];                                 /* Future expansion */

} otype_t;
```

```
typedef struct oclass {

        struct oclass   *next;                                  /* Next class pointer */

        struct oclass   *nexthashp;                             /* Next hash pointer */

        char            *name;                                  /* Name of class */
        int             level;                                  /* DAG level number */
        long            flag;                                   /* Flags */
        otype_t         *derivedtype;                           /* Derived object type */
        otype_t         *actualtype;                            /* Actual object type */
        roleof_t        *roleoflist;                            /* Role-of connection list */
        categoryof_t    *categoryoflist;                        /* Category-of list */
        partof_t        *partoflist;                            /* part-of connection list */
        partof_t        *tupleoflist;                           /* tuple-of connection list */
        roleof_t        *rolegenlist;                           /* Role-Generalized list */
        categoryof_t    *categorygenlist;                       /* Catg.-Generalized list */
        partof_t        *partgenlist;                           /* Part-Generalized list */
        eattribute_t    *eattributelist;                        /* Essential attribute list */
        orelation_t     *orelationlist;                         /* ordinary relationships */
        erelation_t     *erelationlist;                         /* Essential relationships */
        drelation_t     *drelationlist;                         /* Dependent relationships */
        mvrelation_t    *mvrelationlist;                        /* Multi-valued */
        mverelation_t   *mverelationlist;                       /* Multi-valued Essential */
        mvdrelation_t   *mvdrelationlist;                       /* Multi-valued Dependent */
        long            attributecnt;                           /* Attribute count */
        long            roleofcnt;                              /* Roleof count */
        long            groleofcnt;                             /* Generalized count */
        long            categoryofcnt;                          /* Categoryof count */
        long            gcategoryofcnt;                         /* Generalized count */
        long            partofcnt;                              /* Partof count */
        long            gpartofcnt;                             /* Generalized count */
        long            tupleofcnt;                             /* Tupleof count */
        long            gtupleof;                               /* Generalized count */

        long            *foruser[2];                            /* User private slots */
        char            *future[8];                             /* Future expansion */

} oclass_t;

/* Global externs for library users */
extern oclass_t         *classptr;        /* Pointer to list of object classes */
extern otype_t          *typeptr;         /* Pointer to list of object types */
extern oclass_t         **classhashptr;   /* Pointer to list of hashed classes */

/* Hash table size */
#define CHASHSIZE  256

/* Hashing Function -- use first character of classname into an ASCII table */
#define CHASHINDX(classname)        (((int)(*(classname))) % CHASHSIZE)
```

APPENDIX D: USER API LIBRARY CALL FOR DAL

This section contains the manual page for the User API routine *dal()* incorporated in the **dallib.a** library.

NAME

dal - **DAL** library routine invocation.

C SYNOPSIS

```
#include "dal.h"

int       dal(heapfilename, oclassp, objectp, dag, integrate, debug_on)

char           *heapfilename;
oclass_t             **oclassp;
otype_t              **objectp;
int            dag;
int            integrate;
int            debug_on;
```

DESCRIPTION

Upon invocation this routine parses the heap file generated from the **OODINI** object oriented database graphics editor. It returns a pointer to a list of object classes and a pointer to a list of object types defined by the user during the **OODINI** session.

ARGUMENTS

The *heapfilename* argument is the heap file name where the graphical image for **OODINI** was stored by the user.

The *oclassp* argument is pointer to an object class whose definition is given in the **dal.h** header file.

The *objectp* argument is a pointer to an object type whose definition is given in the **dal.h** header file.

The *dag* argument if set to **1** will allow the *dal()* routine to verify if the database schema is a directly acyclic graph (**DAG**) or not. If the schema is not a **DAG** then a comment is generated on the standard output to indicate this during **DAL** code generation. If the value of this argument is **0** then *dal()* does not verify if the schema is a **DAG** or not.

The *integrate* argument if set to **1** will allow the *dal()* routine to **collapse** object types. Object types that have a super type and do not have any out-going relationships, relations and attributes are deleted from the system. All object classes that originally referred to this object type has these references changed to that of the super object type. If the value of this argument is **0** then *dal()* ignores this feature. In the *future* releases of this library this option will allow for partial/full structural integration of object types.

The *debug_on* argument allows the *dal()* to generate debugging messages (*if any*) on the standard output. A value of **1** turns on debugging, while a value of **0** turns off debugging.

RESULTS

On success *dal()* returns a value of 1. On failure it returns a value of 0 and the values of the last 2 arguments are undefined. Severe errors in input database schema aborts execution and prints appropriate error messages on the standard error file descriptor.

EXAMPLE

Sample code to invoke *dal()*:

