Stabilis*: A Case Study in Writing Fault-Tolerant Distributed Applications Using Persistent Objects

L.E.Buzato[†] A.Calsavara

l.e.buzato@newcastle.ac.uk a.calsavara@newcastle.ac.uk

Department of Computing Science, Computing Laboratory

The University, Newcastle upon Tyne, NE1 7RU, U.K.

Tel.: +44 91 222 8035

Abstract

This paper presents Stabilis, a fault-tolerant object-oriented distributed database management system that has been written as an exercise in persistent programming. Stabilis is implemented on top of Arjuna, an object-oriented programming system that provides the basic mechanisms for fault tolerance and distribution. The computational model used by Arjuna is based upon the concept of using atomic actions[1] to control operations upon persistent objects. Stabilis aims at experimenting with Arjuna to build large applications that use persistent objects. Such experiment has led us to extend some of the mechanisms for persistent programming already existent in Arjuna. Stabilis manages objects that are persistent, recoverable and can be accessed remotely and concurrently in a consistent manner. Objects with such properties have an important function in the overall operation of Stabilis. The database manager can be operated either through a visual database interface or a query interpreter; both translate commands into a series of operations of the database manager. All operations of the database manager make use of atomic actions and locks to structure and control accesses to objects. A flexible use of nested atomic actions permits objects to retrogress to previous consistent states. Stabilis has been developed using C++; dispensing with the use of any specifically designed language for persistent programming.

Key words:

Persistence, Atomic Actions, Type Inheritance, Fault Tolerance, Object-Oriented Databases

---

*Latin word synonymous with **stable**. This work has been partially funded by ESPRIT Project ISA (Project no. 2267), and CNPq (grants no. 200410/88-1 and 201905/91-4). In Persistent Object Systems, San Miniato 1992: Proceedings of the Fifth International Workshop, Italy, 1-4 September, pp. 354-375, Albano, A. and Morrison, R. (Eds.), Workshops in Computing, Springer-Verlag.

†On leave from the Department of Computing Science,University of Campinas, São Paulo, Brazil.

1 Atomic transactions.

1 Introduction

As distributed systems become more widely used, there is an increasing need for effective ways of organising distributed applications. Distributed applications can be hard to design and write because of the problems of concurrency and partial failure inherent to distributed systems. Keeping data consistent in the presence of failures is another serious problem whose solution can rely on the use of persistent objects and atomic actions.

Distributed applications typically have a number of requirements, including reliability, reconfigurability, concurrency control, availability and consistency. An application domain that contains problems covering such a comprehensive set of requirements is that of distributed database management systems. In this paper we survey the experience we have had during the implementation of Stabilis, a fault-tolerant object-oriented distributed database management system. We have implemented Stabilis on top of Arjuna[1], an object-oriented distributed programming system.

Objects are instances of abstract data types. An *atomic object* is an object that is recoverable and can be used concurrently in a consistent way. Atomicity is achieved through the use of atomic actions controlling operations upon objects. The definition of robust object evolves naturally from the definition of atomic object [2, 3]. A *robust object* is an atomic object that is persistent and can be accessed remotely[2].

Stabilis uses robust objects as its database entries, henceforth denominated *database objects*. To construct a distributed database using Stabilis a user-defined object-oriented data model is converted into a set of classes from which database objects are generated. Nested atomic actions are used to allow a flexible management of the database objects, permitting it to undo operations.

Another interesting aspect of our work is that Arjuna and Stabilis make use of C++ [4] to implement all mechanisms necessary for persistent programming, dispensing with the use of any specifically designed language.

The next Section comments on those aspects of Arjuna that were relevant to the development of Stabilis. Some of the modules of Arjuna are analysed in more detail because they have been extended during the implementation of Stabilis. Section 3 shows how persistent programming is accomplished using Arjuna. Section 4 introduces a module developed to extend some of the programming facilities offered by Arjuna for persistent programming. Section 5 is built around the design and implementation of Stabilis, its use of Arjuna and the platform it offers to the development of distributed databases. Finally, Section 6 describes Dbib, a bibliography database implemented to assess Stabilis. The paper concludes with the discussion of possible improvements that could be made to both Arjuna and Stabilis as a result of our experience. It also evaluates the use made so far of robust objects and type inheritance, considering their appropriateness to persistent programming.

[2] Ideally, a robust object should also be replicated. Arjuna is being extended to include replication of objects.

2 An Overview of Arjuna

Arjuna is an object-oriented programming system which provides a set of tools for constructing fault-tolerant distributed applications [1]. Arjuna supports nested atomic actions for structuring applications. Objects in Arjuna can be made either atomic objects or persistent atomic objects; they are the main repositories for holding system state. Operations upon them are invoked under the control of atomic actions. In Arjuna, operations on objects are of type *read* or *write*, following the locking rule that permits *multiple reads, single writes*. The well-known *strict two-phase* locking policy is adopted to ensure serialisability. Locks on objects are acquired inside an atomic action, and are released only when the outermost atomic action ends (or aborts) [5]. By ensuring that objects are persistent and only manipulated within an atomic action, it can be guaranteed that the integrity of objects – and hence the integrity of the system – is maintained in the presence of failures such as node crashes and the loss of network messages. This is the *object and action* model of computation – atomic actions controlling operations upon persistent objects.

2.1 Architecture

The main modules of the Arjuna system are shown in Figure 1. The RPC module is used to invoke operations on persistent objects. The Name Server module keeps identification and location information about persistent objects. The Object Store module encapsulates the stable representation of persistent objets. The Atomic Action module is the application-level interface. These modules are described further in the subsequent sections.

