Analysis, design & implementation of a high-performance main-memory object database for Splice2

Erwin Gribnau

April 2007

Faculteit Informatica,
Universiteit Twente

Signaal's command & control systems require high-performance data distribution and main-memory storage for periodically produced data, sensor data for instance. The developed main-memory object database can be used to built a new generation of SPLICE, Signaal's real-time distributed database.

As a starting point the object database interface defined by the Object Database Management Group was used. Due to Signaal's real-time requirements the transaction system had to be removed from the interface. It is replaced by a reference mechanism that uses three types of references: a read & write reference, a read-only reference and a keep-only reference. This mechanism is created using C++ smart pointers. These smart pointers provide easy of use to the programmer.

Contrary to Signaal's strong desire to use a standard API, the standard ODMG API had to be changed. The database used in SPLICE has such special requirements in terms of performance that it is not feasible to use transactional disk-resident object database to replace the main-memory storage.

The work done was eventually used in a commercially available product called OpenSplice by PrismTech.

Contents

List of Figures

List of Tables

Preface

Even the longest periods have an end. That includes my study at the department of Computer Science at Twente University. Having spent three months at Holland Signaal working on SPLICE for a student internship earlier, I knew it was a great place to do a graduation project, because of the freedom you get to try and experiment. Also, the idea to work for the same coach as during my internship appealed to me.

My graduation project was at the Systems Infrastructure & Software department of Holland Signaal, the Infrastructure department in particular. This department provides the middleware on which Signaal's Command & Control systems are built. SPLICE is such a middleware product.

Unfortunately, during the course of my work at Signaal I started suffering from RSI. By the end of 1999 I wasn't able to lift a teapot with my right hand anymore.

I was advised by my doctor to take a lot of rest. Early 2000 money constraints made me start working. The internet hype made it very easy for me to find a job. KPN offered me a job with little computer work, or so they claimed. The following years I had to stop working several times because of RSI. It took over three years and a lot of training before I was able to work without recurring pain. And Twente University was far away.

My professional life changed in januari 2006 because of a huge re-organization of my employer at that time, Atos Origin. This made me look back and realize I still really wanted to finish my degree in Computer Science. Twente University was willing to cooperate. A requirement from the university was that the original supervisor from Signaal, Robert Poth, was willing to review the final result. And he was. Even though by the end of 2006 Robert Poth was very, very busy for his new company he took the time to read this report and approve it.

It is hard to express my gratitude towards Twente University, Pierre Jansen and Robert Poth in particular. This thesis would never have been finished without their support.

Hellevoetsluis, April 2007,

Erwin Gribnau

Introduction

New developments like object orientation and pressure to use commercial off the shelf (COTS) software and hardware components force Signaal to review its current middleware solutions.

SPLICE is such a middleware solution. The SPLICE system is currently 8 years old. A new project is running to replace the old implementation with a more efficient one and provide new functionality that is needed to be able to develop SPLICE-software more efficiently and adapt SPLICE to more open standards and new techniques.

Assignment

From the original assigment defined by Signaal in March 1999:

> *The assignment is about the analysis, design and implementation of a real-time object-oriented main-memory database system for Splice2. Due to the time-critical nature of C&C*[1] *systems, this database will have to implement high-performance real-time transactions. Most data distributed in C&C systems, will never need to be stored on disk, therefore the databse will have to support persistent and non-persistent data.*

Influences

Several developments in Signaal strongly influenced this assignment:

1. A strong desire within Signaal to adopt standard API's strongly influenced the choices made in chapter 3. The reasons for this desire are mainly
   - to reduce the amount of training needed for application developers, and
   - to have drop-in replacements available for standardized components.
2. Increase developer productivity by adopting new programming language features, such as object orientation in C++. This made C++ the language of choice for the implementation part of this assignment.

Changing the title

During my assignment part of the title of this assignement was changed from *real-time* to *high-performance*. The main reason for this change is that there are no real real-time constraints defined for this new database system. Signaal's distributed systems use the following measures to be as close to real-time as possible

- Minimize latencies where possible
- Provide clock-synchronization amongst the nodes of the distributed system

[1]Command & Control

- Add timestamps to all data

The measures above are best-effort measures, and not (hard/soft) real-time guarantees. Hence, changing title change.

Transactions

The original assignment stated that *this database will have to implement high-performance real-time transactions.* Chapter 6 will discuss the disadvantages of a transaction system for real-time systems. Chapter 8 will show a replacement for the transaction mechanism.

Limitations

Just considering the implementation part, this assignment is a huge undertaking. Building such a system as a commercial quality product will probably take many man years to complete. Therefore, the scope has been limited. The requirement to support persistent data has been dropped. Altogether, the emphasis of this assignment is more on the analysis & design part, than on implementing a complete product. The parts that were actually implemented have been implemented to make sure the ideas put forward in this report actually work.

Literature

Literature on designing databases in general and main-memory databases in particular is very limited. There is an enormous amount of literature on using and programming databases, however. This is strange because

- numerous databases have been implemented,
- some databases are open source,
- even some main-memory databases are open source [Knia].

This report

This report describes the work and research done on replacing a part of SPLICE, the local database, with a high-performance main-memory object database. This report is divided into five parts:

1. Analysis
   Problem analysis.
2. Design
   Design of a new object database
3. Implementation
   Implementation details
4. Results
   Results, conclusions and retrospect
5. Appendix
   Splice2 Object Definition Language and the bibliography

Part I

Analysis

1 SPLICE introduction

SPLICE is an acronym of:

> **S**ubscription **P**aradigm for the **L**ogical **I**nterconnection of **C**oncurrent **E**ngines

SPLICE in short:

> 'SPLICE is a real-time distributed database system, that provides an asynchronous communication mechanism for data exchange between various applications in a distributed environment' [Sig93]

SPLICE is based on a subscription paradigm, applications subscribe to data that other applications publish. This chapter explains what SPLICE is and how it works. A good introduction to SPLICE can be found in [Sig93].

1.1 SPLICE conceptually

The notions *producer* and *consumer* are introduced because an application can publish data and subscribe to data multiple times. Consumers and producers must identify themselves to SPLICE, they need to tell SPLICE what data they consume or produce. Producers and consumers communicate with each other using messages defined by a datasort. A datasort defines the layout of the message. A producer can only produce *one* datasort, i.e. one type of message. A consumer can only consume *one* datasort.

If an application wants to publish (consume) two or more different datasorts, it plays the publisher (consumer) role two times or more and identifies itself to SPLICE for every datasort it wants to produce (consume).

In traditional client-server systems data is kept at a central server and may be resident in a client-side cache *after* a client's request for data. In a publish/subscribe system the system knows in advance which applications require which data. This gives the system the ability to send the data immediately to the applications that are interested.

In real-time systems *sending in advance* has the advantage that an application has all necessary data locally when it accesses it. Consumers do not have to wait for (possibly slow) responses from a server. In this way consumers can run completely asynchronous with respect to their producers.

Every consumer has its own buffer in which SPLICE stores all messages[1]. The buffer is part of SPLICE (see figure 1.1). In this way, SPLICE handles the arrival of new data asynchronously. The data can be read by the consumer at any moment it wants. An application that is subscribed to various datasorts, is associated with a separate buffer for each subscription it has.

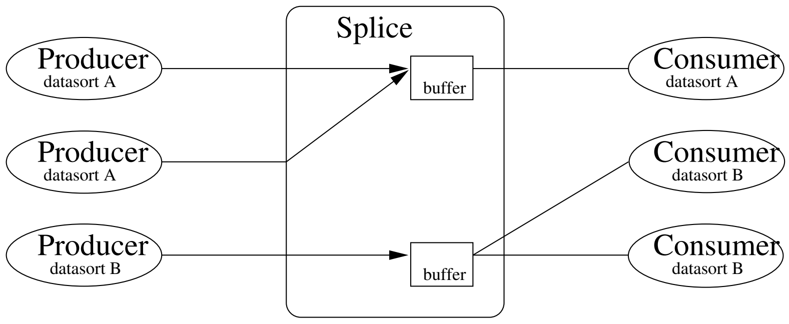


Figure 1.1: SPLICE overview

Figure 1.1 shows the SPLICE-system with consumers and producers for two different datasorts. The SPLICE-system depicted here, is the overall system, across all nodes in the distributed environment.

Figure 1.2 show a system consisting of three nodes. Node 1 contains one producer for a specific datasort, nodes 2 and 3 contain one and two consumers for that datasort respectively. When the producer in node 1 produces a message, this message is send across the network and copied into all the node-specific buffers for that datasort.

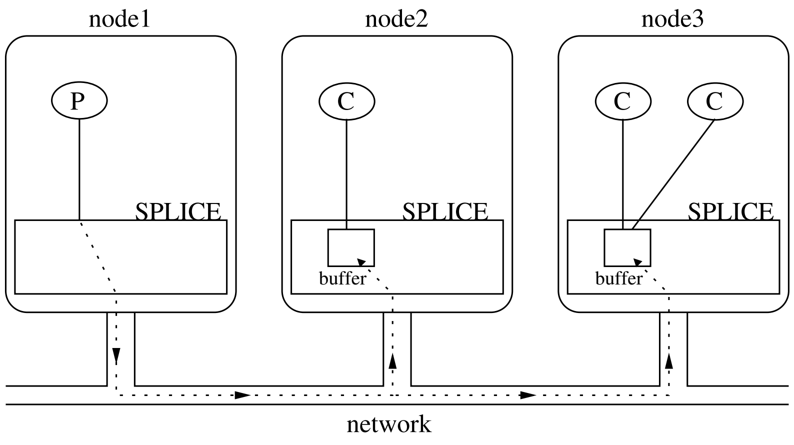


Figure 1.2: Consumer buffers in nodes

1.1.1 Producers

Producers publish data without knowing whether this data will be used by a consumer. Therefore, the producer

- does not know whether there are consumers interested in its data,
- will not wait until the data is received, and
- does not need to be informed that data has been received.

[1]All messages in a buffer are of the same datasort, i.e. the same type of message

Publication

To a producer a publication consist of the following steps

1. Create an instantiation of a datasort locally
2. Fill it with with data (e.g. from sensor)
3. Ask SPLICE to publish it

1.1.2 Consumers

Consumers will consume data without knowing which producer produced the data. Also, it does not matter to the consumer, whether the data have been produced by one or several producers.

Consumption

To a consumer a consumption consists of the following steps

1. Be informed by SPLICE that new data is available
2. Ask SPLICE to copy the latest data to the consumer's process space
3. Use it

It is important to note that before being available to a consumer the data needs to be copied from the SPLICE buffer to the process space of the consumer.

1.1.3 Datasorts

Communication between producers and consumers is purely message-based. The definition of a message is called a datasort. A datasort describes the layout, like a struct in the language 'C', of a message. Instantiations of datasorts are distributed from producer to consumers. Datasorts contain type-definitions, attributes and datasort-properties. They are called `component_definitions`, `component_spec` and `attributes` in SPLICE respectively. Figure 1.3 shows an example datasort definition of `tactical_track`.

One of the datasort-properties is the persistence level, based upon the real-time characteristics of the data.

Data characteristics

SPLICE handles three types of data with different persistence levels

1. `periodic` (real-time data)
   e.g. periodic sensor data from radars, trackers, etc.
   - If there are no consumer in the system, data will not be distributed and is lost
   - Low integrity, losing a message *does not* jeopardize system integrity

```
datasort tactical_track;
  component_definitions(
    component_spec track_id_type = (
      INTEGER,
      range:  0...2000000000,
      unit:  "track id"
    );
    component_spec source_id_type = (
      INTEGER,
      range:  0...2147483647,
      unit:  "source"
    );
    component_spec category_type = (
      ENUMERATION,
      range:  air_point, land_point, subsurface_point, surface_point, acoustic_bearing,
ecm_bearing, esm_bearing, visual_bearing, air_ref_pos, asw_ref_pos, gen_ref_pos, datum_ref_pos,
ecm_fix_ref_pos, esm_fix_ref_pos, acoustic_fix_ref_pos
    );
    component_spec validity_type = (
      ENUMERATION,
      range:  invalid, valid
    );
  );
  datasort_components(
    tact_track_id of type track_id_type ;
    tact_source_id of type source_id_type ;
    category of type category_type ;
    datasort_validity of type validity_type ;
  );
  with attributes(
    dataclass = CONTEXT ;
    context_expiry_time = INFINITE ;
    context_spectrum = tact_track_id ;
  );
end datasort tactical_track;
```

Figure 1.3: Example datasort definition

- No arrival checking, a message may get lost
- No overflow checking, a message may get lost
- (very) Short life span
- Produced periodically

2. `context` (non-real-time data)
   e.g. system state, mission context
   - Distributed to a node when it is added to an operational system
   - Distributed to all nodes even if there is no consumer
   - New consumers for context-data can ask SPLICE for initial data, to get all data available
   - High integrity, losing a message *does* jeopardize system integrity
   - Long life span (mostly valid until shutdown)
   - Not periodically produced
3. `persistent`
   e.g. system parameters and configuration
   - Comparable with context data, with higher integrity
   - Must remain available when system is switched off and restarted

- Stored on disk
- Read into SPLICE at startup

Because context and persistent data need to be stored indepent of any existing consumers, every node in the system has separate buffers for these types of data. Figure 1.4 shows a node containing these separate buffers. Every buffer is for a specific datasort. In this case four context data buffers and three persistent data buffers.

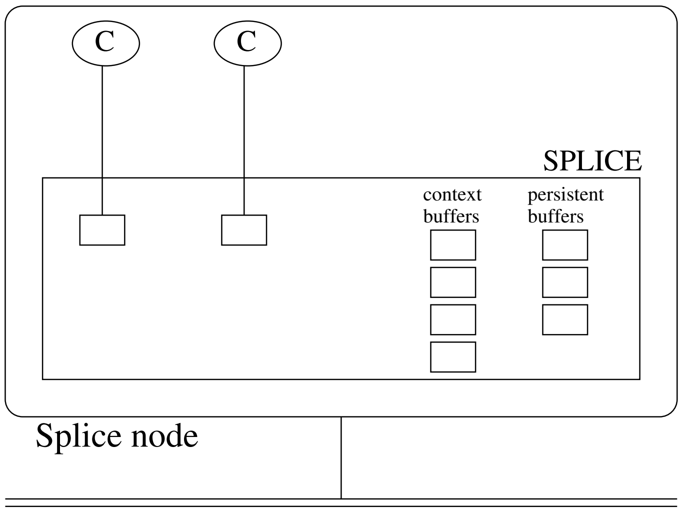


Figure 1.4: SPLICE node internally

Every node also has an administration of available worlds[2], consumers and producers and available SPLICE-programs. This administration is known as the SPLICE-administration.

1.1.4 Consistency

If

- two consumers at different nodes,
- subscribed to the same datasort,
- virtually at the same time,

issue a read request and receive the same data, the database system is consistent.

In a distributed database system this is hard to achieve, since it is not possible to write into the buffer of every consumer at exactly the same time. This is not possible because of some differences between the nodes, like:

- small differences in the system clocks of the nodes
- different locations in the netwerk

[2]This creates groups of SPLICE-programs operating in a different domain. This allows for a separation between real-world data and simulated data for example.

- different loads on the system

Implementing a distributed lock mechanism can overcome this problem, but can lead to unwanted differences between the state of the database and the state of the outside world.

SPLICE is used differently from traditional database systems. Traditional database systems have a dominant read-access pattern. The database is accessed with a high frequency for reading, but with a low frequency for writing. SPLICE has a high-frequency reading and writing access pattern. Write-locking a message across all systems is a very expensive operation and can very easily become a bottleneck when system load increases.

Being behind on reality (state of outside world) due to distributed locking, is in SPLICE considered worse than temporary inconsistencies between nodes. SPLICE does not impose consistency of the database, but tries to follow the outside world as swiftly as possible.

1.1.5 Datasort identity

Every datasort (the type) is assigned a unique ID. This ID consists of a world-ID and a datasort-ID. This is needed to achieve a system-wide identification of datasorts in order to communicate datasort definitions across a network.

1.2 SPLICE real-time properties

Except for being assigned a persistence level, datasorts are also assigned a data quality. This attribute is necessary when data instantiations (of the same datasort) are produced by different producers who produce data with a different quality.

Two mechanisms for assigning quality to data exist:

1. A linear decay of quality
2. An expiry time (maximum quality before, minimum quality after)

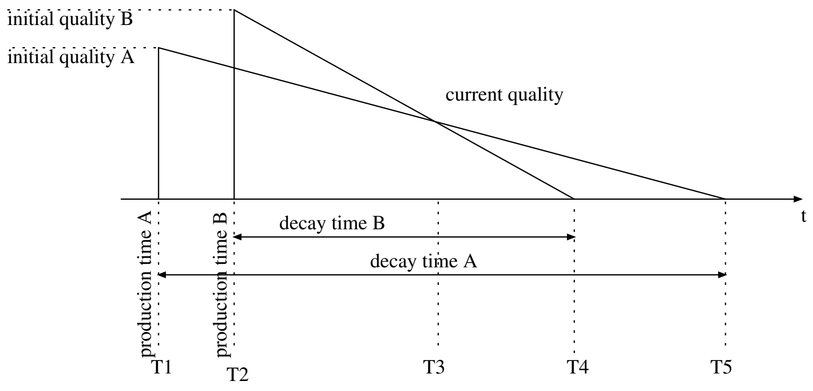


Figure 1.5: Linear decay of data quality

Figure 1.5 shows two datasort instantiations produced by producer A and B using the linear decay mechanism. When the data instantiation from producer B reaches the database (in which the data from producer A is already present), the present quality of both instantiations is determined. The data instantiation will only be updated between T2 and T3 by the data instantiation from producer B, as the data from B has a higher quality during that period.

A garbage collector deletes the expired data in idle time. When a read-action is performed, SPLICE skips the expired data and moves on to the next non-expired data.

1.3 Comparing SPLICE with traditional relational databases

Table 1.1 shows a brief comparison between SPLICE and traditional relational databases.

	SPLICE distributed database system	**Traditional relational database**
Transactions	Every read or write operation is considered atomic	Advanced transaction mechanism with commit and rollback options
Queries	Very limited	Full SQL support
Data Model	Non-relational	Relational
Persistence models	Persistenct, context & real-time	Only persistent and temporary (temporary tables)
Data dictionary	On every node in the form of generated code	Centralized data dictionary

Table 1.1: Comparing SPLICE with a traditional relational database

1.4 SPLICE drawbacks

For Signaal's future needs the current SPLICE has a some drawbacks

1. Datasorts are producer defined (consumers must gather their information from several datasorts which each hold part of the attributes that describe an entity in the real world)
2. It is not possible to define relationships between datasorts and it is not possible to refer to another datasort
3. Non-standard interface
4. Performance-lacks (startup-time and network load)

1.4.1 Producer defined datasorts

Suppose a system containing

- four sensors measuring different properties of an object
- one display that does some calculation on these measurements and displays one number

Each sensor produces a datasort consisting of one attribute, the measured value. The display must subscribe itself to four datasorts, and combine the information in these four datasorts. If the system would contain various displays but doing different calculations, each display would contain the same code for accessing the data. Only the calculation is different.

One way to solve this problem in SPLICE would be to have an intermediate application containing four consumers, that read the data, and a producer that produces a datasort containing all properties. All displays can now consume one datasort, thereby reducing code duplication. But still code that combines datasorts has to be written and maintained.

1.4.2 Relationships, references

The drawback of not having relations or references will be illustrated with an example[3], which will be used throughout this report.

The track example

Track a track is a representation of the track (the path) of a real-world object (e.g. an airplane, a ship, ...)

The following information has to be recorded:

- a trackID
- a creation time
- an identification (enemy, friend, neutral)
- a classification (747, F-16, T80, etc.)
- a series of positions of the object through time

In current SPLICE a consumer program would have to subscribe to four different datasorts: `Track`, `Identification`, `Classification`, `Position`.

These datasorts are defined as shown in table 1.2.

Datasort	`Track`	`Identification`	`Classification`	`Position`
Attributes	trackID	trackID	trackID	trackID
	creation-time	identification	classification	time-stamp
				position

Table 1.2: Basic datasort definitions for the track-example

[3]This example is an educational example, and does not in any way reflect to Signaal's use of the notion 'Track'. In reality this is far more complicated.

The `trackID` field in every datasort can be used to combine data from the four datasorts. The `trackID` field in the datasorts `Identification`, `Classification` and `Position` are foreign keys to `trackID` in `Track`. However, datasort definition do not model them as foreign keys. An application programmer cannot read that knowledge from the datasort definitions.

Every consumer program that uses `Track` has to combine the information for every track itself. This process is repeated within numerous consumer programs, this is costly and error prone.

At first glance it seems logical to simply combine the `Track`, `Identification` and `Classification` datasorts. However, they are (or can be) produced by different producers. The `Identification` datasort may be produced by an IFF[4]-device. The `Classification` datasort may be produced by an application that is used by an operator doing manual classification.

[4]Identification Friend or Foe

2 New Splice2

As part of a redesign of SPLICE a new local high-performance main-memory database will be developed, that will eliminate the drawbacks that were identified for SPLICE. This database will be called Splice2DB from now on. First an architectural overview of Splice2 is discussed in section 2.1. Section 2.3 discusses Splice2DB specific requirements.