```
int             ret;
oclass_t                *oclassp;
otype_t                         *objectp;

if ((ret = dal("/tmp/heapfile",  &oclassp, &objectp, 0, 0, 0)) !=
0) {
        error processing code;
}
```

APPENDIX E: MANUAL PAGE: DAL EXECUTABLE

This section contains the manual page for the **dal** executable invocation.

NAME

dal - **DAL** executable invocation.

USAGE

dal -h *heapfilename* [**-d**] [**-t**] [**-x**] [**-f** *outputfilename*] [**-e** *errorfilename*]

DESCRIPTION

Upon invocation this executable parses the heap file generated from the **OODINI** object oriented database graphics editor. It generates an *abstract* textual code form of the database schema.

ARGUMENTS

The **-h** *heapfilename* argument is used to specify the heap file name where the graphical image for **OODINI** was stored by the user.

The **-d** optional argument allows **dal()** to verify if the database schema is a directly acyclic graph (**DAG**) or not. If the schema is not a **DAG** then a comment is generated on the standard output to indicate this during **DAL** code generation. If this optional argument is not used then **dal** does not verify if the schema is a **DAG** or not.

The **-t** argument allows **dal** to **collapse** object types. Object types that have a super type and do not have any out-going relationships, relations and attributes are deleted from the system. All object classes that originally referred to this object type has these references changed to that of the super object type. If this optional argument is not used then **dal** ignores this feature. In the *future* releases of this software this option will allow for partial/full structural integration of object types.

The **-x** optional argument allows **dal** to generate debugging messages (*if any*) on the standard output. If this optional argument is not used then **dal** does not generate any debugging messages.

The **-f** *outputfilename* optional argument allows **dal** to generate the **DAL** code in *outputfilename*. If this option is not used then the **dal** output is generated on the standard output which can also be redirected to an output file at the **UNIX** shell level.

The **-e** *errorfilename* optional argument allows **dal** to generate all error messages (*if any*) in *errorfilename*. If this option is not used then all error messages are generated on the standard error which can also be redirected to an output file at the **UNIX** shell level.

RESULTS

On success **dal** exits with an exit code value of 0. On failure the exit code value is non-zero. Severe errors in input database schema aborts execution and prints appropriate error messages.

APPENDIX F: MANUAL PAGE: VML EXECUTABLE

This section contains the manual page for the **vml** executable invocation.

NAME

vml - **vml** executable invocation.

USAGE

vml **-h** *heapfilename* [**-d**] [**-t**] [**-x**] [**-f** *outputfilename*] [**-e** *errorfilename*]

DESCRIPTION

Upon invocation this routine parses the heap file generated from the **OODINI** object oriented database graphics editor. It generates **VML** (**VODAK Data Modeling Language**) code syntax of the **OODINI** database schema.

ARGUMENTS

The **-h** *heapfilename* argument is used to specify the heap file name where the graphical image for **OODINI** was stored by the user.

The **-d** optional argument allows **vml** to **disable** verification if the database schema is a directly acyclic graph (**DAG**) or not. By default, **vml** always *sorts* the object types and object classes into a **DAG**. This is necessary for generating correct code for *parametrized* **VML** object types. Using the **-d** option allows the user to disable *sorting* of object types and object classes. In such a case, however, correct generation of **VML** parameterized object types are **not** guaranteed. If the **-d** option is **not** used and the schema is not a **DAG** then **VML** code generation will fail.

The **-t** argument allows **vml** to **collapse** object types. Object types that have a super type and do not have any out-going relationships, relations and attributes are deleted from the system. All object classes that originally referred to this object type has these references changed to that of the super object type. If this optional argument is not used then **vml** ignores this feature. In the *future* releases of this software this option will allow for partial/full structural integration of object types.

The **-x** optional argument allows **vml** to generate debugging messages (*if any*) on the standard output. If this optional argument is not used then **vml** does not generate any debugging messages.