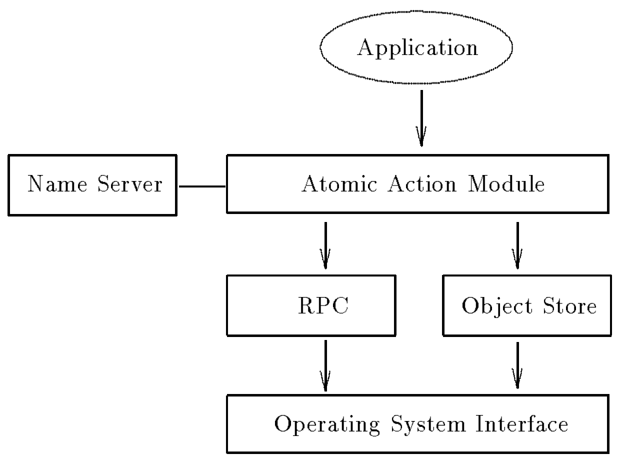

Figure 1: The architecture of Arjuna.

2.1.1 *RPC Module*

Arjuna adopts the client-server model for accessing persistent objects. A server manages an object state; it defines and executes operations that are exported to clients. Clients invoke these operations to manipulate the object state guarded by the server. Invocations of operations on persistent objects are implemented as *remote procedure calls* (RPC), which are supported by the RPC module.

The Arjuna programming environment provides a tool called Stub Generator [6] that processes definitions of C++ classes whose instances are persistent objects to be remotely accessed and, as a result, produces the corresponding client and server stub code. Transparency of location and access is obtained by making any invocation of an operation on the client stub object to trigger the same operation on the corresponding (remote) server stub object, using RPC.

2.1.2 *Name Server Module*

Names are used for several purposes in computer systems; one of the uses is to refer to objects. To name and find objects in a distributed system, *naming* and *binding* functions are usually provided through a *name server*. The naming function maps a user-suplied object name to a unique object identifier already assigned to the object. And, the binding function maps the unique identifier of an object to its location.

To further experiment with Arjuna and have a portable name server[7], we have decided to implement a naming and binding facility as an Arjuna application using persistent objects. In our version of the name server, two classes, `NameServer` and `NameServerManager`, implement a portable name server that relies solely on the mechanisms of the Atomic Action module of Arjuna. Instances of `NameServer` are robust objects, which act as servers, distributed among the nodes of the network, and instances of `NameServerManager` are clients that co-ordinate the operation of the servers. To a program, the Name Server module gives an illusion of a virtually centralised name server. All operations (for registering, unregistering and resolving names) invoked by a program are transfered by the Name Server client to the servers for execution. In the current implementation, all operations are sent to the servers using a round robin algorithm. In the future, other strategies for forwarding operations to the servers can be experimented. This issue has received lower priority since the main goal of the project was not to build a very complex name server but only to verify the possibility of building one based only on the services provided by the Atomic Action module of Arjuna.

2.1.3 *Object Store Module*

The Object Store provides an access service to the passive state of persistent objects. The stable representation of the passive state of a persistent object, usually stored in disk, has to be machine independent to permit its transmission between stable storage and volatile storage, and also its transmission as a message. The class `ObjectState` implements such a representation, providing operations for packing and unpacking the state of a persistent object into/from an instance of `ObjectState`. The function of the Object Store is to manage instances of the class `ObjectState`.

The set of operations provided by the Object Store include: **read_state**, which returns an instance of the `ObjectState` designated by a unique identifier, and **write_state**, which stores an instance of `ObjectState` identified by a given unique identifier. Figure 2 shows the lifetime and state transitions of a persistent object along with the operations that produce the transitions. The Atomic Action module activates a persistent object by first calling **read_state** and then **restore_state**, which unpacks the object state. The reverse operation comprises the execution of **save_state**, which packs the object state, and the invocation of **write_state**.

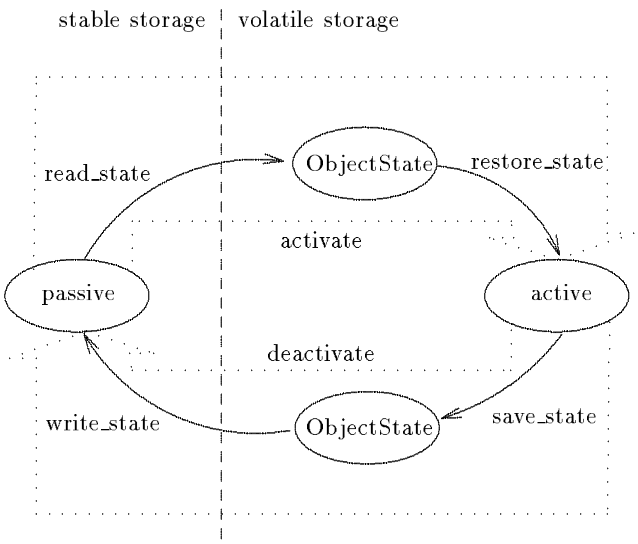

Figure 2: Object state transitions.

2.1.4 Atomic Action Module

The Atomic Action module provides the programming interface of Arjuna. The mechanisms necessary for concurrency control, persistence, recovery, and atomic action control are implemented by the classes of the class hierarchy depicted in Figure 3. These classes represent the internal structure of the Atomic Action module. To write an application that conforms to the *object and action* model of computation, a programmer declares instances of the class `AtomicAction` in his program; the operations provided by this class (**begin**, **end** and **abort**) can then be used to organise atomic actions. The only objects controlled by the resulting atomic actions are those objects that are either instances of Arjuna classes or user-defined classes derived from the class `LockManager` – type inheritance is used to make user-defined classes members of the hierarchy shown in Figure 3.