2.1 New architecture of Splice2

Figure 2.1 shows a layered view of the new Splice. The top level consists of the applications using Splice2. They can be written in various languages (e.g. C, C++, Java, Ada) and will use a language specific API to communicate with Splice2. This language specific API itself, will use a lower level Splice API to communicate with the Splice2 Engine.

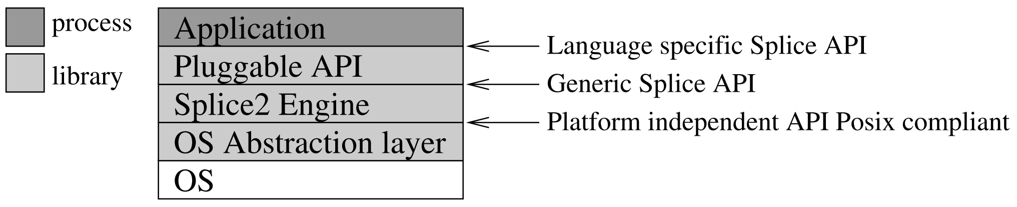


Figure 2.1: Splice2 layered view

The Splice2 Engine consists of three parts (see figure 2.2):

- Kernel, for processing requests from applications
- Network part, provides basic network IO functions
- Database part, provides local data storage for the consumer buffers

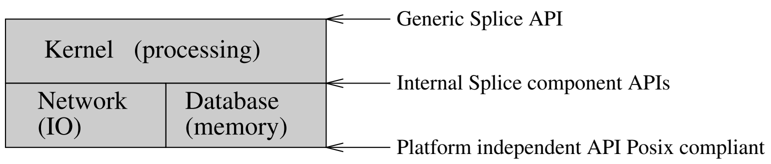

Kernel (processing)
Network (IO)
Database (memory)
Generic Splice API
Internal Splice component APIs
Platform independent API Posix compliant

Figure 2.2: Splice2 Engine

The Splice2 system will be accompanied by several daemons to assist in distributing data across the network and to provide fault-tolerant services to the system. Figure 2.3 shows all of these services together.

A more detailed architecture discussion can be found in [Pot99].

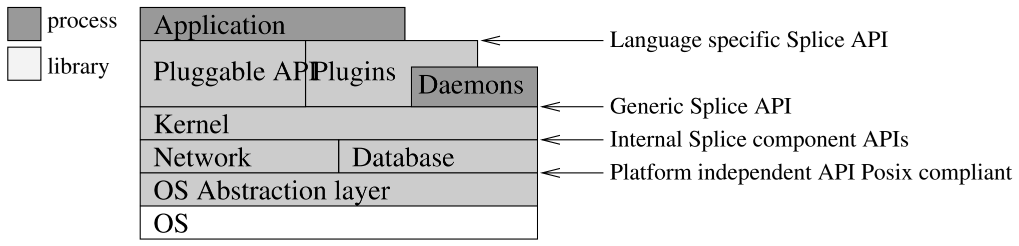


Figure 2.3: Complete Splice2 system layered view

2.2 Splice2DB

The database depicted in figure 2.2 and 2.3 is the scope of this assignment. This database is *not directly accessible* to Splice2 clients. It is used by Splice2 to replace the SPLICE buffers.

Splice2 will use the new Splice2DB for:

1. Storing its internal administration of worlds, consumers and producers
2. Storing data received from producers

2.3 Splice2DB requirements

2.3.1 No unnecessary copying of data

Internally Splice2DB should use references where possible, to avoid unnecessary copying of blocks of memory. If there is more than one consumer for the same data within a node, they should each have a reference to the data, not a copy. In current SPLICE only in a few cases[1] a copy of the data is made. Copying blocks of memory is expensive and should be avoided. Figure 2.4 shows both the unwanted and the wanted situation.

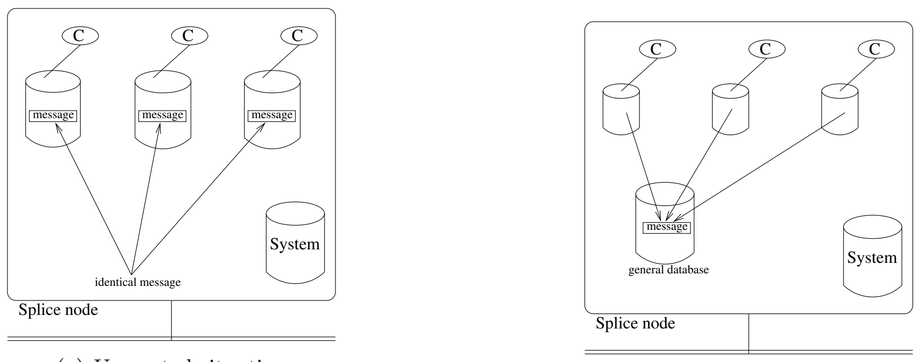
2.3.2 Relations and references

On of the major drawbacks of SPLICE is a lack support for relations and references between datasorts.

Relations are bidirectional. If a datasorts A contains a relation to a datasorts B, then datasorts B contains the inverse relation to a datasorts A.

References are unidirectional. A datasorts A can contain a reference to a datasorts B, B does not necessarily contain a reference to A.

[1]When providing initial data for consumers to context data and when copying data from network buffers to SPLICE memory



(a) Unwanted situation

(b) Wanted situation

Figure 2.4: Distribution of messages

In the 'track'-example (see section 1.4.2) every datasort has a `trackID` field. When an application wants to know the classification of a certain track, it has to scan all classifications for a particular trackID. Of course, using keys and an index mechanism on the `trackID` field can speed up these searches, but still the classification has to be sought for.

If the `Track` datasort could contain a direct reference to its `Classification`, no seeking will be necessary, the classification can be read directly. Table 2.1 shows the track-example using references. The `Identification`, `Classification` and `Position` types don't need a `trackID` field anymore, because you can navigate towards these types using the references in `Track`. This example show the use of unidirectional references. It also shows a collection of unidirectional references, set of Position

Datasort	`Track`	`Identification`	`Classification`	`Position`
Attributes	trackID	identification	classification	time-stamp
	creation-time			position
	reference to Identification			
	reference to Classification			
	***set** of* Position			

Table 2.1: Track-example using references

2.3.3 Persistence

Persistency will not be part of the Splice2DB prototype. Persistency can be dealt with at kernel level. Splice2DB will only implement a main-memory storage initially.

2.3.4 Optimized locking mechanism

The performance of a high-frequency reading & writing database system, is very much related to its locking mechanism. The locking mechanism to be implemented in Splice2DB should be optimized for Signaal's use of SPLICE. The type and granularity of locking will be researched in chapter 8.

2.3.5 Standardized interface

Splice2DB should use a standard database interface as much as possible. In chapter 3 a comparison is done on current database interface standards and a choice is made which standard to use for Splice2DB.

3 Database standards

In order to make a decision which database standard to use in Signaal's future Splice2 database implementation a comparison is done between the two major standards. The two standards considered are the upcoming de jure SQL3 standard and the de facto ODMG 2.0 standard.

First the models are discussed based on the data models and querying mechanisms they provide. After that the standards are compared based on Splice2 requirements and a proposal is done which standard to use as a model for the interface of Splice2DB.

3.1 The models

SQL3 is the successor of SQL2 and is an extension to the relational model. This new model can be called 'object-relational' or 'extended-relational'.

ODMG 2.0 [C$^+$97] is the successor of the ODMG-93 Release 1.2 standard [C$^+$94]. ODMG 2.0 is defined by the Object Database Management Group, a joint effort of the largest Object Data Base Management System vendors.

The article *'Object Database versus Object-Relational Databases'* by Steve McClure [McC97] gives a good overview of the models and standards.

3.1.1 Relational model

The relational data model was originally described by E.F. Codd [Cod70] and is implemented in several products. The eventual standard to describe relational databases is published by ANSI as the X3H2 specification and its ISO counterpart. More popularly, it is referred to as SQL plus a version number, the latest being SQL2. Relational databases have a mathematical foundation in relational theory, but it is not always implemented fully compliant with the theory. This mathematical basis is being further vitiated by the object extensions now being discussed for SQL3.

Data model

A relational database stores data in a database consisting of one or more tables of rows and columns. The rows correspond to a record (tuple); the columns correspond to attributes (fields in the record). Each column has a data type. The types of data that can be stored are confined to a very limited set of data types. A field of a record can store only a single value. Variable length fields are not supported. Relationships are not explicit, but rather implied by values in specific fields. These values are called foreign keys. A foreign key matches the key of a record in a second table. Many-to-many relationships typically require an intermediate table that contains just the relationships.

Query language

A view is a subset of a database that is the result of the evaluation of a query. In a relational database the result of a query is a table. Data is retrieved based on the value in a certain field in a record. The standard query language for relational databases is SQL. SQL can be embedded in the host application.

Extensions

To overcome the shortage of data types most proprietary extensions to the SQL standard have been the introduction of more complex data types, like lists, sets and bags. Another type extension has been the introduction of BLOBs, binary large objects, to include multimedia data types in the database. There are no provisions for querying the content of a BLOB. Thus, one can save and retrieve a BLOB, a document for instance, but cannot query the database for all BLOBs containing a certain pattern.

3.1.2 Object model

Object databases have no official standard. The de facto standard[1] is ODMG 2.0 defined by the Object Database Management Group. The mathematical foundation of object technology is still being debated.

Data model

Object databases use a data model that has object-oriented aspects like:

- classes with attributes and methods,
- provides object identifiers (OID) for any persistent instance of a class,
- support encapsulation (data and methods),
- (multiple) inheritance,
- abstract data types.

Currently there is no consensus on an object-oriented data model, but several proposals exist. These models differ on several issues, like multiple versus single inheritance, and whether the operations/methods or only the definitions should be stored. Part of the source of these differences is the difference in capabilities of OO-languages that should use the OODB.

For example, C++ supports multiple inheritance, so all object databases that support object-persistence and C++ as the host programming language must support multiple inheritance in the object database. The database can also restrict the database programmer to use only single inheritance.

If the object database supports different programming languages, it probably should store the interface of the methods rather than the methods itself. With this kind of problems, it is very difficult to reach an agreement on all parts of the modelling perspective.

[1]November 1999

Query language

An object-oriented programming language is the language for both the application and the database. It provides a very direct relationship between the application object and the stored object. Data definition, manipulation and querying are part of the binding between database and the programming language. The functionality of the database system is mapped into the object-oriented programming language.

The ODMG-standard also defines a query language, OQL, for read-only query of database objects. This query language is also embedded in the host programming language in the form of an API.

3.1.3 Object Relational model

Extended relational and object relational are descriptions for databases that try to unify aspects of both relational and object databases. For most of the products from major vendors 'extended relational' is the most appropriate term. Object relational databases will be specified by the extensions to the SQL standard, SQL 3/4, which is still under development[2]. ORDBMS vendors have been anticipating these extensions in their products, which are therefore proprietary. This category of databases is really a no-man's land between object and relational.

Data model

Object relational databases employ a data model that attempts to 'add OO-ness to tables' [McC97]. All persistent information is still in tables, but fields can have richer data structures, abstract data types (ADT). An ADT is a data type that is constructed by combining basic types. The new data type can be used to index, store and retrieve records based on the content of the new data type. This approach, though it provides more structures for modelling, lacks the fundamental object requirement of encapsulation of operations with data. It also has limited support for relationships, identity, inheritance and integration with host object oriented languages.

Query language

An object relational database supports an extended form of SQL. It has to be an extended SQL to support the object model (e.g. queries involving object attributes).

3.2 Comparing the standards

SQL3 will be the future de jure standard for object-relational databases.
ODMG 2.0 is the de facto standard for object databases.

The standards will be compared on the following issues

- Standard
- Complex data relationships
- SQL2 compatibility
- Integration with host programming languages

[2]November 1999

3.2.1 Standard

SQL3 is not yet a standard, it is expected somewhere in 1999. The ODMG 2.0 standard is the successor of the ODMG-93 standard, ODMG 2.0 has been around since 1997.

3.2.2 Complex data relationships

Because of the use of OIDs in object databases, they are much more suitable for applications that have to process complex relationships. Object relational databases have little support for these capabilities. modelling relationships may require additional tables in the object relational database. Processing usually involves multiple time-consuming joins of tables.

3.2.3 SQL2 compatibility

Object relational databases have the advantage of being SQL-based. The availability of SQL-expertise is enormous. SQL is the most universal database language. Recognizing this fact, most object databases now support at least some of SQL, sometimes including extensions to support object queries using methods, relationships, inheritance and others. OQL is partly SQL-compatible.

3.2.4 Integration with host programming languages

Object databases are tightly integrated into object-oriented programming languages. The ODMG 2.0 standard defines C++, Java and SmallTalk bindings. There is a direct correspondence between an application object and a stored object. Object persistence can be provided by declaring an application class 'persistent capable' by deriving them from a 'persistent capable' base class, and then object instances may be either persistent or transient, but are otherwise the same. There is no overhead due to converting data from the database to a data-object in the programming language. In relational and object relational databases there is no such direct binding. All database functions are executed from embedded SQL statements inserted into the host programming language. This pure query language approach has advantages when providing end-users with querying capabilities but is not an advantage when dealing with a programmer's environment where performance is key issue.

3.2.5 Comparison overview

Table 3.1 shows a short overview of the comparison done in the previous paragraphs.

3.3 Conclusions

- Splice uses the database from a programmer's viewpoint, there are no end-users who need to access the database ad-hoc. Because of the ability to integrate database actions into the host programming language the ODMG standard is most suitable for Splice2.

	SQL-3	**ODMG 2.0**
Standard?	Not yet, de jure standard	Yes, de facto standard
Complex data relationships	Complex data relationships	Can handle arbitrary complexity
SQL2 compatibility	Can take advantageof RDBMS tools and developers	Limited, adapted to pureobject-oriented style
Integration with host programming language	Database functionality is separated from programming language; continuous mapping of programming data to database data	Database functionality integrated into programming language

Table 3.1: Overview SQL-3 / ODMG 2.0 comparison

- SQL is currently not used in SPLICE, so complete SQL-compatibility is not needed to allow backwards compatibility with legacy SPLICE applications.
- Object databases can easily store complex administrations, like the consumer-, producer- & world-administration in SPLICE. Using references and relations, this administration can be used efficiently. No joining of tables is required.
- ODMG 2.0 is the standard to choose

4 Object-oriented approach

Object-oriented modeling has become widely accepted as a very powerful modeling concept, and is still gaining popularity for industrial use. This chapter will show what the advantages of an OO-approach are for Splice.

4.1 Track-example OO model

Modeling the track-example from section 1.4.2 in an object-oriented fashion, the model[1] would look like figure 4.1.

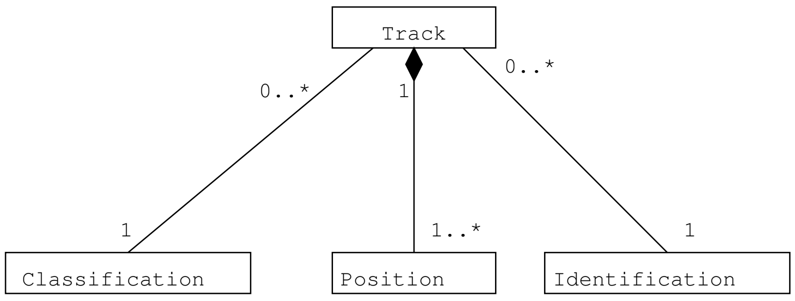


Figure 4.1: Track-example OO model

This model states the following:

- A `Track` associates with **one** `Classification`.
- A `Track` associates with **one** `Identification`.
- A `Track` associates with **one or more** `Position`.
  There can only be a `Track` if you have at least one measured (or estimated) position.
- A `Classification` associates with **zero or more** `Track`.
  A classification 'F-16' can hold for several tracks, and even if there are no `Track`-objects in the system, you want to keep the possible classifications, hence *zero or more.*
- A `Identification` associates with **zero or more** `Track`.
  An identification 'hostile' can hold for several tracks, and even if there are no `Track`-objects in the system, you want to keep the possible identifications, hence *zero or more.*
- A `Position` associates with **one** `Track`.
  This is an composition relation. If there are no tracks in the system, there are no positions in the system. A `Position` (even if there are two position objects with the same values) always belongs to one `Track`.

[1]This is a class diagram using UML notation. For an introduction to UML and design patterns see [Lar98]

Thus, `Classification`-objects and `Identification`-objects have a 'right to live' even if there are no `Track`-objects in the system. `Position`-objects have no 'right to live' if there are no `Track`-objects in the system.

4.2 Track-example in an OODB

Using an OODB the classes can be defined like this[2]:

```
class Identification {
    attribute enum Ident {friend, hostile, neutral, ... } ident;
    relationship set<Track> identifies
        inverse Track::is_identified_by;
}

class Classification {
    attribute enum Classification {F-16,747,T-80, ... } classification;
    relationship set<Track> classifies
        inverse Track::is_classified_by;
}

class Position {
    attribute timestamp time;
    attribute Position position;
}

class Track {
    relationship Identification is_identified_by
        inverse Identification::identifies;
    relationship Classification is_classified_by
        inverse Classification::classifies;
    attribute set<Position> positions;
    attribute timestamp creation_time;
}
```

As opposed to the track-example using unidirectional references in section 2.3.2, this example uses bi-directional relations for the attributes `is_identified_by` and `is_classified_by`. This makes it possible to navigate in both directions.

Class `Identification` has a relation with a set of `Track`. When the relation is defined, the inverse relation has to be defined too. The positions of a track are not implemented as a relation in the definition above, but as a set of `Position` in `Track`. This is because a `Position` always belongs to *one* `Track`. This way, when a `Track` instance is removed, all `Position` instances of that track are removed too.

4.3 Advantages of the OO-track model

An OO-approach has several advantages, only the most notable advantages from the track-example will be discussed here. These are

[2]This definition uses the ODMG Object Definition Language syntax. More on this syntax in chapter C.

- Accessing related data
- Complex types
- Object identity
- Inheritance

4.3.1 Accessing related data

The main advantage illustrated in the previous example is that the relations between object lie within the database. If you have a reference to a `Track`-object called `trackref` that has been constructed earlier, you can create a reference to its identification, like this:

```
Ref<Track> trackref;
Ref<Identification> mytrack_ident = trackref→is_identified_by;
```

If you have the `Identification`-object with `ident=hostile`, you can access all hostile tracks, like this:

```
Identification hostile_ident;
set<Track> hostile_tracks = hostile_ident.identifies;
```

You now have a set containing all hostile tracks. You need an iterator[3] (see [Str97]) to access the elements of the set. An example:

```
set<Track>::iterator hostile_tracks_begin = hostile_tracks.begin();
set<Track>::iterator hostile_tracks_end = hostile_tracks.end();
while(hostile_tracks_begin != hostile_tracks_end) {
   print( (*hostile_tracks_begin)→creation_time );
   hostile_tracks_begin++;
}
```

The type `set<Track>::iterator` is an iterator that returns objects of type `Track` when you access it. The operator `*` returns the object the iterator is currently pointing to. The operator `++` moves an iterator to the next element in the set. An end iterator points *past* the last element in the set and can be compared with a NULL-pointer in the language C.

4.3.2 Complex types

Complex types are built from simpler ones by applying constructors to them The simples objects are objects such as integers, characters, booleans and floats. Using an OODB you can create complex types, using constructs like *sets*, *lists*, *bags*, *arrays*, etc. These complex types are used in the track OO model.

[3]In short, an iterator is a generalization of a pointer. It provides functions to move to the next element and to dereference the iterator. The main difference between pointers and iterators is the fact that the elements don't need to be in memory sequentially. The increment operator can move to the next element by traversing a tree for example.

4.3.3 Object identity

In an OODB every object is assigned an unique number, the *object identifier*. In a model with a unique number, an object has an existence which is independent of its value. Thus two notions of object equivalence exits:

1. two objects can be identical (they are the same object)
2. two objects can be equal (they have the same value)

This has implications:

- object sharing
- object updates

Object sharing

In an identity-based model two objects can share a component. In the track example two tracks can have a relation with the same classification.

Object updates

If in the track example a classification object is updated, all the tracks having a relation with this object now have access to the updated information.

4.3.4 Inheritance

Using inheritance it is possible to distinguish between different kind of tracks, like `SurfaceTrack`, `SubSurfaceTrack`, `Airtrack`, `LandTrack`, all inheriting from `Track` and adding new attributes and/or methods. This is useful if you would like to see them as different concepts.

5 Splice2DB system scope

The main purpose of the Splice2DB is to replace the buffer mechanism used in SPLICE, while adding the needed extra functionality identified in chapters 2 and 4. Additionally, an object-oriented database is powerful enough to store the administration model of SPLICE[1], therefore it is possible to built the SPLICE administration using Splice2 database features.

5.1 Restrictions

5.1.1 Network transport capability

In SPLICE the buffer mechanism itself has no transport capability. Within a node a network daemon[2] acts as a special consumer for all data that needs to be transported across the network to remote consumers. Therefore the replacement, Splice2DB, will not need this capability.

5.1.2 Data characteristics

As with networking capbilities, handling the data characteristics mentioned in section 1.1.3 (`periodic`, `context`, `persistent`), is left to special daemons in SPLICE. Therefore the replacement, Splice2DB, will not need this capability. This means all data will always be in main memory, no disk-access will be necessary.