The **-f** *outputfilename* optional argument allows **vml** to generate the **VML** code in *outputfilename*. If this option is not used then the **vml** output is generated on the standard output which can also be redirected to an output file at the **UNIX** shell level.

The **-e** *errorfilename* optional argument allows **vml** to generate all error messages (*if any*) in *errorfilename*. If this option is not used then all error messages are generated on the standard error which can also be redirected to an output file at the **UNIX** shell level.

RESULTS

On success **vml** exits with an exit code value of 0. On failure the exit code value is non-zero. Severe errors in input database schema aborts execution and prints appropriate error messages.

APPENDIX G: MANUAL PAGE: HEAP FILE COPY

This section contains the manual page for the **oocopy** executable invocation.

NAME

oocopy - **OODINI** Persistent Heap file **copy** command.

USAGE

oocopy *from-filename* *to-filename*

DESCRIPTION

The **oocopy** command *should* be used to copy persistent heap files used by the **OODINI** executable. The persistent heap file used by the **OODINI** executable is *often* very large in size. However, the heap file has *holes* in between, i.e., disk blocks for unused portions of the file are not allocated by the underlying file system. But if the traditional **UNIX cp** (copy command) is used to copy a heap file to another file, then this new file will have disk blocks allocated for unused portions of the file. This is a typical file system behavior. To avoid this problem **oocopy** only copies the used/allocated blocks and the header portion of the unused/free blocks from the persistent heap file to the new file. The saving in disk blocks in the new file are due to **2** reasons. First, the holes in the persistent heap file are ignored by the **oocopy** command. Second, too many allocations followed by deallocations from the persistent heap file by the **OODINI** process will allocate file system blocks for the unused/free portions. The **oocopy** only copies the header portion of these deallocated (and now free) blocks. The entire freed block is not copied and is not necessary.

ARGUMENTS

The *from-filename* argument is used to specify the persistent heap file name where the graphical image for **OODINI** was stored by the user.

The *to-filename* is used to specify the *new* file where the persistent heap file will be copied to.

RESULTS

On success **oocopy** exits with an exit code value of 0. On failure the exit code value is **1** and a failure notification message is generated on the standard output.

APPENDIX H: BNF DESCRIPTION FOR OODAL

This section gives the **BNF** description of the **OODAL** language.
<Start> --> <class>

<class> --> <classbody> <class> | <classbody>

<classbody> --> **class** <classname> <classdefinition> **end** ;

<classdefinition> --> <relations> <attributes> <relationships> <methods>

<relations> --> <setof> <memberof> <categoryof> <roleof> <partof> <tupleof>

<setof> --> **setof** : <class-name-list> ; | <NULL>

<memberof> --> **memberof** : <class-name-list> ; | <NULL>

<categoryof> --> **categoryof** : <class-name-list> ; | <NULL>

<roleof> --> **roleof** : <class-name-list> ; | <NULL>

<partof> --> **partof** : <class-name-list> ; | <NULL>

<tupleof> -> **tupleof** : <connect-list> ; <tupleof> | <NULL>
<connect-list> --> < <connector-name> : <classname> <more-connect>
<more-connect> --> > | , <connector-name> : <classname> <more-connect>

<attributes> --> **attributes** <attrs> **endattributes;** | <NULL>
<attrs> --> <attr> ; <attrs> | <attr> ;
<attr> --> <attribute-name> <attr-separator> **unknown_type**
<attr-separator> --> : | :+

<relationships> --> **relationships** <relationbody> **endrelationships;** | <NULL>

<relationbody> --> <relationshipname> <typeofrelation> <classname> ;
<relationbody> | <NULL>

<typeofrelation> --> : | :+ | :> | :: | ::+ | ::>

<methods> --> **methods** <method-body> **endmethods** ; | <NULL>
<method-body> --> <method-name>(); <method-body> | <method-name>();

<classname> --> *Character string constituting a class name*

<class-name-list> --> <classname> , <class-name-list> | <classname>

<attributename> --> *Character string constituting an attribute name*

<relationshipname> --> *Character string constituting a relationship name*

<connector-name> --> *Character string, a connector name in a tuple-of relation*

<method-name> --> *Character string constituting a method name*

<NULL> --> ""

APPENDIX I: USER API HEADER FILE FOR OODAL

This section lists the contents of the user **API** header file, **oodal.h**.