All Arjuna classes are derived from the base class `StateManager`, which provides the basic facilities needed for constructing persistent objects and atomic actions. The class `LockManager` uses the operations of the class `StateManager` to provide concurrency control. The other classes shown in Figure 3 implement most of the support operations of the Atomic Action module; further details about the class hierarchy of Arjuna can be found in [1].

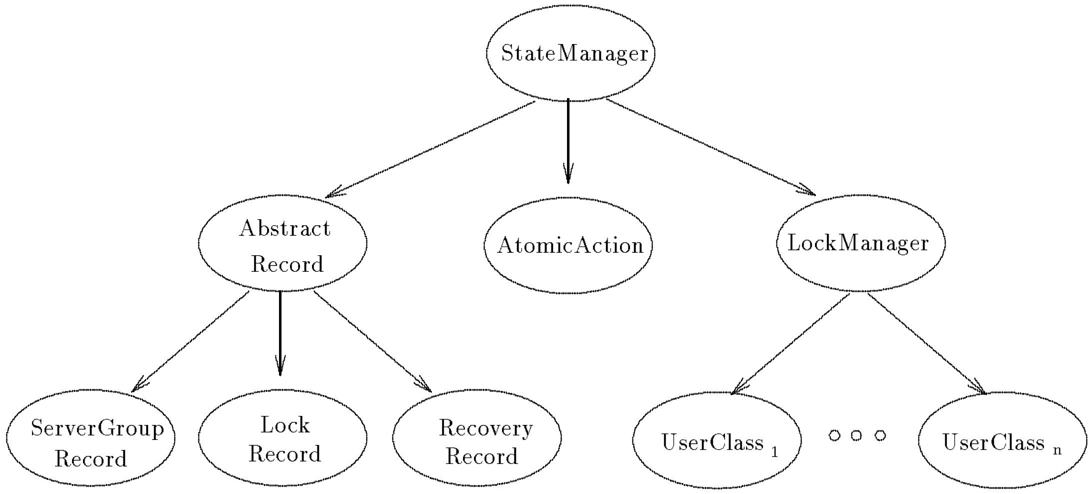

Figure 3: The Arjuna class hierarchy in the Atomic Action module.

2.2 Failure Assumptions

It is assumed that the hardware components of the system are workstations (nodes), connected by a communication sub-system (for example, a local area network). A node is assumed to work either as specified or simply to stop working (crash). After a crash a node is repaired within a finite amount of time and made active again. A node is assumed to have both stable and non-stable (volatile) storage. All of the data stored on volatile storage is assumed to be lost when a crash occurs; any data stored on stable storage, as stated earlier, remains unaffected by a crash. It is also assumed that faults in the communication sub-system are responsible for failures such as lost, duplicated or corrupted messages. The RPC system is assumed to be responsible for coping with such failures using well-known network protocol level techniques; it returns a failure exception to the caller if it suspects that the called server is not responding.

3 Persistent Programming in Arjuna

In this Section, a class denominated `Article` is used to explain the various steps involved in the conversion of a user-defined class into a class whose instances are robust objects using Arjuna. The programming procedure for the implementation of robust objects comprises the steps:

- Derive `Article` from the class `LockManager` provided by Arjuna (Figure 3);
- Implement the function to save the state of an `Article` instance into an `ObjectState` (**save_state**), and its reciprocal (**restore_state**);
- `Article` must have two constructors: one to be executed whenever a new persistent instance of `Article` is created; other to be executed every time an already existent instance of `Article` is reactivated;
- Every method of `Article` that modifies the state of the object has to be atomic.

At this stage instances of the class `Article` are persistent and recoverable but cannot be accessed remotely. To make them remotely accessible the class definition has to be submitted to the Stub Generator and the code for stub server has to be installed in each node of the distributed system where activations of `Article` occur.

The programming procedure described hitherto has some drawbacks. Let c be the number of classes an application (say A) controls, and n be the number of nodes where instances of these classes reside. If the procedure described is used to program the robust objects of the application A, then c classes will have to be submitted to the Stub Generator and $c \times n$ class-specific servers will have to be installed. Consequently, if c and n are large the development and management of a distributed application has its complexity considerably increased: more computational resources are used and more configuration errors can occur. Furthermore, the code obtained using this procedure does not intrinsically provide a caching scheme for robust objects.

An alternative procedure to implement robust objects can be based on the existence of a class (say `RobustObject`) that exempts the use of the Stub Generator and provides a caching mechanism. The programming procedure resulting from the adoption of this solution comprises steps very similar to those used by the procedure described earlier for Arjuna, except that:

- `Article` has now to be derived from `RobustObject`;
- it has not to be submitted to the Stub Generator.

A module, denominated Object Manager, described in the subsequent Section, has been implemented to provide the abstraction given by the class `RobustObject`.

4 Object Management

The Object Manager module provides an abstraction of robust objects that exempts the use of the Stub Generator while giving the possibility of caching objects to improve the efficiency of an application. Additionaly, a simple object migration mechanism can be developed using this module.

Several factors have to be considered before deciding on the use of the Object Manager: the size of the objects; the frequency of accesses made to the objects; the computational cost of the operations and the number of classes managed by the application. In particular, the Object Manager is very suited to the implementation of database managers because they control persistent objects that usually: have a small size, are frequently accessed, have no CPU-bound operations and are generated from a large number of classes.