5.2 New capabilities

Chapters 2 and 4 identified the following new capabilities of Splice2DB:

- Relationships
- References
- Object orientation
  - Complex types
  - Inheritance

[1]The consumer, producer and world-administrations mentioned in chapter 1
[2]This daemon is part of the Splice2 system

5.2.1 Reading datasort definitions

In SPLICE datasort definitions are communicated between nodes to check whether a consumer in a node has the same definitions as the producer. This is because of possible different versions of datasort definitions. This is still needed in Splice2. Checking whether two users of the database have the same idea about the contents of an object is necessary for system stability. Therefore there should be a mechanism that retrieves information about an objects structure. In the ODMG-standard this information is called meta-data. There is a complete, read-only, interface to the meta-data defined by ODMG.

5.3 Architecture

In chapter 2 figure 2.3 shows a complete Splice2 architecture. The database is just a little part of it. Figure 5.1 shows a magnification of this database part into three components.

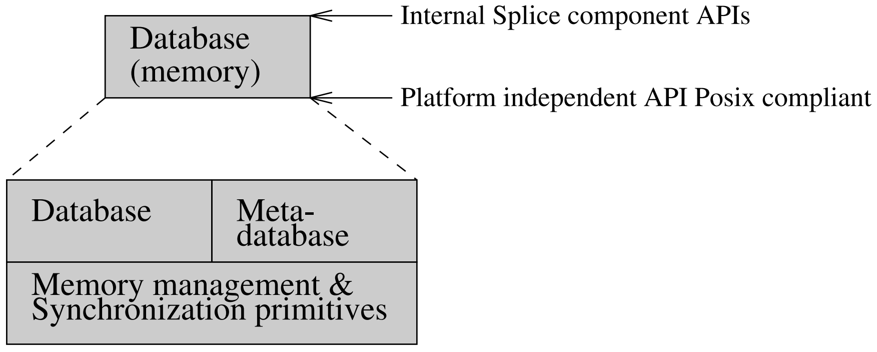

Database (memory)
Internal Splice component APIs
Platform independent API Posix compliant
Database
Meta-database
Memory management & Synchronization primitives

Figure 5.1: Splice2 database architecture

Explanation of parts:

Database

This part will implement the data interface defined by ODMG.

Meta-database

This part will implement the meta-data interface defined by ODMG.

Memory-management & Synchronization primitives

This part will contain low level memory-management routines and synchronization primitives used for concurrency control.

Part II

Design

6 Design overview

This part of the report describes the design of Splice2DB. This chapter will give an introduction to main memory databases and their characteristics. After that an overview is given of a series of basic assumptions and design choices for the database system.

The next chapters in this part will give a more detailed design, describing the following topics:

- Data representation
- Concurrency control
- Data-access interface
- Storage
- Metadata

The definition of the Splice2 Object Definition Language can be found in chapter C in the Appendix.

6.1 Main memory databases

Main memory database (MMDB) systems or memory resident database systems store their data in main physical memory and provide very high-speed access. In conventional database systems (DRDB) data is disk resident, data may be cached into memory for access; in a MMDB the memory resident data may have a backup copy on disk. So in both cases, an object can have copies both in memory and on disk. The key difference is that in a MMDB the primary copy lives *permanently* in memory.

A computer's main memory clearly has different properties from that of magnetics disks, and these differences have profound implications on the design and performance of the database system. A summary of these differences [GMS92]:

1. The access time for main memory is orders of magnitude less than for disk storage.
2. Main memory is normally volatile, while disk storage is persistent.
3. Disks have a high, fixed cost per access that does not depend on the amount of data that is retrieved during the access. For this reason, disks are block-oriented storage devices. Main memory however is not block oriented, it can be accessed byte at a time.
4. The layout of data on a disk is much more critical than the layout of data in main memory, since sequential access to a disk is faster than random access. Sequential access is not that important when accessing main memory.
5. Main memory is normally directly accessible by the processor(s), while disks are not. This may make data in main memory more vulnerable than disk resident data to software errors.

The question arises if DRDB systems with large caches are any different from MMDB systems, because if the cache of a DRDB is large enough, copies of the data in the database will always reside in memory . Although such systems have enough performance for some application areas, they still do not use the full advantage of being able to access data directly. For example, index structures in a DRDB are designed for disk access (B-trees or more advanced equivalents), even though their data may be in memory. Also, applications may have to access the data through a buffer manager, as if the data were on disk. The buffer manager computes the address of needed data. Clearly, if the data always resides in memory, it is more efficient to refer to it using its memory address.

Advanced DRDB systems, that recognize the fact that big parts of their active data set will actually reside in memory and implement some of the optimizations possible for MMDB systems, exist.

Concurrency control

Because access to main memory is much faster than disk access, transactions complete more quickly in a MMDB. In systems that use locking based concurrency control, this means that locks will not be held as long as in DRDBs. For further discussion on concurrency control see chapter 8.

Commit processing

In traditional DRDBs and even in legacy MMDBs it is necessary to have a backup copy of the database and to keep a log of transactions, in order to be able to reconstruct the database if something goes wrong or the system needs to be restarted. In Splice2DB however, all real-time and context data do not need to be copied to disk. If the system goes down, all non-persistent data are gone and they should be gone. Only persistent data need a disk copy. Because Splice2DB will be a part of Splice2 beneath a generic Splice interface (see section 2.1), Splice2DB will not need to have support for persistent data initially. This support can be provided by higher layers, where persistent data can be given to the database and to a separate persistence-manager. On startup this data is read by the persistence-manager and stored in the database. This persistence-manager is not part of the assignment.

Access methods

In a main-memory database, index structures like B-Trees, which are designed for block-oriented storage, lose much of their appeal. Designing a query and index mechanism is not part of the assignment.

Data representation

Main memory databases can also take advantage of efficient pointer following methods for data representation. Complex structures can be navigated directly, instead of having to look up every object. These methods are described in chapter 7.

Database architecture

The ANSI/SPARC architecture divides the architecture of a database system into three levels [Dat90]:

1. *internal*, closest to physical storage
2. *external*, closest to the users
3. *conceptual*, between internal and external, a level of indirection

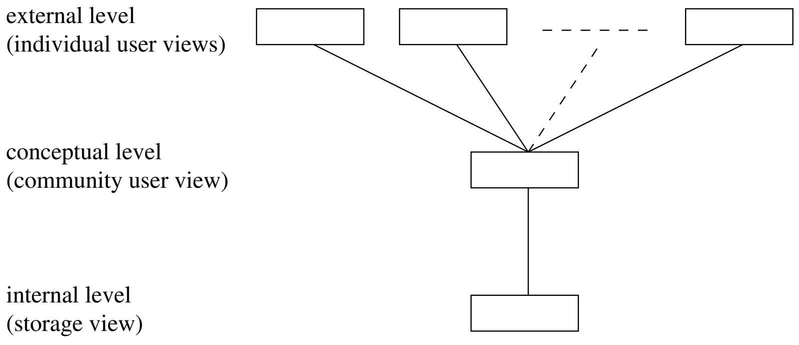


Figure 6.1: ANSI/SPARC architecture

These three levels require two mappings of methods and data, one mapping between the internal level and the conceptual level and one mapping between the conceptual level and the external level . In a high-performance database these mappings must be avoided as much as possible because they slow down the system. The external level should be mapped as directly as possible to the internal level. It is easy to remove the top-level mapping by removing the possibility of different user views. This way the external and conceptual level become the same level. When objects are stored in the database exactly the same way they are stored in application memory, the conceptual level practically becomes the internal level. Of course some parts have to be hidden from an application, but this is easily accomplished using the protection and visibility levels on class attributes in an object-oriented language.

6.2 ACID

In ODMG-standard complying systems all actions on the data have to be executed inside the context of a transaction. Transactions are a way to group database actions. A transaction collects all actions and will only commit the changes at the end of the transaction. Within a transaction all used objects will be locked and, in case of a client/server system, will be collected from the server. Inside a database program this mechanism looks like this:

```
Transaction transaction;
transaction.begin();
   ...
   various database actions
   ...
transaction.commit(); or transaction.abort();
```

This transaction mechanism has four characteristics, denoted by the acronym *ACID*. These characteristics are:

- **A**tomicity
- **C**onsistency
- **I**solation
- **D**urability

Atomicity requires that either all or none of the operations included in the transaction is reflected in the database. These operations often have interdependencies, and performing only a subset of them compromises the overall intent of the transaction

Consistency requires that a transaction ends with a new correct state of the database, preserving its semantical and physical integrity. For example, if a relation has been created, both ends are connected when the transaction ends.

Isolation requires that each transaction appears to be the only transaction manipulating the database even though other transactions may be running concurrently. A transaction should not see intermediate results of other transactions that have not committed yet.

Durability implies that committed transactions are guaranteed to be reflected in the database, even if failures occur after the commit. This has no meaning in a main-memory database, since all changes will be lost when the system is shut-down.

6.2.1 Splice2DB and transactions

Transactions are a method to combine serveral database actions into one action that satisfies the ACID constraints.

But this transaction mechanism has three disadvantages:

1. The possibility to abort a transaction
   To be able to abort a transaction, the database has to do the actions in the transaction on a copy of the original data because of the *isolation constraint.* When the transaction commits these copies will replace the old data. This also requires a log of all actions done. This process uses a lot of resources and is slow.
2. Locks to objects are kept until the transaction completes
   When an object is accessed, the database management system locks that object for the remaining duration of the transaction, even though the object might not be accessed anymore. This can lead to unnecessary locked objects, and can slow down other concurrently running transactions.
3. References to objects only valid inside a transaction
   Because in an ODMG-compliant database there is no other unit of control than a transaction that satisfies the ACID constraints, all references to objects lose their validity outside of a transaction. A transaction is the only unit in which access to objects is granted. Objects in the SPLICE administration are accessed with a high frequency. If finding these object will have to be done by looking up a name, this becomes very expensive (in CPU-time). Thus, it should be possible to have valid references outside of a transaction.

In chapter 8 a design based on object locking, using references, will be discussed.

6.3 Storing data

A database stores data. An object database stores objects. And a lot more. An object database that complies to the ODMG standard has to store the following data:

1. Complex administrations to keep track of the types of objects that can be stored in the database (metadata).
2. Administrations to keep lists of all objects of one class (extents).
3. Administration to associate names with objects.
4. Index-administration for faster access to objects.
5. The object-storage itself.

These different types of data have different usage and access characteristics.

Type of data	Usage	Access
metadata	Intensively read by system, contents is relatively stable in running system	Written by DB-system, read by system & apps
extents	Updated by the DB-system every time an object is inserted in or removed from database	Written by DB-system, read by apps
name-object association	Only written when name is added or changed, only read when an name is looked up	Read & write by DB-system
indices	Updated by the DB-system when an object is changed, inserted in or removed from database, read by DB-system for evaluation of queries	Written by DB-system, read by DB-system
object storage	Intensively read & written	High frequent read & write by apps

The data stored for each data-type also has certain characteristics:

Type of data	Sizes	When is size known?	Data coupling
metadata	Low number of fixed size data chunks	DB library compile time	Very tight
extents	Low number of fixed size data chunks	DB library compile time	Loose
name-object association	Low number of fixed size data chunks	DB library compile time	Loose
indices	Low number of fixed size data chunks	DB library compile time	Loose
object storage	Varying size data chunks	Application compile time	Tight

The field *Data coupling* shows how strong the data of that type is inter-related , how much references there are between data of that type.

It is possible for a system to allocate all storage for the types of data in the same storage arena, using the same allocator. However, this approach eliminates the possibility to use storage-type-optimized allocators and reduces concurrent access to different storages. Taking the usage patterns, access patterns and data characteristics into account, three types of storage will be distinguished in the Splice2DB system:

1. Metadata storage arena
   For metadata
2. Data storage arena
   For object data
3. Administration storage arena
   For the other administrations

The advantages of splitting arenas are:

1. More concurrent access to different storage arenas
2. It is possible to use different allocators, that are optimized for the data they have to store

Disadvantages:

1. Every arena has its own free space. These free spaces do not add together, thus the system will probably use more memory than a system with a one-for-all allocator.
2. When using fixed size storage arenas, choosing the sizes for the arenas is important. Choosing the size to small can result to system failure when a storage gets exhausted.

Figure 6.2 shows the database system consisting of three storage arenas.

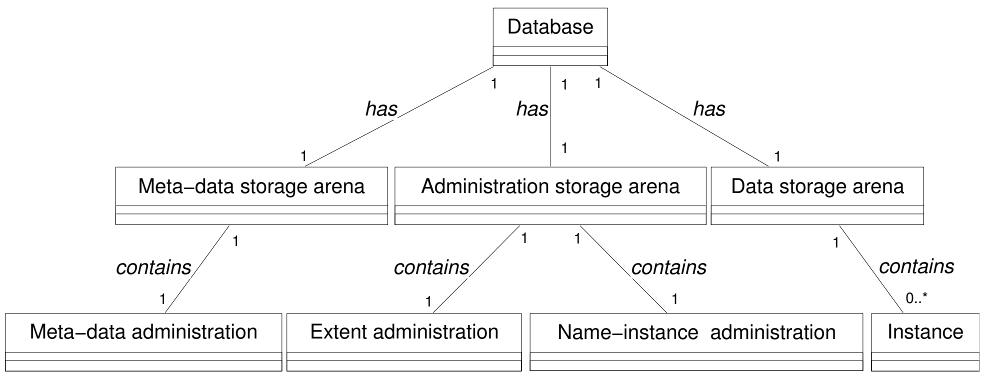


Figure 6.2: Database system storage overview

7 Data representation

Applications and the database system itself, need a way to refer to data stored in the system. This chapter discusses the possible mechanisms to be used. Deciding which mechanism to choose, will be done mainly on basis of expected performance.

7.1 Object identifiers

In a disk resident object database (DRODB), object identifiers (OID) are very important. They have the following roles:

1. OIDs are used (internally) to refer to other objects
2. Two references point to the same object if, and only if, their OIDs are equal

In Splice2DB neither role is necessary, because pointers and pointer comparison can suffice.

OIDs in a DRODB may contain any of the following components [Jor98]:

- Database identifier for the current database of the object
- Identifier of the database in which the object was originally created
- Segment identifier
- Page identifier
- Page offset of an object

None of these components are necessary in a MMODB, since most of them are used to translate an OID to a position in cache or on disk.

However, when sending objects to another node over a network, pointer comparisons do not suffice any more. Objects need to be identified independent of a node's memory layout. Therefore, OIDs are still necessary in Splice2DB. They must be assigned unique across a distributed environment. Splice2DB will assume such a service present, but will not implement such a mechanism itself, since this kind of resource management services will be part of Splice2 [Pot99].

7.2 References

Jordan [Jor98] discusses four mechanisms to refer to data in the system from an application. Figure 7.1 shows two of these four mechanism schematically. Only these two are applicable to MMDBs.

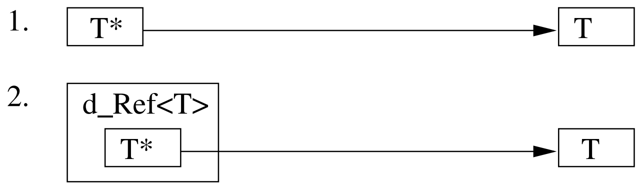


Figure 7.1: Approaches for mapping references to objects

Approach 1 Let applications use a direct pointer into the data memory. One of the major disadvantages of this approach is a complete lack of concurrency control. Locking will have to be done explicitly. When objects are referenced this way, it will never be possible to move objects and accessing already deleted objects will most probably result in a crashing application and/or MMDB.

Approach 2 Applications use a special reference class that contains the pointer to the object. Special actions when using the pointer can be handled by the reference class. When this reference class is a C++ template class, they are completely type safe[1].

Approach 2 will be the only approach supported by Splice2DB (for applications that is) because of the inherent safety. Chapter 8 will use this reference class approach to give a design of a concurrency control mechanism using different type of reference classes.

7.3 Relations

The ODMG object model supports only binary relationships, i.e. relationships between two types. The model does not support n-ary relationships, which involve more than two types. A binary relationship may be one-to-one, one-to-many or many-to-many, depending on how many instances of each type participate in the relationship.

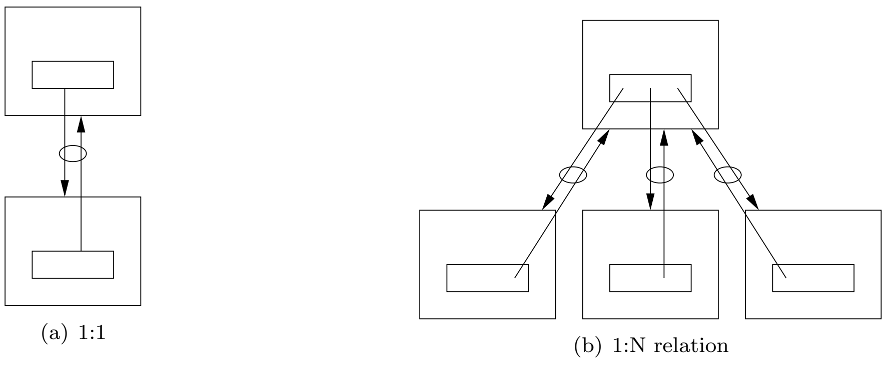


Figure 7.2: Relations

In a one-to-one relationship (see figure 7.2a), setting the relation on one side, sets the inverse relation on the other side (and clears any old relations these two objects may have had). In a

[1]You cannot reference an object of another type than the type the template is specialized with

one-to-many relationship (see figure 7.2b) setting the relation on the one-side, adds the inverse relation to the many side (optionally clearing an old relation on the one side). In a many-to-many relationship, adding the relation to a side, adds the inverse relation to the other side. Removing a relation on a side, removes it from the inverse side too. Thus, relations maintain *referential integrity*.

Direct pointers are the fastest way to access the other object. An object with a to-one relationship has a pointer to the other object. An object with a to-many relationship has a set of pointers to other object. These direct pointers can be encapsulated in smart-pointer classes [Str97].

7.4 Administration

7.4.1 Name-instance administration

The name-instance administration is a part of the database, it associates names with instances.

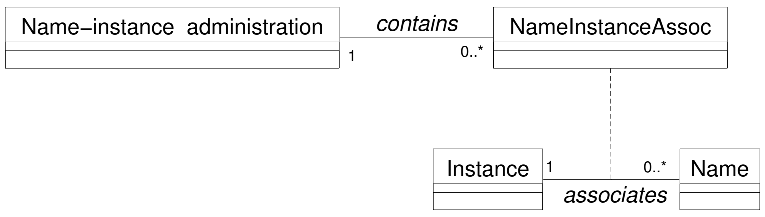


Figure 7.3: Name-instance administration

Representation of objects

- Instances can be represented as direct pointers to instances or OIDs. Direct pointers are faster, since there will be no lookup of an OID necessary, and will be used where possible.
- Names will be represented as strings

Finding

Finding an object in the name-instance administration will always be done by looking up a name.

Definition

Using STL[2] [Str97], this can be defined in C++ as

```
typedef name-instance-adm map< Instance*, set<string> >;
```

[2]Standard Template Library, part of C++ standard

if a pointer to an instance is chosen to be the representation for instances.

This maps pointers to instances to a set of strings. This might be a correct implementation, of the object model given in figure 7.3, but it is absolutely not efficient. With the additional requirement that no two instances can be associated with the same name, and the fact that lookup will always be done by name, the following definition is better:

```
typedef name-instance-adm multimap<string, Instance*>;
```

A map maps keys (strings here) to data (instance pointers here). A map is a unique associative container [Sil96], meaning that no two elements have the same key.

To decrease lookup time it is better to use:

```
typedef name-instance-adm hash_map<string, Instance*>;
```

7.4.2 Extent administration

The extent administration is a part of the database, it keeps track of all instances of a class. The type definer can specify whether the database should keep an extent for a type or not. It contains 1:n relations between classes and instantiations. Thus, the administration associates classes to sets of instances.

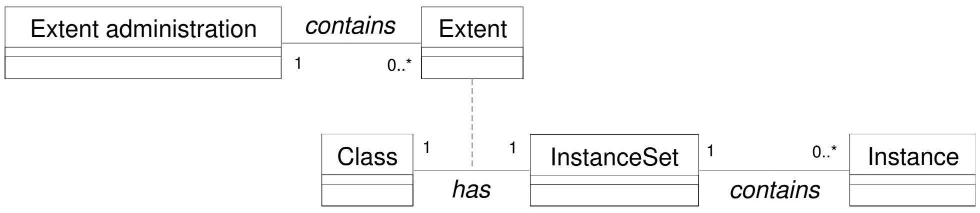


Figure 7.4: Extent administration

Representation

- Classes can be represented by a pointer to their associated class definition in the metadata administration or a MID[3].
- The instances can be represented as direct pointers to instances or OIDs. Direct pointers are faster, since there will be no lookup of an OID necessary, and will be used where possible.

Finding

Finding an element in the extent administration will be done my looking up a MID or pointer to a class definition. Thus, Class is the KEY field and the associated set of Instances the DATA field

[3]Like objects in the database the metadata objects also have unique IDs (MID)

Definition

If pointer to instance is chosen to be the representation of Instance and MID is chosen to be the representation of Class, we can define the extent administration using STL like this:

```
typedef instance-set set<Instance*>
typedef extent-adm map<mid,instance-set>
```

Again, a lookup is faster if a hash map is used:

```
typedef extent-adm hash_map<mid,instance-set>
```

7.4.3 Metadata administration

Details about data representation in the metadata administration can be found in chapter 9.