```
typedef struct basestruct {

        struct basestruct           *next;              /* Next list element pointer */

        int         flag;                               /* What kind of arc */
        char        *name;                              /* Name field */

        union {                                         /* Variant */

                    struct oclass  *classp;             /* Class pointer */
        } un;

        char        *foruser[2];                        /* 2 private slots for users */
        char        *future[4];                         /* Future expansion slots */

} basestruct_t;


typedef   basestruct_t   orelation_t;     /* Ordinary relationships */
typedef   basestruct_t   erelation_t;     /* Essential relationships */
typedef   basestruct_t   drelation_t;     /* Dependent relationships */
typedef   basestruct_t   mvrelation_t;    /* Multi-valued relationships */
typedef   basestruct_t   mverelation_t;   /* Multi-valued Essential */
typedef   basestruct_t   mvdrealtion_t;   /* Multi-valued Dependent */
typedef   basestruct_t   roleof_t;        /* Role-of connection */
typedef   basestruct_t   categoryof_t;    /* Category-of connection */
typedef   basestruct_t   partof_t;        /* Part-of connection */
typedef   basestruct_t   setof_t;         /* Set-of connection */
typedef   basestruct_t   memberof_t;      /* Member-of connection */
typedef   basestruct_t   tupleof_t;       /* Tuple-of connection */
typedef   basestruct_t   subtypeof_t;     /* Subtype-of connection */
typedef   basestruct_t   attributeof_t;   /* Attribute list */
typedef   basestruct_t   essential_t;     /* Essential attribute list */
typedef   basestruct_t   method_t;        /* Method name list */
typedef struct oclass {
        struct oclass   *next;                                  /* Next class pointer */
        struct oclass   *nexthashp;                             /* Next hash pointer */
        char            *name;                                  /* Name of class */
        int             level;                                  /* DAG level number */
        long            flag;                                   /* Flags */
        setof_t                         *setoflist;             /* Set-of connection list */
        memberof_t      *memberoflist;                          /* Member-of list */
        roleof_t        *roleoflist;                            /* Role-of connection list */
        categoryof_t    *categoryoflist;                        /* Catg.-of list */
        partof_t        *partoflist;                            /* part-of connection list */
        partof_t        *tupleoflist;                           /* tuple-of connection list */
        roleof_t        *rolegenlist;                           /* Role-Generalized list */
        categoryof_t    *categorygenlist;                       /* Category-Generalized list */
        partof_t        *partgenlist;                           /* Part-Generalized list */
        eattribute_t    *eattributelist;                        /* Essential attribute list */
        orelation_t     *orelationlist;                         /* ordinary relationships */
        erelation_t     *erelationlist;                         /* Essential relationships */
        drelation_t     *drelationlist;                         /* Dependent relationships */
        mvrelation_t    *mvrelationlist;                        /* Multi-valued */
        mverelation_t   *mverelationlist;                       /* Multi-valued Essential */
        mvdrelation_t   *mvdrelationlist;                       /* Multi-valued Dependent */
        method_t        *methodlist;                            /* Method-name list */
        long            attributecnt;                           /* Attribute count */
        long            setofcnt;                               /* Setof count */
        long            memberofcnt;                            /* Memberof count */
        long            roleofcnt;                              /* Roleof count */
        long            groleofcnt;                             /* Generalized count */
        long            categoryofcnt;                          /* Categoryof count */
        long            gcategoryofcnt;                         /* Generalized count */
        long            partofcnt;                              /* Partof count */
        long            gpartofcnt;                             /* Generalized count */
        long            tupleofcnt;                             /* Tupleof count */
        long            gtupleof;                               /* Generalized count */
        long            methodcnt;                              /* Method count */

        long            *foruser[2];                            /* User private slots */
        method_t        *future[8];                             /* Future expansion */
} oclass_t;

/* Global externs for library users */
extern oclass_t         *classptr;      /* Pointer to list of object classes */
extern oclass_t         **classhashptr; /* Pointer to list of hashed classes */

/* Hash table size */
#define CHASHSIZE   256

/* Hashing Function -- use first character of classname into an ASCII table */
#define CHASHINDX(classname)        (((int)(*(classname))) % CHASHSIZE)
```

APPENDIX J: USER API LIBRARY CALL FOR OODAL

This section contains the manual page for the User API routine *oodal()* incorporated in the **oodallib.a** library.

NAME

oodal - **OODAL** library routine invocation.

C SYNOPSIS