4.1 Module Organisation

Figures 4a and 4b will be used to explain how the Object Manager module is organised. Figure 4a shows how persistent objects are remotely accessed in Arjuna: a Client invokes operations on a remote Server that uses the Object Store Manager to access the object. The dotted line shows the activation/deactivation of the persistent object, and the dashed line represents the communication between Client and Server. Figure 4b shows the configuration obtained when the Object Manager module is used. An extra client-server pair, the Object Manager and the Object Server, is interleaved between the Server and the Object Store Manager. In this case, the Server does not affect the persistent object directly because it is (potentially) remote: it works on a local *cache object*, a volatile copy of the robust object.

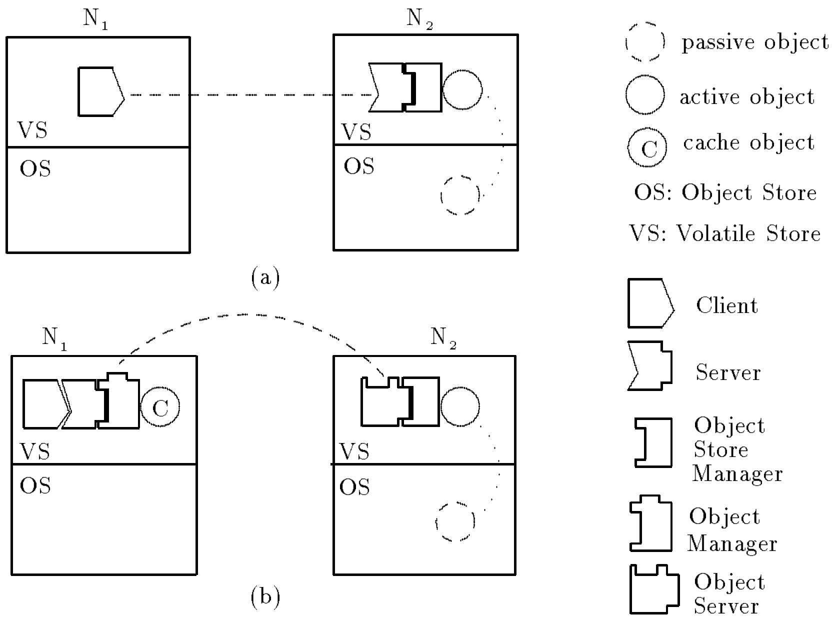

Figure 4: Organisation of the Object Manager.

4.2 Consistency of the Cache Object

The consistency of the robust object and cache object is guaranteed by the computational model of Arjuna. When a robust object is instantiated, the cache object is initiated with the state of the passive object. Subsequently, the server modifies the cache object and then its state is immediately copied to the passive object, i.e., the cache data is made permanent. If the update of the passive object fails then the state of the cache object has to be recovered. This implies that every server operation that modifies the cache object has to be atomic; its result (commit or abort) has to be the same of the update operation on the passive object. Depending on the type of lock (read/write) a client acquires on a robust object, the cache object behaves as a hint or as a proper cache[3]. When locked for read[4], the passive object can be modified by another client and, consequently, the cache object becomes stale. On the other hand, the write lock guarantees that the cache object will be always correct.

4.3 Implementation

The classes that implement the two components of the Object Manager module, Object Manager and Object Server, are respectively the classes `RobustObject` and `Package`.

The class `Package` is derived from the class `LockManager` of Arjuna. The atomic operations it exports are:

create ▷ stores a new object state on stable storage;

get ▷ reads an object state from stable storage using the specified access mode;

put ▷ overwrites the existing stable state of a designated object;

destroy ▷ removes an object state from stable storage.

The class `RobustObject` is also derived from `LockManager` to provide recoverability to the cache object. The operations it exports are:

create ▷ calls the Package operation **create** and registers the newly created object with the Name Server;

make_volatile ▷ locates the object (via Name Server) and calls **refresh_volatile**;

refresh_volatile ▷ calls the Package operation **get** with the designated access mode and restores the object state into the cache object;

make_permanent ▷ saves the cache object into an object state and calls the Package operation **put**;

destroy ▷ calls the Package operation **destroy**.

[3] An introduction about the use of hints, stashing and caching techniques can be found in [8].

[4] This read lock is released as soon as the object state has been read.

5 Stabilis

Stabilis is a fault-tolerant object-oriented distributed database management system built on top of Arjuna. Its design is based on the same principles that governed the design of Arjuna; type inheritance plays a key role in the development of classes that can be used by the database developer to add new qualities to his own classes.

The use of Stabilis entails the design of an object-oriented data model of the database from which database objects (robust objects) are obtained. Stabilis fully benefits from the functionality of the Object Manager module to simplify the implementation of distributed databases and the writing of application programs. In certain cases (Section 5.3), the distributed database can be automatically generated from a description of the object-oriented data model.

The installation and configuration of a new database is straightforward; the only measure that has to be taken is to register the database with the administration tool of Stabilis. All that this tool requires is the set of nodes where the database is to be made available.

In Stabilis, the database objects can have methods to implement services invoked on request, e.g., a database object could have a method (service) to write itself out in Postscript format.

Stabilis has two standard interfaces: a visual database interface [9] and a query interpreter. The visual database interface can mainly be used for short term queries involving small sets of objects. On the other hand, the query interpreter is proper for the processing of longer and more complex queries involving large sets of objects. New interfaces can be programmed as applications that access the database manager.

5.1 Object-Oriented Data Model

The object-oriented data model supported by Stabilis is represented as a *directed acyclic graph* where the nodes are classes, and the edges relationships between them.