8 Concurrency control

As identified in the design overview (chapter 6), transactions are an unwanted mechanisme for a MMDB for Splice2. In a distributed environment the price (performance-wise) for achieving distributed transactions is to high for Splice2. Even within a node there is no necessity for transactions. As transactions provide a unit in which the database satisfies the ACID-constraints, another mechanism should take that role. This chapter describes a reference locking mechanism based on references, that satisfies the ACID constraints without using a complete transaction mechanism. The possible level of concurrency increases, especially in multi-processor systems. However, the ability to abort actions or rollback actions is lost. The reference locking mechanism will be explained in section 8.1.

The ODMG-interface defines six ways to get access to objects in the database:

1. Create a new object (section 8.2)
2. Access a collection within an object (section 8.3)
3. Lookup an object by name (section 8.4)
4. Extents (section 8.5)
5. Relationships from an object to another object or other objects (section 8.6)
6. Reference from an object to other objects (section 8.7)

All these six ways must be protected by a concurrency control mechanism.

Applications generally use references to refer to an object that resides in the database. With the new mechanism this has to be the only way applications can access an object.

8.1 Reference locking mechanism

Although modification of an object requires exclusive access, it is acceptable to allow several threads/processes to simultaneously read the data, as long as no one is trying to write to it at that time. This requires a complex lock that permits both shared and exclusive modes of access. These readers/writer locks are part of the Solaris 2.6 thread library. If not present on a system they can be built using *spinlocks* and *condition variables*[Vah96]. Every object in the database will have such a readers/writer lock. This lock will be used by the reference to set the desired access level.

In Splice2DB there will three types of references with an associated lock:

1. A read & write reference for full access to an object (also right to delete) (RW) called `d_RW_Ref`
2. A read-only reference for read-only access to an object (R) called `d_Read_Ref`
3. A keep-only reference for no access to an object (K) called `d_Ref`

The system should be able to upgrade or downgrade these references. The purpose of the third type is to be able to keep a reference while you are currently not accessing it. When access is needed, the reference must be upgraded to the needed access level. In case of a keep-only reference, the object can be deleted without warning.[1] The application will have to check if it is still valid.

Current mode	**Requested mode**		
	RW	R	K
RW	deny	deny	grant
R	deny	grant	grant
K	grant	grant	grant

Table 8.1: Locking compatibility matrix

Table 8.1 shows the locking compatibility matrix. This table in short:

- If a process in the system holds a RW-lock, only a request for a keep-only lock can be granted,
- If no process in the system holds RW-lock, requests for read-only and keep-only locks can be granted,
- If no process in the system holds a RW- or an R-lock, request for all types of locks can be granted.

In the ODMG-standard a reference template class is defined, `d_Ref<T>`, that is used to provide type-safe access to objects in the database. The implementation of the reference locking mechanism will be based on this interface. The most important member functions are:

- constructor `d_Ref()`, an empty constructor, creates a null-reference
- constructor `d_Ref(T* p)`, a constructor that takes an object pointer as argument
- constructor `d_Ref(d_Ref<T>& r)`, a copy constructor
- several assignment and equality checking operators
- `void clear()`, clears the reference
- `T* operator→() const`, returns a pointer to the object, does not change the reference
- `void delete_object()`, deletes the referenced object

In case of a keep-only reference only functions that change the reference will be provided, no access or deletion functionality. In case of a read-only reference the operator → will return a `const T*`. Only a read-write reference will have all the functionality of the original `d_Ref<T>`.

8.1.1 Upgrading and downgrading

Three mechanisms will be supported to up- or downgrade references and their associated locking level:

1. Copy constructor mechanism

[1]This is a matter of choice. It is possible to add a fourth type of reference, or change the keep-only reference, that prevents deletion of the referenced object.

2. Assignment operator mechanism
3. Cast mechanism

Copy constructor mechanism This mechanism creates a new object, copying the reference of its argument. Table 8.2 shows the semantics of the constructor mechanism. When the argument is a locked reference type (RW or R) the arguments gets cleared, i.e. unlocked and the reference cleared. The reference also gets cleared when a read-only reference lock is asked for, with a read-only reference as the argument, even though not clearing the argument will not cause deadlock[2].

Example:

```
d_Ref r1;
d_RW_Ref a(r1);
```

Constructed reference	**Argument**		
	RW	**R**	**K**
RW	DENY	copy referenceunlock Rclear argumentlock RW	copy referencelock RW
R	copy referenceunlock RWclear argumentlock R	copy referenceunlock Rclear argumentlock R	copy referencelock R
K	copy reference	copy reference	copy reference

Table 8.2: Copy-constructor action table

Assignment operator mechanism Using this mechanism, an existing reference can be assigned a new value from another reference. Table 8.3 show the semantics of the assignment operator mechanism. When the argument is a locked reference type (RW or R) the arguments gets cleared, i.e. unlocked and the reference cleared. The reference also gets cleared when a read-only reference lock is asked for, with a read-only reference as the argument, even though not clearing the argument will not cause deadlock.

Example:

```
d_Ref r1;
d_RW_Ref r2;
r1=r2;
```

Cast mechanism This mechanism creates a casted reference from its argument. The cast mechanism does not release the old lock since this mechanism will mostly be used in arguments to function calls. Therefore, a lot of casts are not allowed. Table 8.4 shows the semantics of the cast mechanism.

Example:

```
d_RW_Ref r1;
do_something((d_Ref)r1);
```

[2]This is a matter of choice. The risk of leaving an extra read-lock in the system, is a little smaller this way.

Assigned reference	**Argument**		
	RW	**R**	**K**
RW	DENY	copy referenceunlock Rclear argumentlock RW	copy referencelock RW
R	copy referenceunlock RWclear argumentlock R	copy referenceunlock Rclear argumentlock R	copy referencelock R
K	copy reference	copy reference	copy reference

Table 8.3: Assignment operator action table

Cast to	**Argument**		
	RW	**R**	**K**
RW	DENY	DENY	copy referencelock RW
R	DENY	copy referencelock R	copy referencelock R
K	copy reference	copy reference	copy reference

Table 8.4: Cast operator action table

8.2 Locking of new objects

Applications create objects in the database using a overloaded new operator, e.g.:

```
Track *t = new(&db, "Track") Track(...)
```

Now the application has a direct pointer to the object in memory. The application can now change `t`'s attributes:

```
t→trackID = 4123;
t→trackID = 4123;
```

and assign a name to `t`:

```
db.set_object_name(d_Ref_Any(t),"A track");
```

When an other application request an extent of type Track (the collection of all Track objects) or looks up the 'A track' object by name, it can access the object even while the application that created it has a pointer and can access it too.

This behavior of the `new` operator cannot be changed, as it is the behavior of the built-in C++ `new` operator. It always returns a direct pointer to the type it created.

Therefore, an object has to be locked for reading & writing when it is created. It is the responsibility of the application that created the object to release this initial lock. The three reference classes will unlock the object and re-lock it at the appropriate level when given a pointer to an object. Thus

```
d_Ref<Track> new_ref(t);
```

will unlock `t`, keeping a keep-only reference to `t`.

In general applications will use the reference mechanism by directly assigning the result of the `new` operator to a reference class:

```
d_Ref<Track> tref = new(&db, "Track") Track(...),
```

or

```
d_Read_Ref<Track> tref = new(&db, "Track") Track(...),
```

or

```
d_RW_Ref<Track> tref = new(&db, "Track") Track(...)
```

8.3 Access a collection within an object

A class definition like

```
class Track {
   d_Set<Position> positions;
}
```

contains a collection of `position` objects.

Iterators are used to access collection types ([C$^+$97] & [Str97]). Because iterators define their own operators for returning the elements in the collection type they access, the access level of the composing object does not matter when accessing objects in the collection. The access level however does matter when changing the collection itself, for instance by inserting or deleting an object.

When two applications acquire a read-lock on the same Track-instance, they can both use the interface provided by the collection type. Therefore, access control to the elements in the collection has to be provided by the interface of the collection type.

Concurrency control for accessing the elements in the set will be done with a three types of iterators. These three represent the types of references introduced in this chapter. Each collection type will define three iterator-types. For `d_Set` for example:

- `d_Set<T>::iterator`
  `get_element()`[3] returns a `d_Ref<T>`
- `d_Set<T>::read_iterator`
  `get_element()` returns a `d_Read_Ref<T>`
- `d_Set<T>::rw_iterator`
  `get_element()` returns a `d_RW_Ref<T>`

[3]Or the operator `*`, which is equivalent.

The programmer has to choose which type of access is needed. To access all position objects in track a programmer will have to write:

```
d_Read_Ref<Track> track = db.lookup_object("A track");
d_Set<Position>::read_iterator first_position = track→positions.begin();
d_Set<Position>::read_iterator last_position = track→positions.end();
while(first_position != last_position) {
   do something with *first_position
   first_position++;
}
```

In the ODMG-standard iterators are used in three places:

1. In composite objects (like the `Track-Positions` example above)
2. In extents
3. In 1:N or N:M relations

The second and third case are discussed in section 8.5 and 8.6 respectively.

8.4 Looking up an object by name

When an application asks the database to look up a single object by name,

```
d_Ref_Any track = db.lookup_object("A track");
```

the database returns an unlocked generic[4] reference, a `d_Ref_Any`.

This generic reference cannot be used to access the object it references. The reference has to be upgraded to a specific reference with an associated locking level:

```
d_RW_Ref<Track> track = db.lookup_object("A track");
```

The application now has a RW-lock on the object.

Other options:

```
d_Read_Ref<Track> track = db.lookup_object("A track");
```

for a read-only reference to `"A track"` or,

```
d_Ref<Track> track = db.lookup_object("A track");
```

for a keep-only reference to `"A track"`. This is similar to the generic `d_Ref_Any`, except for the type information.

[4]Not type-safe

8.5 Locking extents

In ODMG, extents are accessed by means of iterators. To guarantee safe access to the objects in the extent, these iterators should not directly return objects or pointers to objects, but references.

Extents will support three types of iterators conforming with the three types of references.

An `Extent<T>` will have three associated iterator types:

- `Extent<T>::iterator`
  `get_element()`[5] returns a `d_Ref<T>`
- `Extent<T>::read_iterator`
  `get_element()` returns a `d_Read_Ref<T>`
- `Extent<T>::rw_iterator`
  `get_element()` returns a `d_RW_Ref<T>`

8.6 Locking of relations

Between objects three types of relations can exist:

1. 1:1 relations, an object of type A relates to one object of type B and vice versa.
2. 1:N relations, an object of type A relates to multiple objects of type B, objects of type B relate to exactly one object of type A.
3. N:M relations, objects of type A relate to multiple objects of type B and vice versa

From the viewpoint of a class-definition, a class can have a *to-one* relationship with another class or a *to-many* relationship with another class, independent of the inverse relation. The interface used to access this relation is therefore a to-one interface or a to-many interface.

8.6.1 to-one relation

Suppose the following (ODMG-like) class definitions

```
class A {
   relationship B b
      inverse B::a;
}
class B {
   attribute integer i;
   //inverse relation of arbitrary type...
}
```

Two objects are created, one object of type A (name `"a1"`) and one object of type B, a relation between them is created. Schematically this looks like in figure 8.1.

Creating the objects and setting the relation has to be done in three steps, a fourth step releases all references:

[5]Or the operator `*`, which is equivalent.

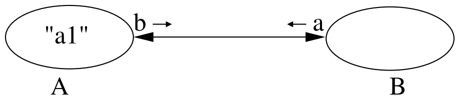


Figure 8.1: to-one relation

1. Create an object of type A and get a RW reference and give it a name
   `d_RW_Ref<A> a_instance = new(&db,"A") A();`
   `db.set_object_name(a_instance,"a1");`
2. Create an object of type B and get a keep-only reference
   `d_Ref<B> b_instance = new(&db,"B") B();`
3. Set the relation between the instances
   `a_instance.b = b_instance;`
4. Release the references (releasing a keep-only reference is not crucial since no guarantees are given for this type of reference)
   `a_instance.clear();`
   `b_instance.clear();`

When another application, or the same application later, looks up the `"a1"` object and needs a RW-lock[6],

```
d_RW_Ref<A> a1 = db.lookup_object("a1");
```

it is possible to access the integer in B like this (in ODMG):

```
a1→b→i = 3;
```

Using the → operator on the relation `b` in `a1`, the elements of the related object can be accessed for reading & writing. This has to be avoided since the related object is not locked by the application. There are two possible solutions:

1. Change the interface of the to-one relation to make it impossible to directly refer to the related object
2. Automatically lock the object that is related by the to-one relation

The first approach changes the interface and makes the use of the database less intuitive. The second approach can potentially lock a lot of objects if there are a lot of to-one relations in the system. In this case ease of use is preferred to locking the related objects.

Automatically locking of related objects creates a possible deadlock problem when to-one relations between objects contain a cycle. Consider objects that relate each other like in figure 8.2. All relations in this figure are to-one relations in both directions.

When object `"a1"` is locked, the system will lock its related B-object, which will lock its related C-object. This C-object tries to lock its related A-object and waits forever to acquire the lock. This automatic locking system should therefore contain a cycle-detection mechanism, which will not let the C-object lock `"a1"`.

[6]By putting the reference to the object found into a d_RW_Ref-class, the application asks for full Read&Write access. The `lookup_object` function returns a unlocked generic reference to an object.

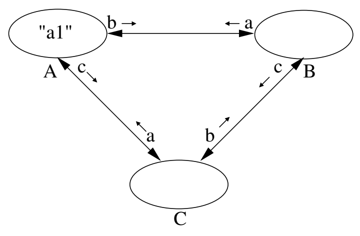


Figure 8.2: 1:1-relations with a cycle

8.6.2 To-many relations

Suppose the following (ODMG-like) class definitions

```
class A {
   relationship set<B> b
      inverse B::a;
}
class B {
   //inverse relation of arbitrary type...
}
```

A series of objects is created and related. Figure 8.3 shows such a structure.

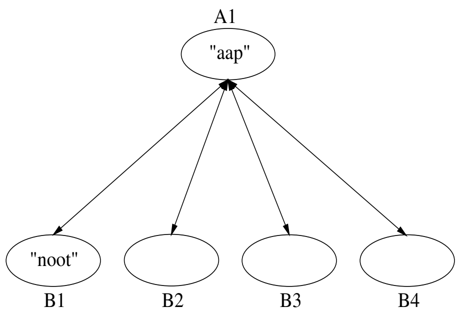


Figure 8.3: 1:N relation

A to-many relation can be seen as a system-maintained collection of references to objects. This type of relation is also accessed using iterators, just like the collection types mentioned in section 8.3. Access control to this special collection type can be done using the same three types of iterators techique as used for the normal collection types (see section 8.3). This way it is not necessary to lock all related objects.

8.7 Locking of references

8.7.1 Locking single references

An object can contain a reference to another object. This reference, unlike a relation, has no inverse reference.

```
class A {
   reference<B> b;
}
class B {
   attribute integer i;
}
```

In the ODMG-standard the same type (`d_Ref<T>`) of reference is used for:

1. references between objects,
2. acces to objects from applications.

This mechanism cannot be used any more, since the single reference type from for applications has been replaced by three types of references. Therefore a separate reference type will be used to reference between objects. The interface will almost be equivalent, however.

Like the interface of a to-one relation, the ODMG-standard defines the → operator to give an application full access to the referenced object. Therefore, all referenced objects have to be locked for reading and writing if the interface remains unchanged.

```
d_RW_Ref<A> a;
a→b.i = 3;
```

Like the deadlock problem identified for to-one relations, this reference mechanism has a potential deadlock problem when the class definitions contain a cycle of references and automatically locking references is done. For discussion of possible actions see section 8.6.1.

8.7.2 Locking collection of references

An object can contain a collection of references to other objects.

```
class A {
   d_Set< reference<B> > b;
}
class B {
   attribute integer i;
}
```

Like accessing a collection of objects within a class , the acces to a collection of references is done using iterators. Therefore, the same three types of iterators can be used. For furthur discussion of this mechanism see section 8.3.

9 Schema Access & metadata

A database system maintains a *schema*, a description of all the objects that can be stored in the database. The schema must include a complete description of each type stored in the database. This information includes relationships, operation prototypes and exceptions.

Because of the rich modeling capabilities of object-oriented databases, the amount of meta-information needed is much bigger than within traditional database systems, such as relational databases. This data, called metadata, is used internally to calculate object sizes, to check query parameters, maintain extents, etc. Some OO-languages support meta-objects or meta-classes, describing the classes declared in the applications. C++ does not have a standard representation for meta information. Most of the information defined in a C++ class must be represented in the database as metadata.

ODMG 2.0 [C$^+$97] defines a set of classes that provide access to the schema in an object database. In order to communicate on metadata-level with the database system two interfaces are needed:

- An interface to feed the database with new metadata
- An interface to read the schema of the database for use in database development tools, distribution systems and management tools

The C++ metadata interface defined by the ODMG 2.0 standard [C$^+$97] is a read-only interface. Extensions to the standard have been planned to include a full read/write interface, to enable programmers to dynamically modify the schema of the database. In this chapter a description is given of the generation of metadata, the representation of metadata in the system and the schema access from applications.

9.1 Metadata generation

The schema definition of a C++ application is most often based on parsing the application header files. The database system must acquire the metadata by parsing C++ code. However, several approaches exist [Jor98]:

1. A schema definition language parser generates both the schema and the C++ header files
2. A preprocessor examines original C++ header files, capturing the schema and sometimes producing new header files to be used by the application
3. A postprocessor examines the symbol table information produced by the compiler, eliminating the need to parse C++ source code[1]

[1]Parsing C++ code might be cumbersome, due to differences in C++ language interpretations by compiler builders

Both parsing C++ code and examining object files are not language-independent. The ODMG 2.0 standard [C+97] defines C++, Java and SmallTalk bindings for ODMG-compliant databases. Generating schema information from code or object files eliminates the possibility of using one definition of classes within several programming languages. Splice2DB will be used from applications written in various languages, therefore the first approach is chosen. ODMG 2.0 defines an object definition language, ODL. Splice2DB will use a subset of ODL, called SODL. The definition of this language is given in chapter C in the appendix.

Thus, the metadata is generated from one SODL input file, by a preprocessor which produces two products:

- Definitions to be included in the application source.
- metadata to be included in the database

Figure 9.1 gives an overview of the applications and files involved. To create a new application a preprocessor, a compiler[2] and a run-time linker are needed. The compiler and run-time linker are standard tools, the preprocessor is a dedicated tool, that can only be used for one OODB implementation.

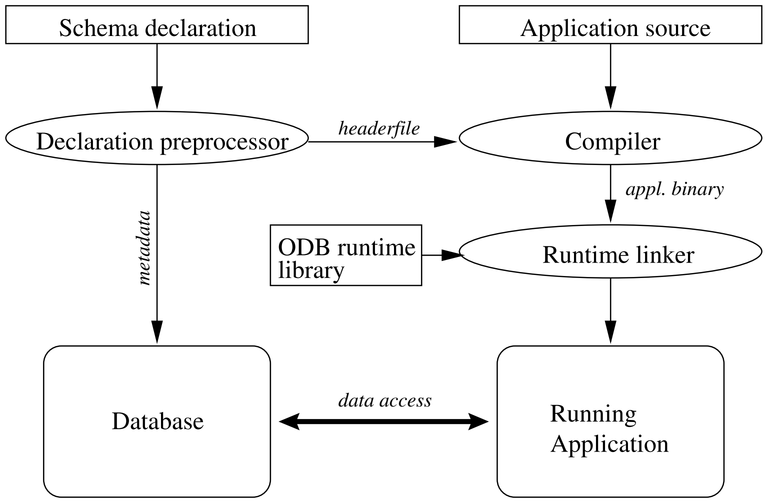


Figure 9.1: metadata flow

In figure 9.1 metadata flows directly from the preprocessor to the database. Logically this is correct, since only the database uses the metadata created by the preprocessor. However, two approaches are possible

- Generate separate application header files and metadata files.
- Generate one application header files which contains both class definitions to be used in the application and metadata to be used by the database.

The advantage of the latter approach is the possibility, to create and start new applications, without having to feed the metadata into the database separately. The generated metadata has to be valid C++ code in this case.

[2]For simplicity, it is assumed that the compiler generates a complete executable

9.2 Representing metadata

Metadata, in the ODMG standard, is described using a set of classes that can be accessed from every application using the database.

9.2.1 Meta objects & scopes

Every class except `d_Scope` in the metadata hierarchy, is derived from `d_Meta_Object`. This `d_Meta_Object` class has a name, a comment and is defined in a scope.

`d_Scope` instances are used to form a hierarchy of meta objects. A `d_Scope` contains a list of `d_Meta_Object` instances that are defined in its scope, as well as operations to manage the list. The method `d_Scope::resolve` is used to find a `d_Meta_Object` by name.

All instances of `d_Meta_Object`, except `d_Module`, have exactly one `d_Scope` object[3]. This represents a 'defined in' relationship. The type `d_Scope::meta_object_iterator` defines a protocol to traverse this relationship in the other direction.