```
#include "oodal.h"

int     oodal(heapfilename, oclassp, dag, debug_on)

char          *heapfilename;
oclass_t            **oclassp;
int           dag;
int           debug_on;
```

DESCRIPTION

Upon invocation this routine parses the heap file generated from the **OODINI** object oriented database graphics editor. It returns a pointer to a list of object classes defined by the user during the **OODINI** session.

ARGUMENTS

The *heapfilename* argument is the heap file name where the graphical image for **OODINI** was stored by the user.

The *oclassp* argument is pointer to an object class whose definition is given in the **oodal.h** header file.

The *dag* argument if set to **1** will allow the *oodal()* routine to verify if the database schema is a directly acyclic graph (**DAG**) or not. If the schema is not a **DAG** then a comment is generated on the standard output to indicate this during **OODAL** code generation. If the value of this argument is **0** then *oodal()* does not verify if the schema is a **DAG** or not.

The *debug_on* argument allows the *oodal()* to generate debugging messages (*if any*) on the standard output. A value of **1** turns on debugging, while a value of **0** turns off debugging.

RESULTS

On success *oodal()* returns a value of 1. On failure it returns a value of 0 and the values of the last 2 arguments are undefined. Severe errors in input database schema aborts execution and prints appropriate error messages on the standard error file descriptor.

EXAMPLE

Sample code to invoke *oodal()*:

```
int            ret;
oclass_t                *oclassp;