A class defines the *domain* of all objects that share the same set of *attributes* and *methods*. The methods of a class are the implementation of the *services* provided by its instances. A relationship between two classes can be any of the following [5]:

- Generalisation/Specialisation: where one class (the *subclass*) inherits attributes and behaviour of another class (the *superclass*). A class with no superclass is called a *root class*. A class with no subclass is called a *leaf class*. *Multipe inheritance* exists when a class has more then one superclass.
- Association: where one class has an attribute whose domain is another class. Such attribute is called a *relational attribute*. It is implemented as a set of objects, whose lower-bound and upper-bound have to be defined.

A class can also be denominated *concrete* or *abstract*, depending on whether it can be instantiated or not [10]. The concrete classes define the classes whose instances are database objects.

[5]A fully object-oriented data model should permit the representation of composite objects.

5.2 Architecture

An application program interacts with the Database Manager (Figure 5) to manipulate objects of a database. This interaction comprehends calling operations of the database manager to access the database objects. A database object is accessed to have its state read and possibly modified by the application; it can be said that the application *edits* the object. The editing of a database object occurs with the help of an *interface object* that acts as a buffer that contains a volatile copy of the database object. This interface object is where an operation of the Database Manager fetches its arguments and returns its results. Database objects and interface objects are created by the Object Generator. The Index is used to resolve queries expedited by the applications.

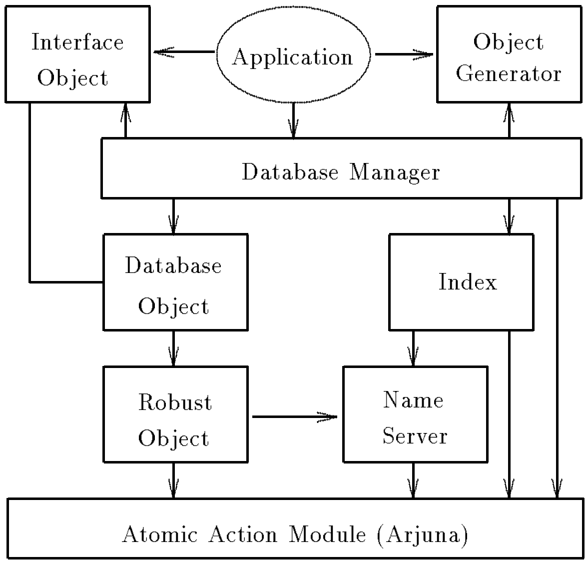

Figure 5: The architecture of Stabilis.

5.2.1 Index Module

The Index module is responsible for evaluating queries formulated by an application program. A query is a declarative specification of a set of objects in the database that satisfy a set of conditions. The conditions are usually specified as a boolean combination of predicates of the form: $<$ a*ttribute_name operator value* $>$, for example, ($title$ = "$Persistent$") **and** ($year > 1980$).

The complexity of the query accepted determines the complexity of the algorithm and data structure used to resolve it. A query may be formulated against either a portion of the object-oriented data model or a single class. It may refer to attributes of a class, a superclass or a related class [11].

In the current version of Stabilis, the queries take as predicates the attributes of a single class. The attributes that can be used as predicates in a condition are denominated *key attributes*. Substrings of a key attribute can be used as values.

Two classes, `IndexManager` and `IndexServer`, implement the Index module. Instances of `IndexServer`, henceforth denominated *index servers*, are robust objects that implement autonomous servers for query processing. Instances of `IndexManager`, henceforth denominated *index managers*, are clients that co-ordinate the operation of the index servers. To the database manager the Index provides an illusion of a virtually centralised query processor.

The main problems that affect the organisation of the Index are:

- concurrency: several queries can be forwarded to an index server concurrently and a reasonable organisation should maximise the concurrency;
- fault tolerance: the data stored in the index servers should always be consistent in the ocurrence of failures. A graceful degradation of availability should occur even when some index servers stop their operation.

The current version of the Index is organised as a cluster of index managers that have references to disjoint sets of index servers. Each set of index servers keep the query data related to only one concrete class of the database. To further optimise the query processing, each index manager keeps a cache of the entries stored in the index servers.

Queries are resolved in the following way: the database manager accepts the query made by the application, e.g., the Query Interpreter, and sends the query expression to the Index. It is then transfered to the class-specific index manager for resolution. The index manager will only forward the query expression to the index servers if its resolution cannot be done using the local cache.

The present organisation of the Index has aimed at solving the problems listed above. The tree-like structure of this organisation permits a faster query processing, by allowing greater concurrency. The Index is well adapted to resolving queries based on attributes of a single class. Additionally, if an index server stops for some reason, only a fraction of the information related to a specific concrete class becomes unavailable.

5.2.2 Database Manager Module

The Database Manager module is the kernel of Stabilis. It provides all basic operations to access database objects.

Nested Atomic Actions Various degrees of operation rollbacks are offered by the database manager to an application program through the use of nested atomic actions. Two atomic actions are used to control the state transitions of the database objects: *outer* and *in-hand*. These transitions are shown in the Figure 6, where they are represented as a finite automaton with two states: *reading* and *writing*. The transitions are caused by the execution of the operations that are structured using both *outer* and *in-hand*.

The atomic action *outer* guarantees that a persistent object is kept locked for write. It is started in every transition from *reading* to *writing* (operations **create** and

mode). And, it can be either ended (operation **mode**) or aborted (operation **abort**) in every transition from *writing* to *reading*. If it ends, all modifications made to the object are made permanent or else, if it aborts, the modifications are ignored and the states of the cache object and interface object retrogress to the previous state saved in the persistent object.