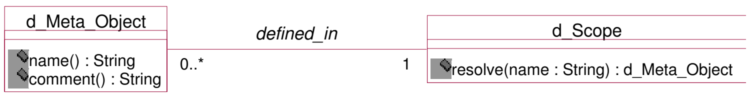


Figure 9.2: `d_Meta_Object` and `d_Scope`

9.2.2 Module

Every application that uses the database has to define a top level module containing definitions of constants, types and operations.

A `d_Module` contains lists of:

- `d_Constant`, constant definitions
- `d_Type`, type definitions
- `d_Operation`, operation definitions

that are defined in its scope. `d_Module` provides an interface for accessing these definitions by means of iterators. Using these iterators it is possible to navigate further down in the hierarchy. For example, from `d_Operation`, the set of `d_Parameter` instances can be reached and so on.

The ODMG standard text does not comply with their class definition for the C++ language binding. Modules cannot be defined within modules, because there is no interface to browse sub-modules.

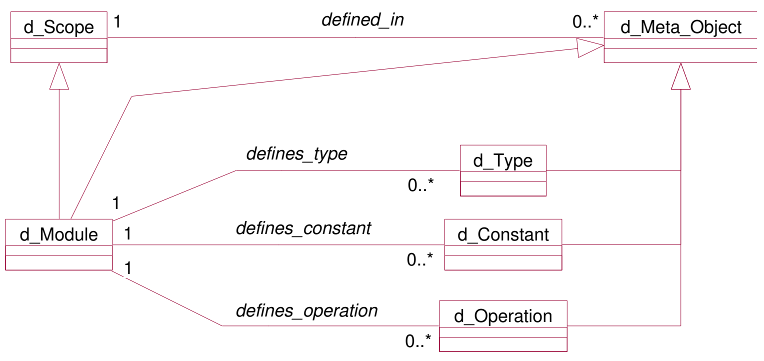


Figure 9.3: `d_Module`

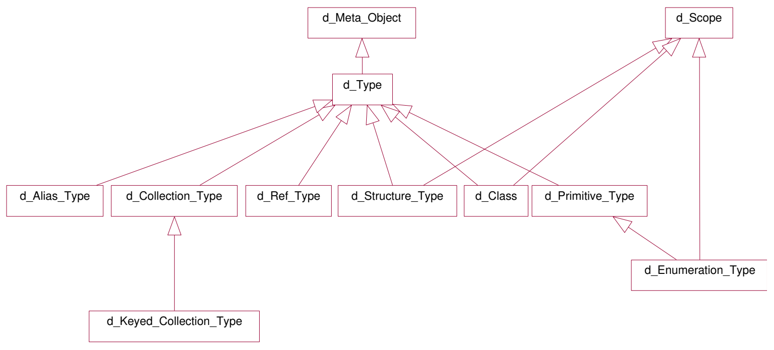


Figure 9.4: Hierarchy of `d_Type`

9.2.3 Types

Figure 9.4 shows the inheritance tree of all classes derived from `d_Type`.

In short:

`d_Alias_Type`	Another name for a type, like a typedef
`d_Collection_Type`	For collection types like set, list, bag
`d_Keyed_Collection_Type`	For collection types that map keys to values, like a map or a dictionary
`d_Ref_Type`	References to other objects
`d_Structure_Type`	Struct definition
`d_Class`	Class definition
`d_Primitive_Type`	Definition of built-in types like int, char and double
`d_Enumeration_Type`	For enumerations, which are of a certain `d_Primitive_Type`

[3]The top level scope is a `d_Module`. This top level module has no 'defined in' relation with a `d_Scope`.

Only `d_Class` and `d_Primitive_Type` will be explained here. For more complete information about the ODMG meta-classes see [C$^+$97] and [Jor98].

d_Primitive_Type

An instance of d_Primitive_Type represents a primitive type. Table 9.1 shows an overview of available primitive types and their properties as defined by ODMG. The column 'Implement as' shows a possible C++ type definition.

Type	**Size**	**Description**	**Implement as**
`d_Char`	8 bit	ASCII character	`char`
`d_Octet`	8 bit	No interpretation	`unsigned char`
`d_Boolean`	undefined	`d_False` and `d_True`	`bool`
`d_Short`	16 bit	Signed short integer	`short`
`d_UShort`	16 bit	Unsigned short integer	`unsigned short`
`d_Long`	32 bit	Signed integer	`long`
`d_ULong`	32 bit	Unsigned long integer	`unsigned long`
`d_Float`	32 bit	IEEE Standard 754-1985 single-precision floating-point	`float`
`d_Double`	64 bit	IEEE Standard 754-1985 double-precision floating-point	`double`

Table 9.1: Overview of available primitive types

d_Class

Every class that is to be stored in the database is represented by a `d_Class` instance in the metadata administration. For classes involved in an inheritance relationship, a `d_Inheritance` instance is used to associate the two classes. A `d_Class` contains lists of:

- `d_Inheritance`, baseclasses
- `d_Inheritance`, subclasses
- `d_Operation`, operation definitions
- `d_Attribute`, attribute definitions
- `d_Relationship`, relationship definitions
- `d_Constant`, constant definitions
- `d_Type`, type definitions

d_Property

The `d_Property` class is an abstract base class for `d_Attribute` and `d_Relationship`. A property is a field of a class or a structure.

Figure 9.6 show the class diagram of `d_Property`. `d_Property` is derived from `d_Meta_Object` and therefore has a name and a comment. A property is of a certain type and has an access kind

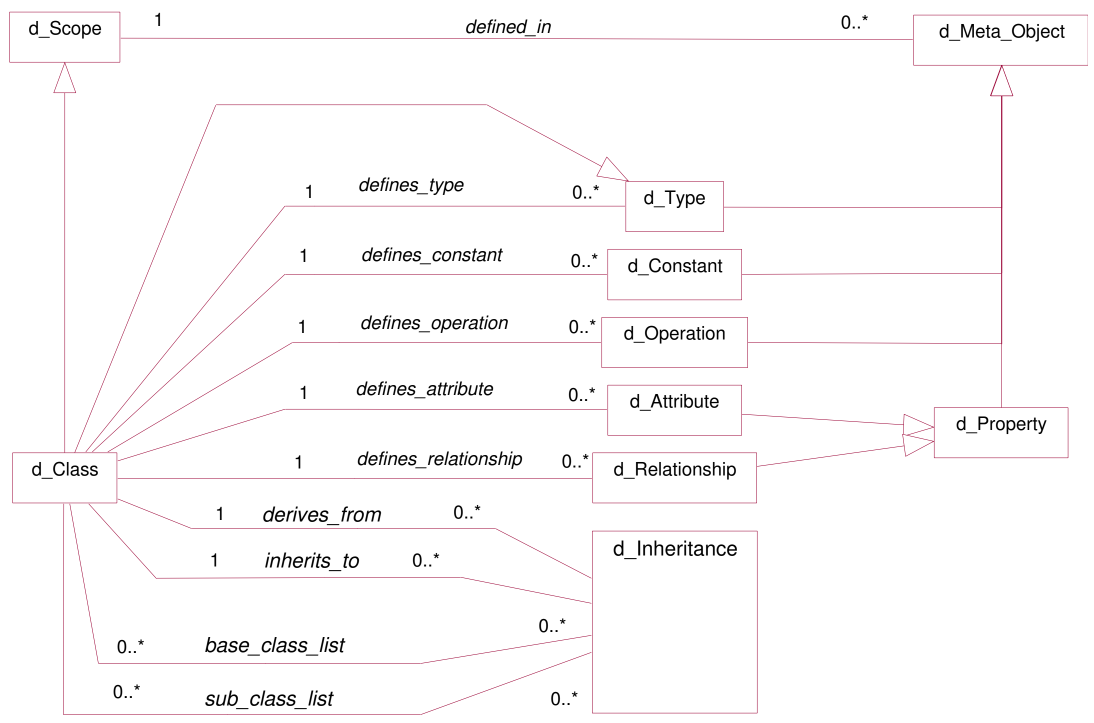


Figure 9.5: `d_Class`

(`private`, `protected` or `public`). Only properties that are defined in a `d_Class` can have an access kind other than `public`.

A d_Attribute has the following additional methods:

- is_static, returns true when this is a static attribute
- is_read_only, returns true when this is a read-only attribute
- dimension[4], returns the dimension of the attribute

9.3 Creation of metadata by applications

Applications have to have the metadata resident when they open a database. This metadata can be present in the form of descriptor strings or in a form that resembles the metadata administration in the database. Because there already is a parser creating C++ header files from SODL files, a structure resembling the metadata administration can be created. An advantage of this approach is that the database does not need to have a parser for SODL-strings.

Figure 9.7 shows a sequence diagram of an application creating a metadata structure. The steps are:

[4]The dimension is the number between the square brackets in an attribute definition like: `attribute int twoints[2];`

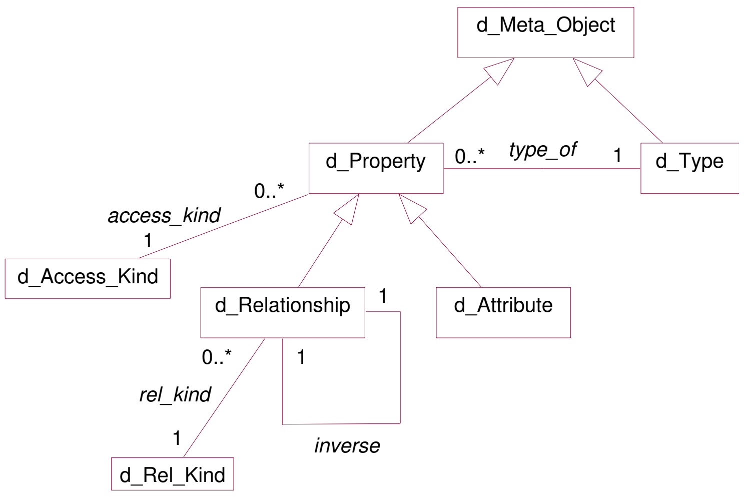


Figure 9.6: `d_Property`

1. Create an attribute
2. Create a class containing the attribute from step 1
3. Create a top level module containing the class from step 2

The methods show in figure 9.7 are simplified, all meta objects have to be given a name, classes can contain much more than attributes only etc.

9.4 Schema Access

The C++ Metadata interface defined by ODMG 2.0 is a read-only interface. Extensions to the standard have been planned to include a full read/write interface, to enable a programmer to dynamically modify the schema of the database.

Figure 9.8 shows the complete inheritance overview of the metadata representation as defined by ODMG 2.0.

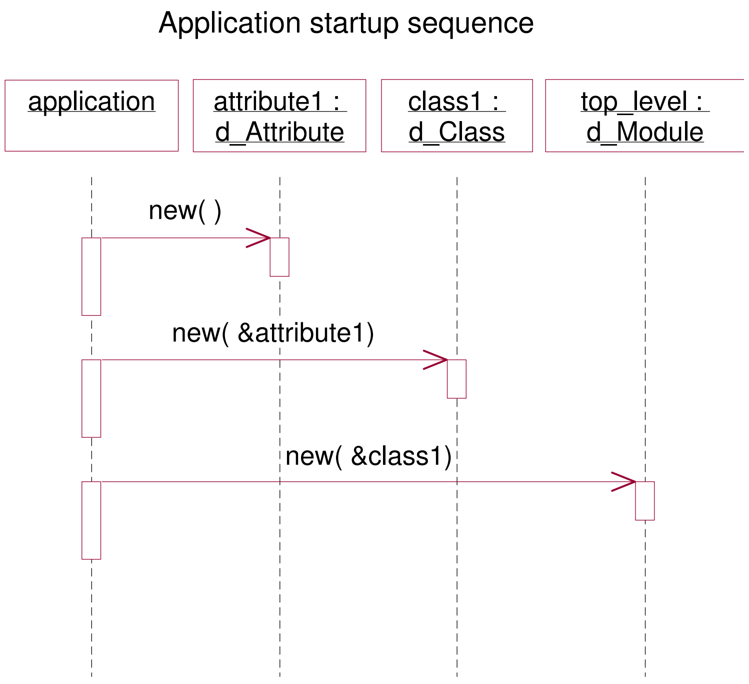


Figure 9.7: Application creates metadata

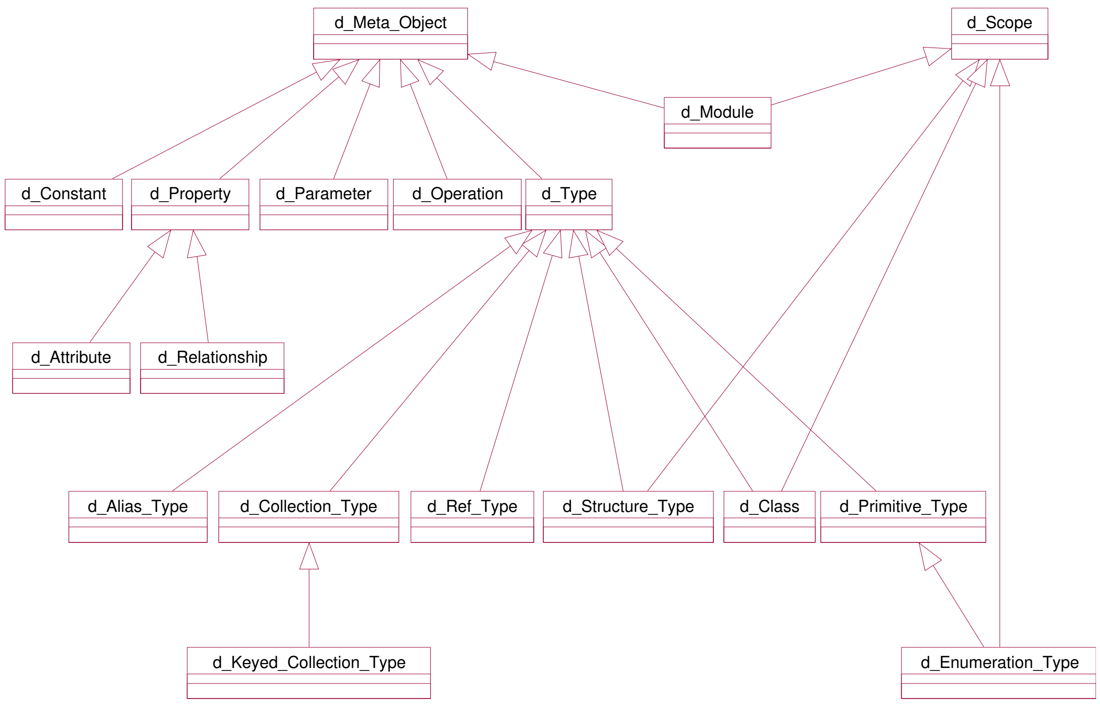


Figure 9.8: Overview of metadata class hierarchy

Part III

Implementation

10 Structure

The prototype for Splice2DB has been divided into several packages that together provide the Splice2DB functionality. An overview of the packages is given in section 10.1. The relations between the packages will be discussed in section 10.2.

10.1 Packages

The Splice2DB library consists of four packages:

1. ODMG data interface package
2. ODMG metadata interface package
3. Memory management package
4. Synchronization primitives package

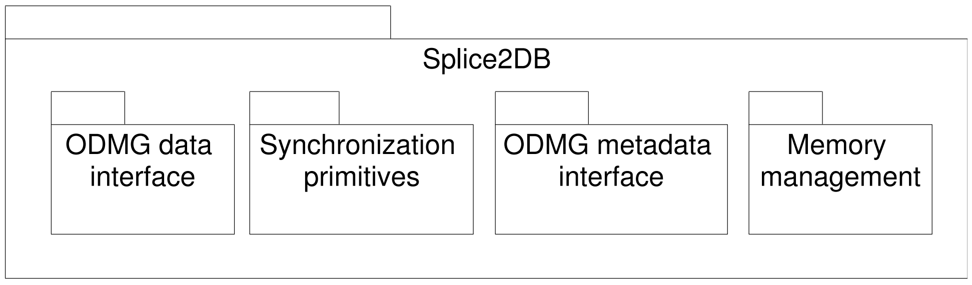


Figure 10.1: Splice2DB packages

The total Splice2DB library is implemented in the namespace[1] `odmg`.

10.1.1 ODMG data package

Provides the implementation for the data access interfaces defined by ODMG. The following classes are implemented here:

[1]namespaces are part of the new C++ standard, see [Str97]

`d_Database`

`d_Extent<T>`

`d_Object`

`d_RW_Ref<T>`

`d_Read_Ref<T>`

`d_Ref<T>`

`d_Ref_Any`

`d_Set<T>`

`d_Rel_Ref<T,MT>`

`d_Reference<T>`

The following classes from the ODMG data access interface are not implemented yet:

`d_Rel_List<T,MT>`

`d_Rel_Set<T,MT>`

`d_Collection<T>`

`d_Bag<T>`

`d_List<T>`

`d_Varray<T>`

`d_Dictionary<K,V>`

`d_Error`

`d_OQL_Query`

The following class will not be implemented:

`d_Transaction`

`d_Iterator<T>`

`d_Iterator<T>` will not be implemented because not all collection types can use the same iterator interface, because for some collection classes a three-level iterator mechanism is used and the STL collection classes cannot use one iterator type because of performance constraints [Sil96]. Instead, each collection type defines its own iterator type, like: `d_Extent<T>::iterator`.

10.1.2 ODMG metadata package

Provides the implementation for the metadata. The following classes are implemented here:

`d_Attribute`

`d_Class`

`d_Meta_Object`

`d_Module`

`d_Property`

`d_Relationship`

`d_Scope`

`d_Type`

The following classes from the metadata are not implemented yet:

`d_Alias_Type`

`d_Collection_Type`

`d_Constant`

`d_Enumeration_Type`

`d_Exception`

`d_Inheritance`

`d_Keyed_Collection_Type`

`d_Operation`

`d_Parameter`

`d_Primitive_Type`

`d_Ref_Type`

`d_Structure_Type`

10.1.3 Memory management package

This package provides wrapper classes for the C[2] based shared memory claimer and memory allocator mentioned in chapter B.

`Spl_Shm`

`Spl_mm`

10.1.4 Synchronization primitives package

This package provides a wrapper classes for operating system dependent synchronization primitives. Currently it only contains a wrapper for the Solaris `rw_lock`.

`d_RW_lock`, an abstraction of a readers-writers lock

10.2 Interrelation

Packages are dependent on other classes. Figure 10.2 show the dependencies between the packages in Splice2DB.

Both the ODMG data interface and the metadata interface use the memory management routines. Only the ODMG data interface uses the synchronization primitives.

[2]the language

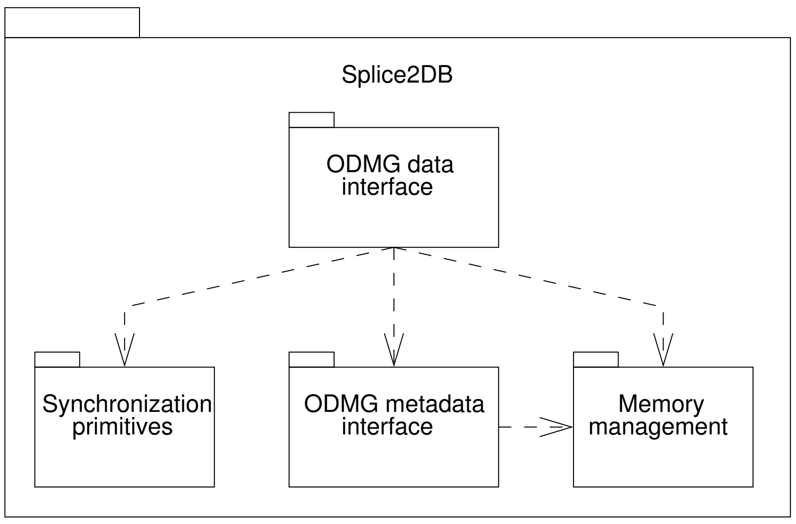


Figure 10.2: Splice2DB package overview and dependencies

11 Creating metadata

Metadata will be resident in a started application in a structure that resembles the structure in the metadata administration. This approach was chosen in chapter 9. This chapter will show how this is implemented in Splice2DB. It will show what information has to be created by a SODL to C++ parser. Because of the size of the metadata API only some examples are given. The rest of the documentation is in the documentation generated from the library source.

11.1 From SODL to C++

Suppose the following class definition in SODL:

```
class example {
   attribute long attr;
};
```

The parser will have to create a C++ class definition and metadata for the database.

11.1.1 C++ class definition

Creating a C++ class definition from the example is straightforward:

```
class example :  public d_Object {
public:
   long attr;
   static const char * const class_name;
}
const char * const example::class_name = "example";
```

The `example` class extends from d_Object to be able to be stored in the database. The parser adds a static member[1] `class_name` to the class definition. This identifies the object to the database in some special cases where type information is needed. A static member has to be initialised outside the class definition, and should be done only once in an application.

11.1.2 C++ metadata

The parser has to generate metadata for the attribute and for the class. Finally, the parser has to generate a module containing the definitions.

[1]A static attribute is shared along all instances of the class. It does not occupy space in instances of the class.