if ((ret = oodal("/tmp/heapfile", &oclassp, 0, 0)) != 0) {
        error processing code;
}
```

APPENDIX K: MANUAL PAGE: OODAL EXECUTABLE

This section contains the manual page for the **oodal** executable invocation.

NAME

oodal - **OODAL** executable invocation.

USAGE

oodal -h *heapfilename* [**-d**] [**-x**] [**-f** *outputfilename*] [**-e** *errorfilename*]

DESCRIPTION

Upon invocation this executable parses the heap file generated from the **OODINI** object oriented database graphics editor. It generates an *abstract* textual code form of the database schema.

ARGUMENTS

The **-h** *heapfilename* argument is used to specify the heap file name where the graphical image for **OODINI** was stored by the user.

The **-d** optional argument allows **oodal**() to verify if the database schema is a directly acyclic graph (**DAG**) or not. If the schema is not a **DAG** then a comment is generated on the standard output to indicate this during **OODAL** code generation. If this optional argument is not used then **oodal** does not verify if the schema is a **DAG** or not.

The **-x** optional argument allows **oodal** to generate debugging messages (*if any*) on the standard output. If this optional argument is not used then **oodal** does not generate any debugging messages.

The **-f** *outputfilename* optional argument allows **oodal** to generate the **OODAL** code in *outputfilename*. If this option is not used then the **oodal** output is generated on the standard output which can also be redirected to an output file at the **UNIX** shell level.

The **-e** *errorfilename* optional argument allows **oodal** to generate all error messages (*if any*) in *errorfilename*. If this option is not used then all error messages are generated on the standard error which can also be redirected to an output file at the **UNIX** shell level.

RESULTS

On success **oodal** exits with an exit code value of 0. On failure the exit code value is non-zero. Severe errors in input database schema aborts execution and prints appropriate error messages.

APPENDIX L: SOURCE TREE AND SOURCE COMPILATION:

This section describes the source tree layout for the **DAL**, **OODAL**, **VML**, **mapmalloc** and **oocopy** implementations. The following diagram gives a high level view of the source tree layout. Note that **beret** is the login name under which all the source were implemented.

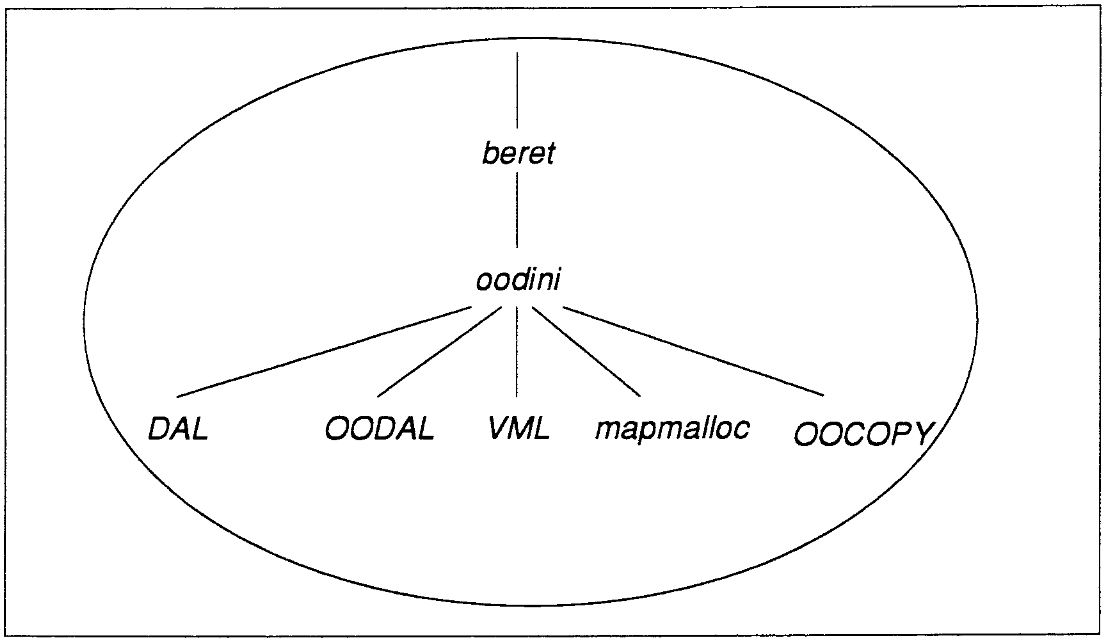

Figure 20. Source Code Tree Hierarchy

DAL Source:

The following table lists the **DAL** source files and corresponding high level implementation comments.

TABLE 1. DAL Source File Names

No.	Source File	High Level Source Comment(s)
1.	makefile	Make file to generate **DAL** executable and library.
2.	genlisp.h	Header file for generic list manipulations.
3.	dalcode.h	Header file for **DAL** code generation.
4.	dal.h	Header file for **dalib.a** library.
5.	genlisp.c	Generic List manipulation code.
6.	dallib.c	**dallib.a** library source code.
7.	daltype.c	**DAL** object type handling source code.
8.	dalclass.c	**DAL** object class handling source code.
9.	dal.c	**DAL** high level source code for **dal** executable.
10.	globals.c	Global variable declarations source file.

Source Compilation:

To compile the source execute the following command at the user prompt:

make -f **makefile all**

Target(s) Generated:

Upon successful compilation the following targets are generated.

TABLE 2. DAL Executable/Library File Names

No.	Target Name	Target Description
1.	**dal**	The **DAL** executable.
2.	**dallib.a**	The **DAL** library.

OODAL Source:

The following table lists the **OODAL** source files and corresponding high level implementation comments.

TABLE 3. OODAL Source File Names

No.	Source File	High Level Source Comment(s)
1.	makefile	Make file to generate **OODAL** executable and library.
2.	oodalcode.h	Header file for **OODAL** code generation.
3.	oodal.h	Header file for **oodalib.a** library.
4.	oodallib.c	**oodallib.a** library source code.
5.	oodalclass.c	**OODAL** object class handling source code.
6.	oodal.c	**OODAL** high level source code for **oodal** executable.
7.	globals.c	Global variable declarations source file.

Source Compilation:

To compile the source execute the following command at the user prompt:

make -f **makefile all**

Target(s) Generated:

Upon successful compilation the following targets are generated.

TABLE 4. OODAL Executable/Library File Names

No.	Target Name	Target Description
1.	**oodal**	The **OODAL** executable.
2.	**oodallib.a**	The **OODAL** library.

VML Source:

The following table lists the **VML** source files and corresponding high level implementation comments.

TABLE 5. VML Source File Names

No.	Source File	High Level Source Comment(s)
1.	makefile	Make file to generate **VML** executable.
2.	vml.h	Header file for **VML** source code generation.
3.	vmltype.c	**VML** object type handling source code.
4.	vmlclass.c	**VML** object class handling source code.
5.	vml.c	**VML** high level source code for **vml** executable.
6.	vmlsemantic.c	Code for **VML** metaclasses.

Source Compilation:

To compile the source execute the following command at the user prompt:

```
make -f makefile all
```

Target(s) Generated:

Upon successful compilation the following targets are generated.

TABLE 6. VML Executable File Name

No.	Target Name	Target Description
1.	**vml**	The **VML** executable.

MAPMALLOC Source:

The following table lists the *persistent heap* (**MAPMALLOC**) source files and corresponding high level implementation comments.

TABLE 7. MAPMALLOC Source File Names

No.	Source File	High Level Source Comment(s)
1.	mapmalloc.h	Header file for **MAPMALLOC** feature.
2.	mapmalloc.c	Source file for **MAPMALLOC** object code generation.

Source Compilation:

To compile the source execute the following command at the user prompt:

```
cc -c mapmalloc.c
```

Target(s) Generated:

Upon successful compilation the following targets are generated.

TABLE 8. MAPMALLOC Object File Name

No.	Target Name	Target Description
1.	**mapmalloc.o**	The **MAPMALLOC** object file.

OOCOPY Source:

The following table lists the **OOCOPY** source files and corresponding high level implementation comments.

TABLE 9. OOCOPY Source File Names

No.	Source File	High Level Source Comment(s)
1.	makefile	Make file to generate **OOCOPY** executable.
2.	oocopy.c	**OOCOPY** source file.

Source Compilation:

To compile the source execute the following command at the user prompt:

```
make  -f  makefile  all
```

Target(s) Generated:

Upon successful compilation the following targets are generated.

TABLE 10. OOCOPY Executable File Names

No.	Target Name	Target Description
1.	**oocopy**	The **oocopy** executable.

REFERENCES

1. Neuhold, E. J., Perl, Y., and Turau, V. "The Dual Model for Object-Oriented Databases, Institute for Integrated Systems." CIS Department and Center for Manufacturing Systems, New Jersey Institute of Technology, Newark, NJ, Research Report: CIS-91-30.

2. Halper, M., Geller, J., and Perl, Y. "An OODB Graphical Schema Representation." CIS Department, New Jersey Institute of Technology, Newark, NJ, Research Report: CIS-92-01.

3. "User's Guide to PSG." Praktische Informatik, Technische Hochschule Darmstadt, MagdalenestraBe 11c, D-61 Darmstadt, West Germany, Report PI-R4/88, October 1989.

4. Geller, J., Neuhold, E. J., Perl, Y., and Turau, V. "A Theoretical Underlying Dual Model for Knowledge-based Systems." Proceedings of the First International Conference on Systems Integration, Morristown, NJ, pages 96-103, 1990.

5. Neuhold, E. J., Perl. Y, Geller, J., and Turau, V. "Separating Structural and Semantic Elements in Object-Oriented Knowledge Bases." Proceedings of the Advanced Database System Symposium, Kyoto, Japan, pages 67-74, 1989.

6. Geller, J., Perl, Y., and Neuhold, E. J. "Structure and Semantics in OODB Class Specifications." SIGMOD Record, Vol. 20, pages 40-43, December, 1991.

7. Geller, J., Perl, Y., and Neuhold, E. J. "Structural Schema Integration with Full and Partial Correspondence using the Dual Mode." Institute for Integrated Systems, CIS Department and Center for Manufacturing Systems, New Jersey Institute of Technology, Newark, N.J. 07102; Institute for Integrated Publication and Information Systems, GMD, Darmstadt, Federal Republic of Germany, Research Report: CIS-91-11.

8. Geller, J., Perl, Y., Cannata, P., and Sheth, A. "Structural Intergration: Concepts and Case Study." Institute for Integrated Systems, CIS department and Center for Manufacturing Systems, New Jersey Institute of Technology, Newark, N.J. 07102; Bellcore 444 Hoes Lane, Piscataway, N.J. 08854; GMD-IPSI Integrated Publication and Information Systems Institute, Dolivostr. 15, D-6100 Darmstadt, Germany, Research Report: CIS-92-02.

9. Kambayashi, Y., Rusinkiewicz, M., and Sheth, A. "Structural Schema Integration in Heterogeneous Multi-Database Systems using the Dual Model." First International Workshop on Interoperability in Multidatabase Systems, Kyoto, Japan, April 7-9, 1991.