While in the *writing* state, an *in-hand* atomic action is kept running nested within the *outer*. It can be either ended (operation **save**) or aborted (operation **cancel**). If it ends all modifications made to the interface object are saved into the cache object or else, if it aborts, the modifications are ignored and the interface object is kept as it is. In both cases a new *in-hand* is started.

Figure 7 depicts a sequence of database operations and the corresponding atomic actions involved. In this sequence a new database object is created and subsequently modified. In the Figure 7, t_i $(0 \leq i \leq 13)$ marks the instants where relevant events occur; in any of the r_j marks $(1 \leq j \leq 3)$ the database application can either cause a commit or a rollback of the operation being executed.

At t_0, the operation **create** is started and as a consequence:

- *outer* is started;
- *in-hand* is started;
- A is started to ensure that the creation of the database object (atomic action B) and its registration with the Index (atomic action E) comprehend an atomic event (t_1 to t_4);
- B is started to guarantee that the persistent object is created (atomic action C) and registered with the Name Server (atomic action D) atomically (t_2 to t_3);

Between t_4 and t_5 the application makes alterations to the interface object. At t_5 the operation *save* is executed to confirm the alterations. At this point, F is started to ensure that the update of the object (atomic action G) and the update of the Index (atomic action H) are atomically executed. The present *in-hand* is committed but the changes are not made permanent yet because *outer* is still running. Another *in-hand* is started at t_8.

Between t_8 and t_9 the application modifies the interface object again. At t_9 the **cancel** operation is invoked to roll the state of the interface object back to the state it had at t_8. The present *in-hand* is aborted and a new one is started at t_{10}.

From t_{10} to t_{11} more alterations are made to the interface object. Finally, at t_{11}, the operation **mode** is executed to make permanent all modifications made since t_0. The same update process executed between t_5 and t_6 is repeated in the interval t_{11} to t_{12}. At t_{13} the *in-hand* and *outer* are ended.

Operations The operation of the database manager is supported by an *operational context*. This operational context is defined by the selection of a class, the *current class*, and by the selection of an object of that class, the *current object*. Two operations, **select_class** and **select_object**, are defined to allow an application to

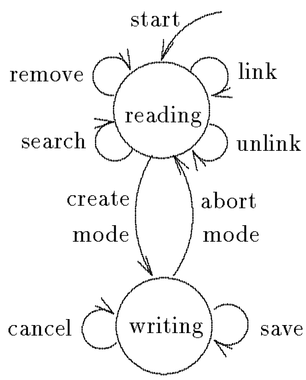

Figure 6: Transition diagram for the finite automaton of Stabilis.

traverse the database by changing the operational context. Some of the operations of the database manager are:

mode ▹ This operation allows the user to change the mode of access to the current object. The current object can be either in *writing* or in *reading* mode.

create ▹ Creates an instance of a database object in any of the object stores available. In the present implementation, an object store is selected at random. Other selection algorithms can consider factors such as: the space available in the object stores, frequency of access to a database object, etc, to decide where to create the database object. Every database object created is registered with the Name Server and has its key attributes entered in the Index.

remove ▹ All reference information related to the current object is removed from the modules Index and Name Server, and the current object is deleted from the database.

save ▹ All changes made up to this point to the current object are made permanent, including the corresponding Index entry.

cancel ▹ All changes made to the current object since the last save operation are ignored; the state of the current object is retrogressed to the last saved state.

abort ▹ The state of the current object is retrogressed to the state it had before its last transition from *reading* to *writing* state; all changes made by operations executed during this interval are ignored.

search ▹ The search operation takes as input a query and returns the set of objects whose key attributes satisfy the query condition.

serve ▹ In Stabilis, database objects can have service operations that are executed on request. A request, triggered by an application program, causes the execution of a designated object method that implements that service operation. For