Attribute

```
d_Attribute *__example_attr = {
   new d_Attribute("attr",
                   "",
                   "LONG",
                   offsetof(example, attr),
                   false, false,1),
   0};
```

The parser generates code for a `d_Attribute`-pointer array containing all attribute definitions for the `example` class. The last element in the array is a NULL-pointer[2] to indicate the end of the defintions. The constructor for an attribute takes the following arguments:

1. `const char* name`, the name of the attribute
2. `const char* comment`, an optional comment
3. `const char* type`, the name of the type of the argument
4. `size_t offset`, the offset of the attribute in the class
5. `const bool is_read_only=false`, whether this is a const-member
6. `const bool is_static=false`, whether this is a static member
7. `const unsigned int dimension=1`, the dimension of the attribute[3]

Class

```
d_Class __example ("example","",0,0,0,
              __example_attr,0,0,0,0,
true,true,false);
```

The parser generates code for a `d_Class` containing the attribute defined above. The constructor of `d_Class` takes the following arguments:

1. `const char* name`, name of the class
2. `const char* comment`, an optional comment
3. `d_Class* subclasses[]`, list of derived classes
4. `d_Class* baseclasses[]`, list of base classes[4]
5. `d_Operation* operations[]`, list of operation
6. `d_Attribute* attributes[]`, list of attributes
7. `d_Relationship* relations[]`, list of relations
8. `d_Constant* constants[]`, list of constants
9. `d_Type* types[]`, list of types
10. `bool persistent_capable=true`, whether the class is persistent capable[5], this is useful classes that are part of other classes and don not have to be persistent capable by themselves
11. `bool has_extent=true`, whether the database should keep an extent for this class
12. `bool forward=false`, whether this class should be declared forward

[2]In the new C++ standard the number `0` is used to indicate a NULL-pointer.
[3]`attribute int[4] ints;` specifies an int-array with four elements.
[4]In single inheritance this is only one class
[5]i.e. can be stored in the database

Module

```
d_Type* __top_level_types[]={&__example,0};
d_Module top_level("top_level","",__top_level_types,0,0);
```

First the class type-definition is put into an array of type definitions. Then a top level module is created, using this array of type definitions.

The constructor of `d_Module` takes the following arguments:

1. `const char* name`, a name (does not matter actually for the top level module)
2. `const char* comment`, an optional comment
3. `d_Type* types_begin[]`, a list of type definitions
4. `d_Constant* constants_begin[]`, a list of constant definitions
5. `d_Operation* operations_begin[]`, a list of operation definitions

11.2 An application framework in C++

Applications are typically composed of several source files to create one executable. Because multiple source files may need the class definitions defined using SODL, these files should include a header-file containing the class definitions. The metadata, however, may only be defined once. The same goes for definitions[6] of static class members . Therefore, the class definitions and the metadata must be split into two header files. Figure 11.1 shows a possible source tree consisting of one main C++ file and two extra C++ files. Only the main C++ file includes the metadata. Every C++ file includes the class definitions.

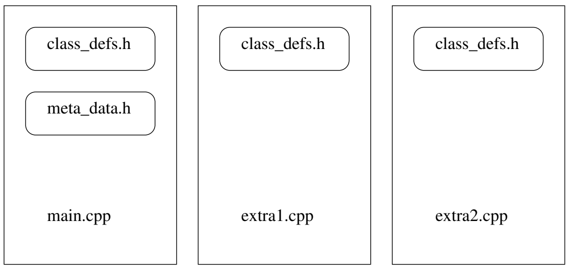


Figure 11.1: An application using Splice2DB

[6]Static class members are *declared* inside a class, but have to be *defined* outside a class.

12 Polymorphism and late binding

One of the main advantages of object-oriented languages is the possibility to defer binding of function-calls to run-time. This allows for polymorphic behavior of classes. In C++ a function whose binding is deferred to run-time, is called a virtual function. This mechanism leads to a problem when used in conjunction with shared memory.

12.1 Virtual functions

Derived classes can redefine functions declared in a base class, if a function is declared to be *virtual* in the base class. Now every derived class can override the function.

An example in C++ code:

```
class Screen {
   virtual void DrawWindow(...)=0;
}
class X11Screen :  public Screen {
   virtual void DrawWindow(...)  { /*implementation*/ };
}
class DirectScreen :  public Screen {
   virtual void DrawWindow(...)  { /*implementation*/ };
}
```

Class `Screen` is an abstract class. By specifying **=0** behind the function declaration, a function is defined to be a *pure virtual function,* i.e. it has no implementation. A class containing one or more pure virtual functions is an abstract class [Str92]. Instances of the `Screen`-class cannot be created since it is abstract, it can only be used as a base class for other classes. If a derived class does not implement all pure virtual functions from its base classes, it is still an abstract class. Both `X11Screen` and `DirectScreen` inerhit from `Screen` and implement the `DrawWindow(...)` function and therefore are not abstract classes. Instances can be created from both `X11Screen` and `DirectScreen`.

12.2 Late binding

Consider the following program fragment:

```
Screen* scrPtr;
if(...)  {
   scrPtr = new X11Screen;
}
else {
   scrPtr = new DirectScreen;
}
scrPtr→DrawWindow(...);
```

It is not possible to determine at compile-time whether `scrPtr` points to an object of type `X11Screen` or `DirectScreen`. Determining which implementation of `DrawWindow` to choose, should be done at run-time. This is called *late binding.*

12.3 Virtual function tables

Because C++ objects normally do not carry around type-information[1], compilers have to add extra information to an object in order to choose the right function at run-time. This is mostly done by adding a pointer to a *virtual function table* to the end of object. Figure 12.1 shows the layout of the objects if in the code fragment above the '`else`'-branch was taken.

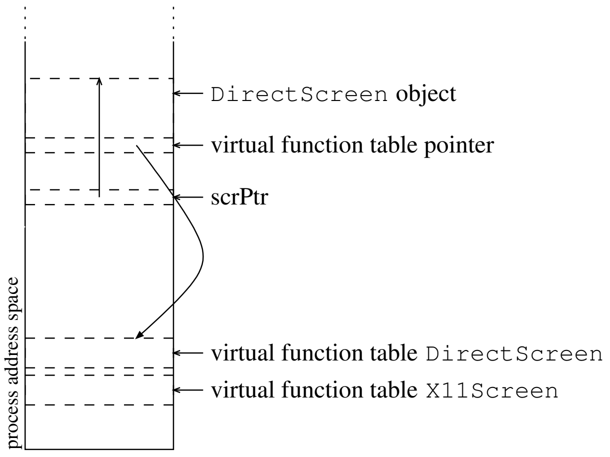

DirectScreen object
virtual function table pointer
scrPtr
virtual function table DirectScreen
virtual function table X11Screen
process address space

Figure 12.1: Screen application object layout

Virtual function tables contain memory addresses of all the functions defined *virtual* in the class. In the example there is only a pointer to the implementation of the `DrawWindow` funtion in both tables. Two instances of the same class point to the same virtual function table .

12.4 Virtual function tables and shared memory

In a main memory database an application opens a shared memory region and attaches this region to its own address space. All objects the application creates in this memory area, contain a virtual

[1]That is, if the new C++ feature Run-Time Type Information is not used.

function table pointer that points to a virtual function table in its own address space, most possibly different from the address in other executables. This is because a linker can map these virtual function tables to a different position in the executable. As shown in figure 12.2 two instances of X11Screen in shared memory contain pointers that are only valid in the process that created the object. In a process that did not create the object, the pointer points to an address which most possibly does not contain valid machine-code or is an illegal address, thereby crashing the program.

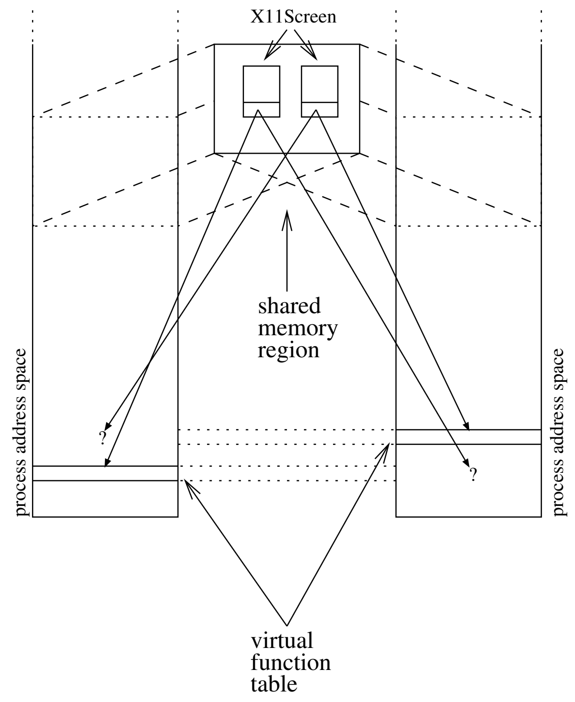


Figure 12.2: Virtual function tables and shared memory

The whole idea of a database is to be able to share objects between applications. There are three possible solutions:

1. Circumvent virtual functions (not really a solution)
2. When a process tries to access an object, activate the object within that process's address space
3. Map all virtual function tables at the same address in all processes that use it

The third solution can be called the 'trivial' solution, because a different mapping of function tables created the problem.

Circumvent virtual functions

Circumventing virtual functions is more a workaround than a solution. It is an 'if everything else fails' option. The type of an object can be identified by adding a type field to every object. A dispatch function in the base class with a switch on the type, can call the correct function. This implies that a base class has to know of all its descendants. This is a very common mechanism in non object-oriented systems, but is hardly preferable in object-oriented systems.

Object activation

An object can be copied from shared memory to a local cache within the process, thereby replacing the virtual function table pointer with a pointer that is correct within the process's address space. Now the application can use the object. Copying an object to a local cache and copying it back if changed is slow, and must be avoided in high-performance systems. This activation mechanism diminishes the performance advantages of a main memory database. Figure 12.3 shows the necessary action in an object activation scheme. It is also possible to change the virtual function table pointer in the shared memory without caching the object. However, this mechanism only allows for one simultaneous reader because the virtual function table pointer is only valid for one process.

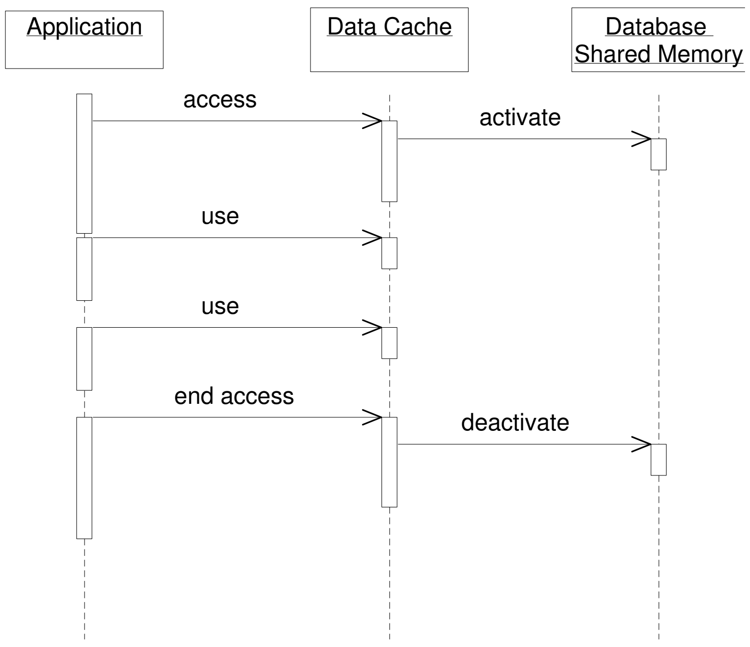


Figure 12.3: Activating objects in a local cache

Map virtual function tables at the same address

This mechanism needs linker support. The linker should be able to map all virtual function tables to a fixed address within all Splice2DB applications. This way, all virtual function pointers point to the same address, which is the start of a correct virtual function table. Two steps are required:

1. Collect all virtual function tables from all source files into one object file
2. Map the contents of this file tothe same address within all executables

Modern C++ compilers like GCC 2.7 and up and Solaris CC 4.0 and up support the first step. The Solaris and the GNU linker support the second step, although the needed map-files are very different.

This approach does not create any overhead like the other two options and is therefore tested for Splice2DB. A map-file for the Solaris linker looks like:

```
vtable = LOAD V0x80000000;
vtable :  .rodata :  tables.o;
```

This map-file maps all entries in the `.rodata` segment in the `tables.o` object file to the `vtable` segment. This is a segment that is defined in this map-file, which will be loaded by the system loader at virtual address `0x80000000`. The syntax of map-file is described in [Sun98].

Every application that uses Splice2DB has to be linked using the same map-file.

This approach has two disadvantages

- It is very compiler and linker dependent. If the compiler does not output its virtual function tables to the `.rodata` segment (which happened in the upgrade from GCC 2.7.2.1 to GCC 2.95) the mapping is not performed.
- Debugging code that faults because of an incorrect virtual function table pointer is very hard, because the offending action cannot be seen in a symbolic source-level debugger.

Part IV

Results

13 Results

13.1 Analysis phase

Aside from gaining knowledge about the current SPLICE system and main-memory database systems, emerging database standards were investigated to find a standard that would be used as a starting point for the Splice2DB system. The ODMG 2.0 standard has been chosen. The reasons for this choice are given in paragraph 3.3. The most important considerations were:

- Strong integration with host programming language
- Ability to store complex data models

13.2 Design phase

In the design phase serveral choices have been made:

- How to represent data and provide access to the data (chapter 7)
- How to handle concurrency control (chapter 8)
- How to generate and represent meta-data (chapter 9)

The following three paragraphs will elaborate on these choices.

Data representation

Based on Signaal's performance demands and C++'s support for smart pointers, smart pointers have been chosen. The smart pointer approach was stretched to its limits by the inclusion of concurrency control in this mechanism. This mechanism is elegant and easy to use.

Concurrency control

Especially in the area of concurrency control the biggest differences from standard client/server databases are noticeable. Chapter 6 discussed the disadvantages of a transaction system for real-time systems. A new mechanism was designed that does not support the full ACID semantics, but is able to prevent race conditions. This system was designed to be used in conjuction with special language features of C++ to be as unobtrusive as possible (chapter 8). The burden of concurrency control is mostly shifted towards features of the C++ language, for a lesser part to the programmer. The main reasons for this approach are:

- Increase in developer productivity
- Decrease in programmer errors caused by forgotten unlock operations

Meta-data generation and representation

In chapter 9 mechanisms have been chosen how to generate metadata and how to represent it in Splice2DB. Based on Signaal's demands a parser mechanism was chosen that generates language dependent interface files (header files for C++) from a language independent meta-data representation (SODL). This language independent representation is easy to read because it does not depend on C macros and compiler trickery. Also, it allows for easy translation to other languages than C++.

SODL was designed to be a subset of ODL. Major differences between SODL and ODL are limitations to enable direct translation to C++. See appendix C.

13.3 Implementation phase

During implementation emphasis was placed on the following functionality:

- Shared memory support
- Meta-data support
- Concurrency control

Shared memory support

Implemented is the ability to place database elements in shared memory by overriding the standard C++ memory manager. The ability to share memory segments seriously reduces memory needs, but is not without challenges. These challenges have been researched and resolved. See paragraph 12.4. However, mixing C++ virtual function tables and shared memory still remains a dangerous practice and should be avoided where possible.

Meta-data support

A mechanism was implemented to allow meta-data to be represented in C++ code. This allows SODL definitions to be translated to C++ and be included in compilation and linking stages of an application. See chapter 11.

Concurrency control

The mechanism designed in chapter 8 was implemented and tested successfully.

Implementation status

Chapter 10 and in particular the paragraphs 10.1.1 and 10.1.2 show the status of the implementation.

14 Conclusions

Standardization

As stated in the introduction to this thesis, Signaal had a strong desire to adopt standard API's. This influenced the decision to use the ODMG API as the API for Splice2DB instead of designing a new API. However, parts of the ODMG interface had to be changed to adapt to Signaal's requirements. This is off course not the idea of using a standard interface, but Signaal's requirements are far from standard.

It is therefore not surprising that an interface defined for a client/server model using transactions does not suffice. It is safe to say that this API is a *mismatch* for Signaal's requirements, because changing just parts of the API renders the original goal of interoperability with COTS systems impossible. You simply cannot compare the requirements for Splice2DB with the features of a COTS object DBMS.

Concurrency control

A transaction system conforming to the ACID requirements as specified by the ODMG 2.0 specification does not match Signaal's requirements. Therefore, a light-weight unobtrusive concurrency control mechanism has been designed. This mechanism strongly relies on the available features of the C++ language such as smart pointers and const pointers.

The combination of the C++ smart pointer approach and concurrency control allows for code that is easy to read and write:

- the strongly typed smart pointers show what kind of access this line of code has at a glance,
- the smart pointers allow for C++ style access to attributes, using the `->` notation.

Meta-data

Splice2DB needs a lot of knowledge about the structure of the objects stored. Splice2DB needs this data to manage relationships between objects and to provide run-time insight in available data definitions, in particular to support ODMG's read-only meta-data API.

The implemented meta-data structure allows applications to provide the meta-data of the objects stored to Splice2DB. Aside from the needs of Splice2DB itself, applications can get insight into the structures of the data stored in Splice2DB without the need of link-time inclusion of this data. Having meta-data available at run-time also opens the possibility to interface with more dynamic scripting languages like Python and Rails.

Performance

Performance, and low latency especially, is crucial to Signaal. The following design elements contribute to Splice2DB's low latency:

- Memory for database objects can be allocated directly in shared memory. This prevents unnecessary copying of memory blocks between application memory and database memory.
- Direct access to data. After having gained the rights to access an object, the attributes of the object can be accessed directly without any indirection.

15 Recommendations

Design a better API

The ODMG API is not a perfect match for a low latency main memory database. As the entire transaction system was removed the API compatibility is gone. The transaction system is so crucial to using the ODMG API, there is no reason left to strictly use the rest of the ODMG API.

It is therefore recommended to investigate whether other parts of the API can be changed to better facilitate the needs of Splice. Possible research topics:

- Collections like historic collections (e.g. a collection that can keep the last hour of data)
- Direct support for data quality in the database like Splice

Separate data from behavior

Some object-oriented databases use the host programming language to define the data dictionary. As class definitions in programming languages combine data and behavior definitions, the definition in the programming language contain more information than the data dictionary alone.

The ODMG 2.0 defines the ODL language as a programming language independent object data definition language. SODL was created to be a subset of ODL to ease the translation to C++ (and thereby to most object-oriented programming languages).

There are several features of (S)ODL tat cannot be expressed (directly) in a programmig language. These are:

- The ability to define key attributes
- The ability to define extents (collection of all instances of a class)
- The ability to define 1:N or N:M relations as collections like set, list & bag[1]

Research shared memory allocation and C++

When allocating a C++ object in shared memory, the compiler adds an extry entry (usually at the end) to the object in memory. This entry points to a virtual function table (see paragraph 12.4). Even when the tables are mapped to the exact same address in memory, this solution is very brittle. This solution requires that *all* applications and libraries that use Splice2DB use exactly the same compiler (both make *and* version).

It is therefore recommended to research the possibility to split the object between a C++ 'holder' object and a pure data object.

[1]Most frameworks contain definitions for these collections but they are usually not part of the language definition.

Consider using C

To support scripting languages that have the ability to add types at run-time, the availability of meta-data is a must. However, these languages tend to have the ability to bridge to libraries written in C, not C++. If support for these languages is important, consider providing a C based API. Possibly, in addition to the C++ API which can be built on top of the C API.

16 Retrospect

All work for this thesis was done in 1999. For the reasons explained in the preface, this report is finished in 2007. This chapter describes developments over the years since early 1999, that have a relation to the Splice2 project and the work done for this thesis.

16.1 Changes to the Splice2 project

During the assignment Signaal changed its mind about the Splice2 project. The more ambitious requirements have been removed and were largely reduced to a serious rebuild and refactoring of the existing SPLICE code base.

- Splice2 will not be implemented in C++ but in C.
- Splice2 will not support any object-oriented features on the API level
- Splice2 will have support for object-oriented features internally, to enable additional API in the future

Also, the decision was made to continue the assignment according to its original definition and plan and to learn from its results.

16.2 The database standards

Chapter 3 is devoted to the, then, upcoming database standards SQL3 and ODMG 2.0. When the research was done, the SQL3 standard was not yet finalized and it was far from clear if the ODMG 2.0 would be successful.

SQL3 was standardized at the end of 1999. Although standardized, its object-relational features are still somewhat controversial and not yet widely supported.

The ODMG suffered an even worse fate as it was disbanded in 2001 after releasing version 3.0 of their specification.

In 2001 the ODMG Java Language Binding was submitted to the Java Community Process as a basis for the Java Data Objects specification. The ODMG member companies then decided to concentrate their efforts on the Java Data Objects specification and disbanded their organisation. In 2004, the right to revise the ODMG specification was granted to the Object Management Group (OMG).

In 2006 the OMG announced plans to work on revision 4 of the ODMG specification.

16.3 A new standard

The Splice2 project was limited to create a new Splice implementation with an original Splice compatible API. This did not imply that the long-term desire to have object-oriented features in Splice and have a standardized API were abandoned.