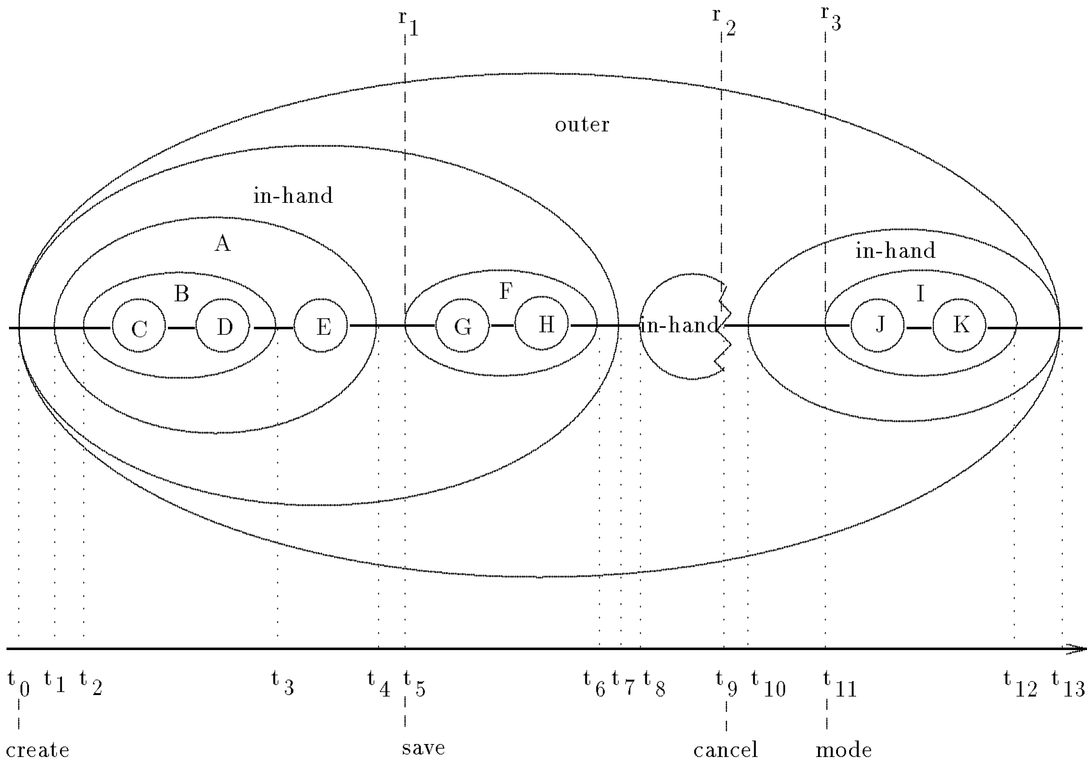

Figure 7: Nested atomic actions: a snapshot.

example, a database object can have a method that implements the service operation *encrypt* to cryptograph its attributes.

relate ▹ In Stabilis, relations are always formed between a pair of objects, the current object, and another one, the *relational object.* When the **relate** operation is executed the current object becomes the relational object, and the current class the *relational class.* Subsequently, a new operational context is set. After the execution of a **relate** operation several link/unlink operations can be executed to do/undo relations between the relational object and the current object. Stabilis allows nesting of **relate** operations.

link ▹ Relations among database objects are expressed through links. A link is a reference to another database object. This operation always involves the current object and a relational object previously selected. When **link** is executed the unique identifiers of both objects are included as attribute values of their relational attributes. The reciprocal operation **unlink** is provided.

resume ▹ Concludes the process initiated by the execution of a **relate** command. The relational class becomes the current class, and the relational object the current object.

hibernate ▷ The current state of the database manager is saved as a persistent object. This operation can be used to migrate the state of the database manager among the nodes of the network.

reinstate ▷ The reciprocal of the operation **hibernate**. When a **reinstate** is executed the state of the database manager is set to exactly the same execution state it had when the corresponding **hibernate** was executed.

5.2.3 Standard Interfaces

A database user accesses the database either by issuing queries to the Query Interpreter or by using the visual database interface Object Editor.

Visual Database Interface The Object Editor is logically composed of two interactive interfaces: a main interface where all commands and query results are shown, and a set of application-specific interfaces corresponding to each concrete class of a particular database. An example is the interface for the class `Article`. When a user wants to change attribute values for instances of `Article` all that he has to do is to select this class, using a command available in the main interface, and the `Article` interface will be made available. At this point he can alter the attribute values for any instance of this class, create relations between objects, submit queries, etc.

Query Interpreter The Query Interpreter executes sequences of database operations coded as simple scripts. It is comparable to the command shell of an operating system. This interface is still under development.

5.3 Developing Databases and Application Programs

The development of a database for Stabilis starts with the creation of an object-oriented data model of the database, where all classes and its relationships are represented. Next, the concrete classes defined in the model have to be converted into classes whose instances are database objects. This conversion is achieved by making every root class a specialisation of the class `DatabaseObject`, a class derived from `RobustObject`.

The database manager will have to instantiate objects of the concrete classes. To enable it to instantiate such objects, a class called `DatabaseObjectGenerator`, which is capable of instantiating objects of the concrete classes, has to be programmed. An instance of `DatabaseObjectGenerator` will be used by the database manager.

At this stage the database manager is already prepared to manipulate any instance of the concrete classes of the database model internally. Another problem that has to be solved is how an application program will have access to the database objects. The solution is to create an interface object to convey information between the application program and the database manager. To permit the creation of the interface objects, the class `InterfaceObject` is provided; the same root classes that were made subclasses of `DatabaseObject` have to be made subclasses of `InterfaceObject`. Similarly, an `InterfaceObjectGenerator` class has to be programmed.

The programming steps above can be automated if all the attributes of the database classes are expressed in terms of C++ basic types (integers, strings, etc). In fact, we have implemented a tool called hgen that takes as input a description of an object-oriented data model and generates the corresponding C++ code for: the classes derived from `DatabaseObject`, the classes derived from `InterfaceObject`, the `DatabaseObjectGenerator` and the `InterfaceObjectGenerator`. As a consequence, applications that use Stabilis can be easily programmed since all classes needed to access the database have already been produced. An application has only to instantiate interface objects and operate on them according to its own algorithm. The application program will necessarily call operations of the database manager.

If the two standard interfaces, Object Editor and Query Interpreter are to be used then some extra work has to be done. The Object Editor uses InterViews [12] to implement the visual interfaces for editing database objects. Consequently, adapting the Object Editor requires the programming of a visual interface for each concrete class of the new database. Again, to automate the adaptation process of the Object Editor to different databases we have programmed a tool, denominated ivgen, to generate the C++ code necessary. This tool accepts as input the same description of an object-oriented data model that hgen accepts, generating the InterViews C++ code for the visual interfaces. The compilation and linking of the programs is all that needs to be done to obtain the new Object Editor version. The adaptation of the Query Interpreter is much simpler since there is no extra code that needs to be generated. It is only to compile and link the code produced by hgen to the code of the Query Interpreter.

In most cases, when the two standard interfaces to Stabilis satisfy the needs of the database user, the only real task the database developer faces is the design of an object-oriented database model.