Under the umbrella of the Object Management Group (OMG) two companies, Thales and Real-Time Innovations, set out to create a new standardized API for data distribution in distributed real-time systems, the *Data Distribution Service* (DDS). There are currently two products that implement the DDS standard:

- OpenSplice by PrimsTech (formerly Thales Splice)
- DDS by Real-Time Innovation

The original Thales Splice found a new home at the company PrismTech that specializes in middleware solutions. Instead of being a home-grown solution only used in Thales products, OpenSplice is now a product on its own and available to more parties than Thales alone.

16.3.1 The Data Distribution Service by OMG

As stated in the the conclusions (chapter 14) there was no good match between Signaal's requirements in 1999 and the ODMG API. The DDS solves this mismatch by providing two data oriented API's:

- Data-Centric Publish Subscribe API (DCPS). This is an API, that focusses on providing a low-level QoS based API to exchange single messages between publishers and their subscribers. This API is designed to be implemented in languages like C.
- Data Local Reconstruction Layer API (DLRL). This layer is able to reconstruct an object-graph between messages exchanged through DCPS based on their keys. DLRL strongly resembles object-relational mappers (ORM), because it uses the implicit relationships in the messages (foreign keys) to create application objects that have directly navigatable relationships. The DLRL layer is meant to be implemented in languages that support object-oriented features like C++ and Java.

The biggest difference between the layers is that the DCPS layer does not handle object graphs. Relationships between objects are modelled as foreign key fields and only recombined to graphs in the DLRL layer. Refer to paragrah 1.4.2 to see an example of relationships modelled using foreign keys.

16.3.2 Relationship to this assignment

The DLRL is the part of DDS that solves Signaal's original goal of being able to use richer object-oriented modelling techniques for their data model. Using the DLRL, applications do not have to recombine the relations between messages themselves, but DDS does this for them. (Parts of) Splice2DB could be used to provide the object storage of the recombined objects.

Also, the OpenSplice product has strong support for run-time availability of meta-data which was inspired on the meta-data abilities of Splice2DB.

In retrospect we may say that the original ambitions of the Splice2 project to provide a full distributed publish/subscribe object database system were too high. Analogue to developments in the relational database world the DDS standard advocates a two layer approach: one layer for simple tuple management, like a relational database, and one layer to combine tuples to objects, like an object-relational mapper.

Part V

Appendix

A Data-access Interface (API)

Earlier on the decision was made to use the ODMG interface as the interface for Splice2DB. This chapter briefly explains the data-access interface defined by ODMG 2.0. More complete descriptions can be found in [C^{+}97] and [Jor98].

Not all ODMG classes are relevant for Splice2DB, like `d_Transaction`. Only the most important classes will be discussed. These classes are:

- `d_Database`
- `d_Extent<T>`[1]
- `d_Object`
- `d_Ref_Any`
- `d_Ref<T>`
- `d_Rel_Ref<T,MT>`
- `d_Rel_Set<T,MT>`
- `d_Rel_List<T,MT>`

Not all functionality needed for Splice2DB can be provided with this interface, the standard has been extended with the following classes:

- `d_RW_Ref<T>`
- `d_Read_Ref<T>`
- `d_Reference<T>`

Database

The main entry point for every application using Splice2DB is the `d_Database` class. Before anything can be done on a database, an application has to create a `d_Database` instance and open a database using its `open` member-function.

d_Database

The database object is transient, that is, a database cannot be stored in a database. Databases must be opened before starting any action that use the database, and closed after ending these actions.

Table A.1 shows the member functions available to applications.

An application that uses the database has to follow a specific calling sequence:

[1]All classes contaning <T> are template classes

Method	**Semantics**
`d_Database()`	Constructor
`~d_Database`	Destructor
`void open(...)`	Opens a database by name
`void close()`	Closes an opened database
`void set_object_name(...)`	Attaches a name to an object in the database
`void rename_object(...)`	Renames an object in the database
`d_Ref_Any lookup_object(...)`	Finds an object by name

Table A.1: `d_Database` member functions

```
d_Database d;     //create a database object
d.open("name");   //open a named database
   //various actions on database
d.close();        //close the database
```
[2]

or

```
d_Database *d = new d_Database;
d->open("name");
   //various actions on database
d->close();
delete d;
```

When an object of this type has been created, applications can use the database. An applications now has to possibilities to retrieve data that is already in the database:

1. Use the `lookup_object` member function, to look up a paritcular object by name
2. Ask for the extent containing all instances of a type

Using the 'Singleton' design pattern [GHJV95], it can be ensured that the database class has only one instance. Although this is not ODMG-compliant, this will prevent the system from having to collect objects from more than one database when the application asks for the extent of a type.

Accessing objects

`d_Object`

All objects that can be stored in the database have to be derived from `d_Object`. In ODMG terms, deriving from `d_Object` makes a class *persistence-capable*.

```
class Track : public d_Object {...};
```

makes `Track` *persistence-capable*.

Table A.2 shows the member functions available to applications.

[2]A call to the desctuctor of the database object, is automatically inserted by the compiler when the object goes out of scope

Method	Semantics
`d_Object()`	Constructor
`d_Object(...)`	Copy constructor
`~d_Object`	Destructor
`d_Object& operator=(...)`	Assignment operator
`void* operator new(...)`	Creates persistent or transient instance
`void operator delete(...)`	Deletes instance from the database

Table A.2: `d_Object` member functions

`d_RW_Ref<T>`

The `d_RW_Ref` template class acts as a smart pointer to an instance of class T. There is also a `d_Ref_Any` class (section A) that provides a generic reference to any type. `d_RW_Ref` is a template class, which means that it provides a type-safe reference to instances of class T. A `d_RW_Ref<X>` cannot be used to reference an instance of class `Y`. References behave like C++ pointers, but provides a safe access to objects in the database. Having a `d_RW_Ref` reference to an object in the database means having *write-accesss* to it. When releasing the reference, the write-lock gets released. When an objects has to be accessed only for reading, it is better to use `d_Read_Ref<T>`. A `d_RW_Ref<T>` reference can be downgraded to a `d_Read_Ref<T>` (read-only) or a `d_Ref<T>` (non-read/non-write) reference.

Table A.3 shows the member functions available to applications.

Method	Semantics
`d_RW_Ref()`	Constructor, null reference
`d_RW_Ref(...)`	Constructor, copy reference from pointer, `d_Ref_Any`, `d_Read_Ref<T>` or `d_Ref<T>`
`~d_RW_Ref`	Destructor
`... operator =(...)`	Assign reference from pointer, `d_Ref_Any`, `d_Read_Ref<T>` or `d_Ref<T>`
`operator ...()`	Various cast operators
`void clear()`	Set reference to null reference
`T* operator→()`	Dereference, return pointer to referenced object
`T& operator*()`	Dereference, return referenced object
`T* ptr()`	Return pointer to referenced object
`void delete_object()`	Deletes referenced object from database
`bool operator !()`	Return true if the reference is a null reference
`bool is_null()`	Return true if the reference is a null reference
`bool operator==(...)`	Various equality operators (friends, not members)
`bool operator!=(...)`	Various not equal operators (friends, not members)

Table A.3: `d_RW_Ref<T>` member functions

`d_Read_Ref<T>`

The `d_Read_Ref` template class is very similar to the `d_Ref` template class. The only difference is that `d_Read_Ref` is a read-only reference. Having a `d_Read_Ref` to an object, means that you have a *read-lock* to it. When releasing the reference, the write-lock gets released.

A `d_Read_Ref<T>` reference can be upgraded to a `d_RW_Ref<T>` (read/write) reference or be downgraded to a `d_Ref<T>` (non-read/non-write) reference. There is no delete function for this class, because that requires a write-lock. All member-functions that dereference the reference, return a const-pointer or a const-reference. This way the compiler can check if the user tries to change the object.

Table A.4 shows the member functions available to applications.

Method	**Semantics**
`d_Read_Ref()`	Constructor, null reference
`d_Read_Ref(...)`	Constructor, copy reference from pointer, `d_RW_Ref<T>`, `d_Read_Ref<T>`, `d_Ref<T>`, `d_Ref_Any`
`~d_Read_Ref()`	Destructor
`... operator =(...)`	Assign reference from pointer, `d_RW_Ref<T>`, `d_Read_Ref<T>`, `d_Ref<T>` or `d_Ref_Any`
`operator ...()`	Various cast operators
`void clear()`	Set reference to null reference
`const T* operator→()`	Dereference, return *const* pointer to referenced object
`const T& operator*()`	Dereference, return *const* reference to referenced object
`const T* ptr()`	Return *const* pointer to referenced object
`bool operator !()`	Return true if the reference is a null reference
`bool is_null()`	Return true if the reference is a null reference
`bool operator==(...)`	Various equality operators (friends, not members)
`bool operator!=(...)`	Various not equal operators (friends, not members)

Table A.4: `d_Read_Ref<T>` member functions

`d_Ref<T>`

The d_Ref template class is a class like `d_Ref`, but you cannot access the instance it references. This class can be used to keep a reference. This is useful when you need the reference later, but you don't want to have a read or write lock on it. Keeping a read or write-lock prevents other processes/threads from updating the referenced object. A `d_Ref<T>` reference can be upgraded to a `d_Read_Ref<T>` (read-only) reference or a `d_Ref<T>` (read/write) reference. Table A.5 shows the member functions available to applications.

`d_Ref_Any`

Class `d_Ref_Any` provides a generic interface to an instance of any persistence-capable class, any class derived from `d_Object`. The standard defines `d_Ref_Any` to have a method `delete_object()`. This member has not been included, since `d_Ref_Any` is a sort of `d_Ref<T>`. That is, it has a pointer to the object, but no way to extract this pointer from the reference in order to manipulate the object. TableA.6 shows the member functions available to applications.

Method	Semantics
`d_Ref()`	Constructor, null reference
`d_Ref(...)`	Constructor, copy reference from pointer, `d_RW_Ref<T>`, `d_Read_Ref<T>`, `d_Ref<T>`, `d_Ref_Any`
`~d_Ref()`	Destructor
`... operator =(...)`	Assign reference from pointer, `d_RW_Ref<T>`, `d_Read_Ref<T>`, `d_Ref<T>` or `d_Ref_Any`
`operator ...()`	Various cast operators
`void clear()`	Set reference to null reference
`bool operator !()`	Return true if the reference is a null reference
`bool is_null()`	Return true if the reference is a null reference
`bool operator==(...)`	Various equality operators (friends, not members)
`bool operator!=(...)`	Various not equal operators (friends, not members)

Table A.5: `d_Ref<T>` member functions

Method	Semantics
`d_Ref_Any()`	Constructor, null reference
`d_Ref_Any(...)`	Constructor, copy reference from pointer, `d_RW_Ref<T>`, `d_Read_Ref<T>`, `d_Ref<T>`, `d_Ref_Any`
`~d_Ref_Any()`	Destructor
`... operator =(...)`	Assign reference from pointer, `d_RW_Ref<T>`, `d_Read_Ref<T>`, `d_Ref<T>` or `d_Ref_Any`
`operator ...()`	Various cast operators
`void clear()`	Set reference to null reference
`bool operator !()`	Return true if the reference is a null reference
`bool is_null()`	Return true if the reference is a null reference
`bool operator==(...)`	Various equality operators (friends, not members)
`bool operator!=(...)`	Various not equal operators (friends, not members)

Table A.6: `d_Ref_Any` member functions

`d_Reference<T>`

Template class `d_Reference<T>` provides a reference from one `d_Object` descendant to another `d_Object` descendant. As described in section 8.7 this reference type requires deadlock detection if the $\rightarrow$ operator is to be supported. Because relations are a more powerful reference mechanism and the possible interrelations are known by the database metadata, the $\rightarrow$ operator will not be supported. This type of reference has to be locked by the application programmer.

Table A.7 shows the member functions available to applications.

Method	Semantics
`d_Reference()`	Constructor, null reference
`~d_Reference()`	Destructor
`... operator =(...)`	Assign reference from pointer, `d_RW_Ref<T>`, `d_Read_Ref<T>`, `d_Ref<T>` or `d_Ref_Any`
`operator ...()`	Various cast operators
`void clear()`	Set reference to null reference
`bool operator !()`	Return true if the reference is a null reference
`bool is_null()`	Return true if the reference is a null reference
`bool operator==(...)`	Various equality operators (friends, not members)
`bool operator!=(...)`	Various not equal operators (friends, not members)

Table A.7: `d_Reference<T>` member functions

Accessing extents

`d_Extent<T>`

The template class `d_Extent<T>` provides an interface to the extent of class T. The extent of class T contains all instances of T in the database. It is maintained automatically as instances are created and deleted. The interface as defined by ODMG also includes member functions to execute queries on an extent. These member functions have been removed because queries are not part of the assignment.

Method	Semantics
`d_Extent(d_Database* db)`	Constructor, gets extent of type T from db
`~d_Extent`	Destructor
`unsigned long cardinality()`	Returns number of T instances in extent
`bool is_empty()`	Returns true if there are no T instances in extent
`bool allows_duplicates()`	Always returns false, extents do not contain duplicates
`bool is_ordered()`	Always returns false, extents are unordered
`d_Iterator<T> begin()`	Returns iterator pointing at beginning of extent
`d_Iterator<T> end()`	Returns iterator pointing past the end of extent

Table A.8: `d_Extent<T>` member functions

Relations

`d_Rel_Ref<T,MT>`

This template class represents a to-one bidirectional relationship between a class `S` (in which an instance of `d_Rel_Ref<T,MT>` is embedded) and the class `T`. The class `T` at the other end of the bidirectional relationship must have a member that is of type `d_Rel_List<S,MS>`, `d_Rel_Set<S,MS>`, or `d_Rel_Ref<S,MS>`. The second template parameter `MT` must be a character string that contains the name of the member in class T that references class S. Similarly, the member in T should have a second template parameter, MS, that refers to the member in class S that is the instance of this class `d_Rel_Ref<T,MT>`.

This class is always contained in an object definition. Therefore, the generation of the code can be left to a SODL to C++ parser that determines the template parameters .

Method	**Semantics**
`d_Rel_Ref()`	Constructor, null relation
`~d_Rel_Ref()`	Destructor
`... operator=(...)`	Assign relation from pointer, `d_Ref<T>`, `d_Read_Ref<T>`, `d_RW_Ref<T>` or `d_Ref_Any`
`void clear()`	
`T* operator→()`	Dereference, return pointer to the related object
`T& operator*()`	Dereference, return reference to the related object
`T* ptr()`	Return pointer to the related object
`void delete_object()`	Delete the reference object

Table A.9: `d_Rel_Ref<T,MT>` member functions

`d_Rel_Set<T,MT>`

This template class represents an unordered to-many bidirectional relationship between the class S (in which an instance of `d_Rel_Set<T,MT>` is embedded) and the class T. In the ODMG interface this class inherits from `d_Set` publicly. Therefore, all public member functions defined for `d_Set` are part of the interface of `d_Rel_Set<T,MT>`. But, to support the three-level iterator mechanism, this interface has to be overridden by `d_Rel_Set<T,MT>`. The most important member functions are the functions that create iterators, insertion and deletion operators.

`d_Rel_Set<T,MT>` defines three types:

1. `d_Rel_Set<T,MT>::iterator`
2. `d_Rel_Set<T,MT>::read_iterator`
3. `d_Rel_Set<T,MT>::rw_iterator`

`d_Rel_Set<T,MT>` has a lot of member functions, for more information see [Jor98] and [C^+97].

`d_Rel_List<T,MT>`

This template class represents an ordered to-many bidirectional relationship between the class S (in which an instance of `d_Rel_List<T,MT>` is embedded) and the class T. In the ODMG interface this class inherits from `d_List` publicly. Therefore, all public member functions defined for `d_List` are part of the interface of `d_Rel_List<T,MT>`. But, to support the three-level iterator mechanism, this interface has to be overridden by `d_Rel_List<T,MT>`. The most important member functions are the functions that create iterators, insertion and deletion operators.

`d_Rel_List<T,MT>` defines three types:

1. `d_Rel_List<T,MT>::iterator`
2. `d_Rel_List<T,MT>::read_iterator`
3. `d_Rel_List<T,MT>::rw_iterator`

`d_Rel_List<T,MT>` has a lot of member functions, for more information see [Jor98] and [C^+97].

B Storage

As identified in chapter 6 there will be three storage arenas in Splice2DB. This chapter will describe the creation and use of these arenas.

Storage allocation

As part of the Splice2 project, Signaal developed a general-purpose shared memory claimer and general-purpose memory allocator. This claimer and allocator will be used in Splice2DB. The memory allocator is able to operate on any contiguous memory area it is assigned to. So it is possible to open one shared memory area, and assign multiple allocators to non-overlapping areas in this single area. Not a region of shared memory, but a contiguous memory area assigned to an allocator will be called an arena.

Creation of arenas

When a database with a given name is opened, it will have to create or attach to storage arenas, depending on whether a database with that name has already been opened.

Figure B.1 shows a sequence diagram of the `d_Database::open` method. The actions are:

1. Create or attach to three arenas
2. Get the root object from the metadata administration
3. Get the root object from the object administration
4. Create an initial object administration for extents and name-instance administration if this is a newly opened database
5. Create an initial metadata administration for type information if this is a newly opened database
6. Store any type definitions defined by the application in the metadata administration

Using arenas

In C++ it is possible to overload the standard `new` and `delete` operators. This makes it possible to do a customized memory-allocation for a class.

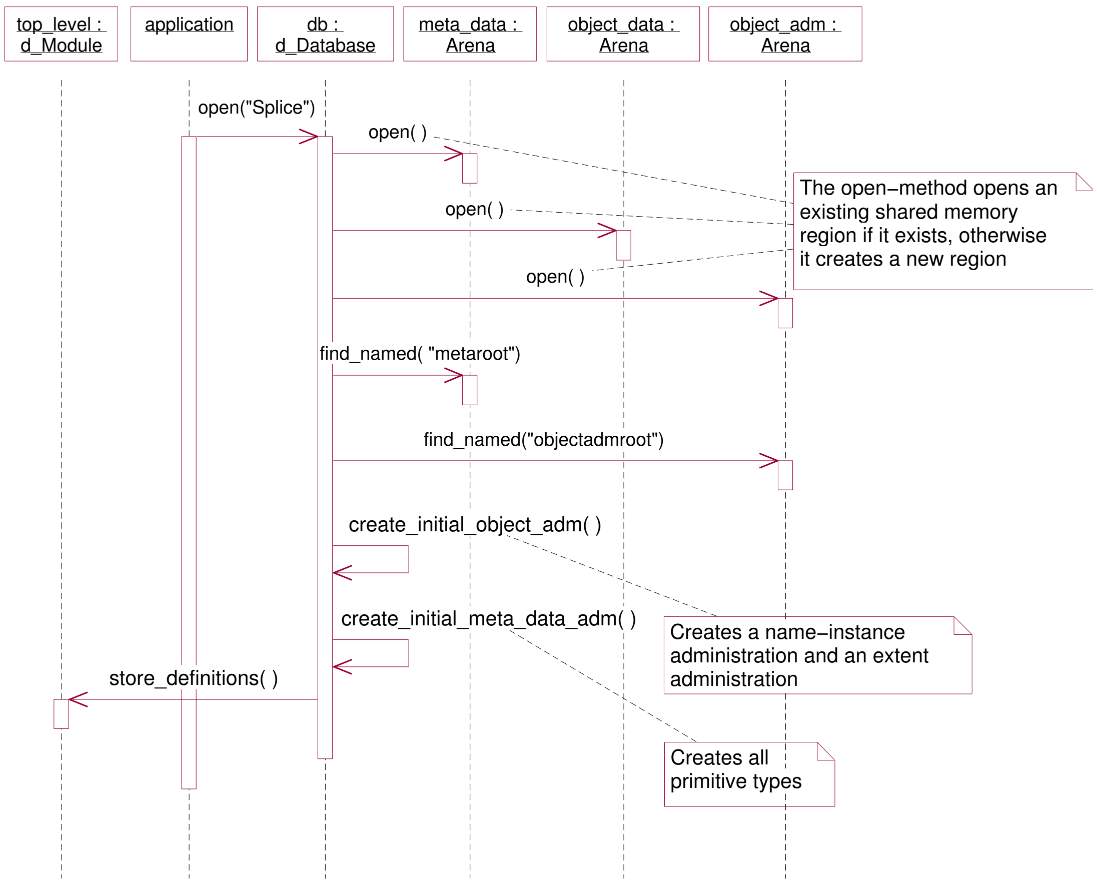

top_level : d_Module
application
db : d_Database
meta_data : Arena
object_data : Arena
object_adm : Arena
open("Splice")
open( )
open( )
open( )
The open−method opens an existing shared memory region if it exists, otherwise it creates a new region
find_named("metaroot")
find_named("objectadmroot")
create_initial_object_adm( )
create_initial_meta_data_adm( )
Creates a name−instance administration and an extent administration
store_definitions( )
Creates all primitive types

Figure B.1: Creation of arenas when opening a database

Metadata classes

All classes in the metadata class hierarchy can be derived from a single base class that overloads the `new` and `delete` operators. This class will be called `d_Meta_Storage`.

Extent-administration

The extent administration will be built using collection classes from the C++ STL. These collection classes can be provided with a customized allocator. Information about building a customized STL allocator can be found in [Str97].

Name-instance administration

The name-instance administration will be built using collection classes from the C++ STL. See previous paragraph.

Object data

In the ODMG model all object that need to be stored in the database are derived from `d_Object`. Therefore, overloaded `new` and `delete` operators in the `d_Object` class can be provided. Derived classes should never redefine the `new` and/or `delete` operators.

C Splice2 Object Definition Language

The Splice2 Object Definition Language is a subset of ODL by ODMG 2.0. The most notable differences are:

- No interfaces
- No modules within modules
- No multiple inheritance
- No constant expression for initializing constant attributes
- No named extents
  A class can only be declared to have or have not an extent

Specification

Rule			Nr.
<class>	::=	<class_header> { <interface_body> }	2a
<class_header>	::=	`class` <identifier> [`extends` <scoped_name>] [<type_property_list>]	2b

Type Characteristics

Rule			Nr.
<type_property_list>	::=	([<extent_spec>] [<key_spec>])	2c
<extent_spec>	::=	`extent`	2d
<key_spec>	::=	`key[s]` <key_list>	2e
<key_list>	::=	<key> | <key> , <key_list>	2f
<key>	::=	<property_name> | (<property_list>)	2g
<property_list>	::=	<property_name> | <property_name> , <property_list>	2h
<property_name>	::=	<identifier>	2i
<scoped_name>	::=	<identifier> | `::` <identifier> | <scoped_name> `::` <identifier>	11

Instance Properties

Rule			Nr.
<interface_body>	::=	<export> | <export> <interface_body>	8
<export>	::=	<type_dcl> `;` | <const_dcl> `;` | <except_dcl> `;` | <attr_dcl> `;` | <rel_dcl> `;` | <op_dcl> `;`	9

Types

Rule			Nr.
<const_dcl>	::=	`const` <const_type> <identifier> = <const_exp>	12
<const_type>	::=	<integer_type> | <char_type> | <boolean_type> | <floating_pt_type> | <string_type> | <scoped_name>	13
<const_exp>	::=	<integer_literal> | <string_literal> | <character_literal> | <floating_pt_literal> | <boolean_literal>	new
<type_dcl>	::=	`typedef` <type_declarator> | <struct_type> | <union_type> | <enum_type>	27
<type_declarator>	::=	<type_spec> <declarators>	28
<type_spec>	::=	<simple_type_spec> | <constr_type_spec>	29
<simple_type_spec>	::=	<base_type_spec> | <template_type_spec> | <scoped_name>	30
<base_type_spec>	::=	<floating_pt_type> | <integer_type> | <char_type> | <boolean_type> | <octet_type> | <date_type> | <time_type> | <interval_type> | <timestamp_type>	31*
<date_type>	::=	`date`	31a
<time_type>	::=	`time`	31b
<interval_type>	::=	`interval`	31c
<timestamp_type>	::=	`timestamp`	31d
<template_type_spec>	::=	<array_type> | <string_type> | <coll_type>	32*
<coll_type>	::=	<coll_spec> `<` <simple_type_spec> `>`	32a
<coll_spec>	::=	`set` | `list` | `bag`	32b
<constr_type_spec>	::=	<struct_type> | <union_type> | <enum_type>	33
<declarators>	::=	<declarator> | <declarator> , <declarators>	34
<declarator>	::=	<simple_declarator> | <complex_declarator>	35
<simple_declarator>	::=	<identifier>	36
<complex_declarator>	::=	<array_declarator>	37
<floating_pt_type>	::=	`float` | `double`	38
<integer_type>	::=	<signed_int> | <unsigned_int>	39
<signed_int>	::=	<signed_long_int> | <signed_short_int>	40
<signed_long_int>	::=	`long`	41
<signed_short_int>	::=	`short`	42
<unsigned_int>	::=	<unsigned_long_int> | <unsigned_short_int>	43
<unsigned_long_int>	::=	`unsigned long`	44
<unsigned_short_int>	::=	`unsigned short`	45
<char_type>	::=	`char`	46
<boolean_type>	::=	`boolean`	47
<octet_type>	::=	`octet`	48
<any_type>	::=	`any`	49
<struct_type>	::=	`struct` <identifier> `{`<member_list>`}`	50
<member_list>	::=	<member> | <member> <member_list>	51
<member>	::=	<type_spec> <declarators> `;`	52

Rule			Nr.
<union_type>	::=	union <identifier> switch (<switch_type_spec>) { <switch_body> }	53
<switch_type_spec>	::=	<integer_type> | <char_type> | <boolean_type> | <enum_type> | <scoped_name>	54
<switch_body>	::=	<case> | <case> <switch_body>	55
<case>	::=	<case_label_list> <element_spec> ;	56
<case_label_list>	::=	<case_label> | <case_label> <case_label_list>	56a
<case_label>	::=	case <const_exp> : | default :	57
<element_spec>	::=	<type_spec> <declarator>	58
<enum_type>	::=	enum <identifier> { <enumerator_list> }	59
<enumerator_list>	::=	<enumerator> | <enumerator> , <enumerator_list>	59a
<enumerator>	::=	<identifier>	60
<array_type>	::=	<array_spec> < <simple_type_spec> , <positive_int_const> > | <array_spec> < <simple_type_spec> >	61*
<array_spec>	::=	array | sequence	61a*
<string_type>	::=	string < <positive_int_const> > | string	62
<array_declarator>	::=	<identifier> <array_size_list>	63
<array_size_list>	::=	<fixed_array_size> | <fixed_array_size> <array_size_list>	63a
<fixed_array_size>	::=	[<positive_int_const>]	64

Attributes

Rule			Nr.
<attr_dcl>	::=	[readonly] attribute <domain_type> <attribute_name> [<fixed_array_size>]	65*
<domain_type>	::=	<simple_type_spec> | <struct_type> | <enum_type>	65a

Relationships

Rule			Nr.
<rel_dcl>	::=	relationship <target_of_path> <identifier> inverse <inverse_traversal_path>	65b
<target_of_path>	::=	<identifier> | <rel_collection_type> < <identifier> >	65c
<inverse_traversal_path>	::=	<identifier> :: <identifier>	65d
<rel_collection_type>	::=	set | list | bag	65e

Operations

Rule			Nr.
<op_dcl>	::=	[<op_attribute>] <op_type_spec> <identifier> <parameter_dcls>	67
<op_attribute>	::=	oneway	68
<op_type_spec>	::=	<simple_type_spec> | void	69
<parameter_dcls>	::=	([<param_dcl_list>])	70
<param_dcl_list>	::=	<param_dcl> | <param_dcl> , <param_dcl_list>	70a
<param_dcl>	::=	<param_attribute> <simple_type_spec> <declarator>	71
<param_attribute>	::=	in | out | inout	72

Example

This example shows a definition of a `Person` class. The `spouse` relation is a 1:1 relation with itself. The `children` relation is a N:M relation. Its inverse relation is the `parents` relation in `Person`. `Person` is specified to have an extent and to maintain indices on the name and address field.

```
class Person
( extent key name, address )
{
   attribute string name;
   attribute struct Address { unsigned short number, string street,
      string city_name } address;
   relationship Person spouse
      inverse Person::spouse;
   relationship set<Person> children
      inverse Person::parents;
   relationship list<Person> parents
      inverse Person::children;
};
```

D Access acceleration methods

A database system is not complete without a query mechanism. Because of the complexity of queries this query mechanism is not part of the assignment. However, efficient access methods are the key to high performance query execution. These mechanisms cannot be provided without support of the database. These access methods have to provide efficient functionality for

- inserting elements,
- deleting elements,
- searching elements,
- range searching elements, and
- sequential scan searching elements[1],

while not using to much resources. Because Signaal explicitly chooses for maximizing performance even when a lot more resources are needed, performance remains key issue.

Indices

An index provides a mapping of a value to a reference, and, by its structure, an efficient means for searching for a particular key. An index can be kept for one or more attributes of a class to allow efficient lookup of instances based on their values for those attributes. There is an entry in the index for each instance of the class. ODMG does not yet support the specification of indices in their interface definition. Some ODMG-compliant implementations allow indices on multiple attributes, some only on one attribute.

An index mechanism needs to compare the values of the indexed attribute. Some database implementations only allow indices on primitive data types that the database software already understands (e.g. integer, double, char, ...) and knows how to compare. Other database implementations provide a mechanism for the database to gain access to operators defined for the application defined type, in order to be able to compare complex attributes.

When the value of an indexed attribute is modified, the index must be updated to reflect the change.

Suppose the following data structure containing three integer attributes:

```
class Data {
   attribute integer field1;
   attribute integer field2;
   attribute integer field3;
}
```

[1]read each item sequentially

When an application programmer asks the database to maintain an index on `field1,` queries on `field1` will perform significantly faster than without an index on `field1`. When the database has to perform a query on all Data instances like `"WHERE field1=3`", the database can now search for the object containing a `field1` value equal to 3 in the index , and return a reference to that object. The database does not have to scan all instances for the value.

It is up to the application programmer or type definer[2] to decide which attributes need indices.

Keys

An object can have one ore more attributes whose values uniquely identify an instance. These attributes are referred to as the *key* for the object.

A key can be simple or composite. A *simple key* consists of a single attribute, whereas a *composite key* consists of multiple attributes. A *foreign key* is an attribute used to contain the value of an object's primary key. Such a foreign key is similar to, but less efficient than, an object reference in an object database.

Supporting keys in the database means that after every update of an object that has attributes tagged as key, the system should check the uniqueness of the key attributes, and reject or permit the change.

Keys and indices

In a relational database system, keys are the primary means for relations between rows in a table. A foreign key in a table refers to a key in another table. Efficiently combining these two tables (joining) can be performed by using the index on the key and foreign key fields.

When an index is requested for a key field, all entries in the index are unique because all values in the key field are unique. Supporting indices on non-key fields implies supporting indices that allow for more than one instance for a key value.

For the future design of the query mechanism two choices must be made:

1. Support for keys or not
2. Support for indices on non-unique values or not

Not supporting keys implies supporting indices on non-unique values.

Maintaining indices & keys

Indices can provide high-speed access to the database, but have to be maintained every time an object is inserted, updated or deleted. Thus, there is overhead involved in maintaining indices. Depending on the type of index, an index can also take up a considerable amount of space if it contains a large number of entries. A trade-off exists between the performance of lookup versus update of index because of these maintenance costs.

The next three paragraphs will discuss the three actions.

[2]There is discussion about who should be responsible for assigning indices to the attributes of a data structure.

Inserting

When an object is inserted in the database it has to be inserted in the indices associated with the type of the object. If insertion does not violate any key-constraints, the database system has to read the indexed attributes and insert the reference to the object at the correct place in the index. If insertion does violate the constraints of a key attribute, insertion must be denied.

Updating

When an object is updated in the database, it might have to be moved in its associated indices.

If the update does not violate any key-constraints, the database system has to read the indexed attributes and possibly have to move the reference to the object to the correct place in the index.

If the update does violate the constraints of a key attribute, the update must be denied and the old key-values must be restored. This also implies that the old key-values should be kept by the database.

Deleting

When an object is deleted its reference must be deleted from the index.

Data representation for indices

The database system can, for instance, create an ordered tree structure as index, containing pointers to instances. In disk-resident databases, it is crucial to add the key value itself to the structure, since the pages containing the needed object might have to be brought into memory first, which is a big performance penalty. In main-memory databases these objects are always memory resident, so their values can be read directly.

Types of indices

Having an ordered index is crucial when a field is mostly queried on inequality, like `"WHERE field1>3`". However, hash-structures are know to be fastest when searching on equality, like `"WHERE field1=3`". Hash-structures however do not maintain an ordering, so range searching is known to be slower than for tree-structures. Sequential searching on hash-structures can be quite fast, but they are not ordered on the key-field. Therefore, it is up to the database programmer or type specifier to choose what type of index structure is to be used for a certain field.

Despite its age the article *'A study of index structures for main memory database management systems'* by Lehman & Carey [LC86] gives a good overview of possible index structures for main memory databases and their strengths and weaknesses. The different structures will not be discussed in this thesis.

Multiple indices on a type

When a type contains multiple attributes, multiple indices can be requested. These indices can be serialized or parallelized.

Serial indices

When indices are serialized, the order of indices matters. Not only for the internal structure, but also for the performance of the queries.

In a serialized index only the last index contains references to instances, the start and intermediate indices only contain references to sub-indices.

Structure

Suppose a type containing three indexed integer fields, `field1`, `field2` and `field3`. Indices are requested in that order. The resulting structure is depicted in figure D.1. The index for `field1` contains a new index on `field2`, the index on `field2` contains a new index on `field3`. Intermediary indices have to have copy of the key value in their index-nodes, because they don not have a reference to an instance. In this mechanism the combination of the indexed fields is a composite key if the last index cannot contain multiple instance references for a key value.

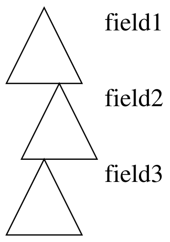


Figure D.1: Serial indices

Query execution

Figure D.2 shows a sample index structure when a binary tree is chosen to be the index mechanism. Three instances are inserted, $(1, 1, 1)$, $(2, 2, 1)$ and $(2, 2, 2)$. The start and intermediate indices only point to the next level index. When a query like `"WHERE field1=2"` is executed, the value 2 is looked up in the `field1` index. The result is all the instances in all sub-indices.

A major disadvantage of this approach is when a query like `"WHERE field3=1"` has to executed. For all nodes in the `field1` index and for all nodes in the `field2` index, the `field3` index has to be checked for a field equal to 1. All results from all sub-indices have to be combined. Therefore, the order of the indices and the expected form of the queries are very important to query performance.

Queries containing boolean expressions such as `"WHERE field1=1 AND field2=2"` can be executed very efficient.

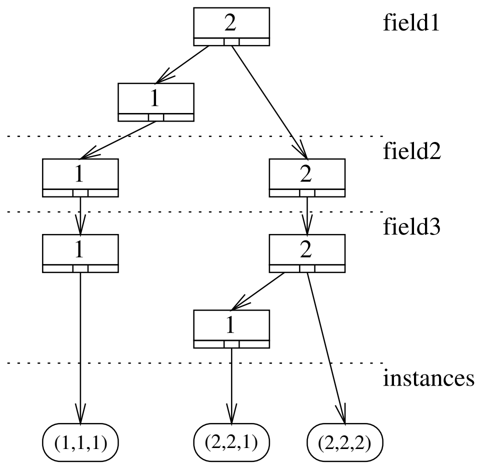


Figure D.2: Serial indices example

Resource usage

When the index is implemented as a tree, the resource usage can simply be calculated. Suppose an n-level index on m instances. The number of nodes for each index level is m in worst case, therefore in the total index, there are $n*m$ tree -nodes. If there are no more than j different values (where $j \leq m$) for the first attribute, the number of worst case tree-nodes can be reduced to:

$$j + (n-1) * m \leq n * m$$

In case of an index on 3 attributes that together form a key for a type , the number of worst case tree-nodes for 30 instances can vary between 90 (if all values of the first attribute are different) and 32 (if all values of the first and second attribute are the same). If all values of the first attribute are the same, the worst-case number of tree-nodes equals 61.

This also shows that the order of the indices is very important. A limited number of different values for the first attributes can reduce the number of sub-trees significantly.

Parallel indices

In a parallelized index the order of attributes does not matter. All indices contain references to all instances. Note that when a type has a composite key (a key containing more than one attribute), the indices must be capable of containing more than one reference for a key value. In case only simple keys are supported this is not necessary.

Structure

When indices are parallelized, a separate non-coupled index is created for every indexed attribute (see figure D.3). The key values don not have to be inserted in the index-nodes, because all nodes have a reference to an instance.

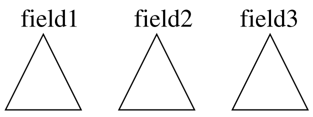


Figure D.3: Parallel indices

Query execution

Figure D.4 shows a sample index structure when a binary tree is chosen to be the index mechanism. Three instances are inserted, $(1,1,1)$, $(2,2,1)$ and $(2,2,2)$. This example treats the three fields as a composite key, that is, for one or two attributes two objects can have the same values, but not for all three. That is why each tree-node points to a list of objects. If every field is a key in its own right, each tree-node can directly point to an instance.

Now a query like `"WHERE field3=1"` performs as fast as `"WHERE field1=2"`. An expression like `"WHERE field1=1 AND field2=2"` has to be executed on the two indices separately and the two resulting sets have to be intersected. This is clearly more expensive than when being executed on a serialized index mechanism.

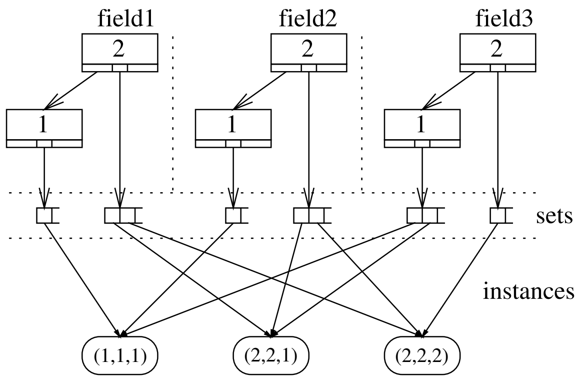


Figure D.4: Parallel indices example

Resource usage

When the index is implemented as a tree, the resource usage can simply be calculated. Suppose an n indices on m instances. Call u_n the number of different values for attribute n. The number of tree-nodes now equals $\sum_1^n u_n$. In worst case u_n equals m for all indices, therefore the number of tree-nodes in worst case equals $n * m$. If all of these indices are based on a key, they all point to an instance directly. If not, they all point to a list of references to instances, and the size of the list administration has to be taken into account too.

In case of an index on 3 attributes that together form a key for a type, the number of worst case tree-nodes for 30 instances can vary between 90 (if all values of the first attribute are different) and 32 (if all values of two attributes are the same, no matter which attribute). If all values of an attribute are the same (no matter which attribute) the number of worst-case tree-nodes equals 61.

Because the number of tree-nodes decreases immediately when the number of different values for one of the key attribute decreases, the average number of tree-nodes will be lower than the average for a serialized index mechanism.

Bibliography

[ADM$^+$89] M. Atkinson, D. DeWitt, D. Maier, F. Bancilhon, K. Dittrich, and S. Zdonik. The Object-Oriented Database System Manifesto. *ALTAIR technical report*, 30-89, September 1989.

[C$^+$94] R.G.G. Cattell et al. *Object Database Standard: ODMG-98.* Morgan Kauffman, 1994.

[C$^+$97] R.G.G. Cattell et al. *The Object Database Standard: ODMG 2.0.* Morgan Kauffman, 1997. ISBN 1-55860-463-4.

[Cod70] E.F. Codd. A relational model for large shared databanks. *Communications of the ACM*, 13(6):377–390, June 1970.

[Dat90] C.J. Date. *An introduction to database systems*, volume 1. Addison-Wesley Publishing Company, Inc., 5th edition, 1990.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns; Elements of Reusable Object-Oriented Software.* Addison-Wesley Publishing Company, Inc., 1995.

[GMS92] Hector Garcia-Molina and Kenneth Salem. Main Memory Database Systems: An Overview. *IEEE transactions on knowledge and data engineering*, 4(6), December 1992.

[Jor98] David Jordan. *C++ Object Databases; Programming with the ODMG standard.* Addison-Wesley Publishing Company, Inc., 1998.

[Knia] Knizhnik. *FastDB Main Memory Database Management System.* URL: `http://www.ispras.ru/~knizhnik/fastdb.html`.

[Knib] Knizhnik. *Generic Object Oriented Database System (GOODS).* URL: `http://www.ispras.ru/~knizhnik/goods.html`.

[Knic] Knizhnik. *GigaBASE Database Management System.* URL: `http://www.ispras.ru/~knizhnik/gigabase.html`.

[Lar98] Craig Larman. *Applying UML and patterns : an introduction to object-oriented analysis and design.* Prentice-Hall, 1998.

[LC86] Tobin J. Lehman and Michael J. Carey. A study of index structures for main memory database management systems. *Proceedings of the 12th international conference on Very Large Data Bases*, pages 294–303, 1986.

[LL96] Suh-Yin Lee and Ruey-Long Liou. A multi-granularity locking model for concurrency control in object-oriented database systems. *IEEE transactions on knowledge and data engineering*, 8(1):144–156, February 1996.

[McC97] Steve McClure. Object Database vs. Object-Relational Databases. *IDC Bulletin*, 14821E, August 1997. URL: `http://www.idcresearch.com`.

[Pot99] Robert Poth. *Software Design Document Splice2.* Internal document Holland Signaal B.V., 1999.

[Sig93] Holland Signaal. *An introduction to (Sigma/)SPLICE.* Internal document Holland Signaal B.V., 1993.

[Sil96] Silicon Graphics Computer Systems, Inc. *Standard Template Library Programmer's Guide*, 1996. URL: `http://www.sgi.com`.

[Str92] Bjarne Stroustrup. *The C++ programming language.* Addison WesleyPublishing Company, Inc., 2nd edition, 1992.

[Str97] Bjarne Stroustrup. *The C++ programming language.* Addison Wesley Publishing Company, Inc., 3rd edition, 1997.

[Sun98] Sun Microsystems, Inc. *Solaris 7 Software Development Collection: Linker and Libraries Guide*, October 1998.

[Vah96] Uresh Vahalia. *UNIX Internals.* Prentice Hall, 1996.