6 Dbib - A Distributed Bibliography Database

This Section explains further, using an example, how databases and applications are developed using Stabilis. Lets suppose someone wants to design and implement a distributed bibliography database (Dbib) based on the types of entries for bibliography citation defined by Lamport in the LaTeX Manual [13]. Lamport's original definition has 14 types of entries, ranging from an entry for storing the bibliographic information about an *article* to an entry for *unpublished notes*. Each entry can have several fields. An article entry, for example, has fields for storing the article's authors, title, etc. Basically, in the object-oriented data model of Dbid (Figure 8) theses entries will be viewed as concrete classes. Later, when using Dbib, the user will be able to call object services to generate the file necessary to create bibliography lists for LaTeX.

The object-oriented data model (Figure 8) defines 18 concrete classes, five root classes and several associations between them. An example of association is found between `Article` and `Journal`; an article is present in one single journal and a journal can contain several articles. This association is implemented by the superclasses

`Include` and `Group`.

Since all attributes defined in the data model are either strings or integers, the implementation of Dbib can be fully automated. First, the data model has to be coded using a graph description language. Second, this textual representation obtained is submitted to `hgen`. Figure 9a shows a description of the path

$$P =< Reference, Unit, Include, Article >$$

using the graph description language, which is self-explanatory.

Figures 9a and 9b juxtapose the representation of the path P using the graph description language (Figure 9a) with fragments of the corresponding C++ code generated by `hgen` (Figure 9b). In both Figures the lines are numbered to help explain the code. From now on we will refer to Figures 9a and 9b simply by stating line numbers followed by the the letter that identifies each Figure, e.g., 11b refers to the line 11 of the Figure 9b.

The code of the line 1a means that the root class `Reference` must be a class derived from the class `DatabaseObject` (line 1b).

The attributes defined in lines 3a to 6a generated the lines 3b to 6b and 13b to 16b. Line 5a is an example of the specification of a relational attribute: `libraries`, that is mapped to a variable of the type `SetOfObjects` (line 5b). This type is a set of unique identifiers of database objects used to implement the association relationship defined in the object-oriented data model. The parameters mean that: the class `Reference` is associated with the class `Library`, the corresponding relational attribute is `references` and an instance of the class `Reference` can be associated to several instances of the class `Library`. Line 9b shows the signature of the function that is used to initialise the data structures that keep the relations for the class `Reference`.

The code between the lines 1a to 22a are essential to specify the derivation path of the class `Article`, instructing `hgen` that the class `Article` inherits the attributes and association relationships of these superclasses.

The code in the lines 21a and 22a means that the class `Article` must be derived from the class `Include` and that it is a concrete class. The code of the line 18b implements this specialisation relationship. The fact that `Article` is a concrete class implies that the `DatabaseObjectGenerator` (not shown here) has to provide the code to instantiate objects of the class `Article`.

The functions **get** (lines 14b to 16b) and the function **put_attributes** (lines 23b to 26b, 28b to 42b) are called by an instance of the class `InterfaceObject` to read and write the state of the database object. For example, after modifying an `Article` interface object, the application program invokes the database manager operation **save**, which will then call the function **put_attributes** to update the database object.

The operation **put_attributes** must be atomic since it modifies the state of the database object. In the line 32b an atomic action A is declared. In the line 33b it is started. The attributes of the cache object are modified (lines 34b to 36b). Lines 38b to 40b have the code for making permanent these modifications. Line 38b shows the invocation of the operation **make_permanent** of the superclass `RobustObject`. If **make_permanent** commits then A can be committed otherwise it is aborted.

This code guarantees the consistency of the cache object, as already discussed earlier (Section 4.2).

7 Concluding Remarks

This paper has presented Stabilis and described how its architecture has been realised using an object-oriented approach. This approach, already adopted by Arjuna, permits the properties of robust objects to be added to user-defined classes in a very flexible manner. Within this approach, type inheritance has also proved useful to simplify the automatic generation of code. The use of C++ in Arjuna and Stabilis has demonstrated to have both advantages and disadvantages. Since both systems make extensive use of inheritance and encapsulation, these features of the language have been of great use. A disadvantage is that the version of C++ used does not have automatic garbage collection of unreferenced objects.

The development of the Name Server, Object Manager and Index modules as portable applications has been possible since Arjuna encapsulates the essential functions for persistence, recoverability and atomicity in its Atomic Action module.

Although the *object and action* model adopted by Arjuna to structure programs has proved to be very adequate to the development of Stabilis as a whole, the provision of extra ways of structuring atomic actions could greatly improve the concurrency control. For example, the Name Server and Index modules would benefit if nested top-level atomic actions [14], currently being incorporated into Arjuna, were available.

The development of the database manager has been greatly simplified by the abstraction of robust object provided by the Object Manager module, since it exempts the database manager of the direct management of persistence, distribution, concurrency and recovery. Moreover, the use of caching has shown to be useful to the efficiency of Stabilis; the database manager can optimise the use of persistent objects by reducing the frequency of accesses to stable storage.

To evaluate Stabilis we have designed and implemented Dbib. At present, Dbib is installed in five nodes and holds aproximately three hundred bibliographic references. After having carried out several tests, we have realised that some improvements can be made to Stabilis. The query processing can be made more powerful, e.g., queries against a portion of the data model should be accepted. Stabilis should have mechanisms for the representation of aggregation of objects. An authorisation scheme should be implemented to improve security. On the other hand, Stabilis has shown that the development of fault-tolerant distributed databases can be made simple. Also, some of the tests involved forcing nodes to crash; in these situations Stabilis continued to operate consistently.

As far as this case study is concerned, the overall behaviour of Stabilis and Arjuna has been satisfactory; the concurrent use of Dbib has shown that the basic requirements set out initially have been fulfilled. In conclusion, the *object and action* model of computation has demonstrated to be a good framework for writing fault-tolerant distributed applications that use persistent objects.

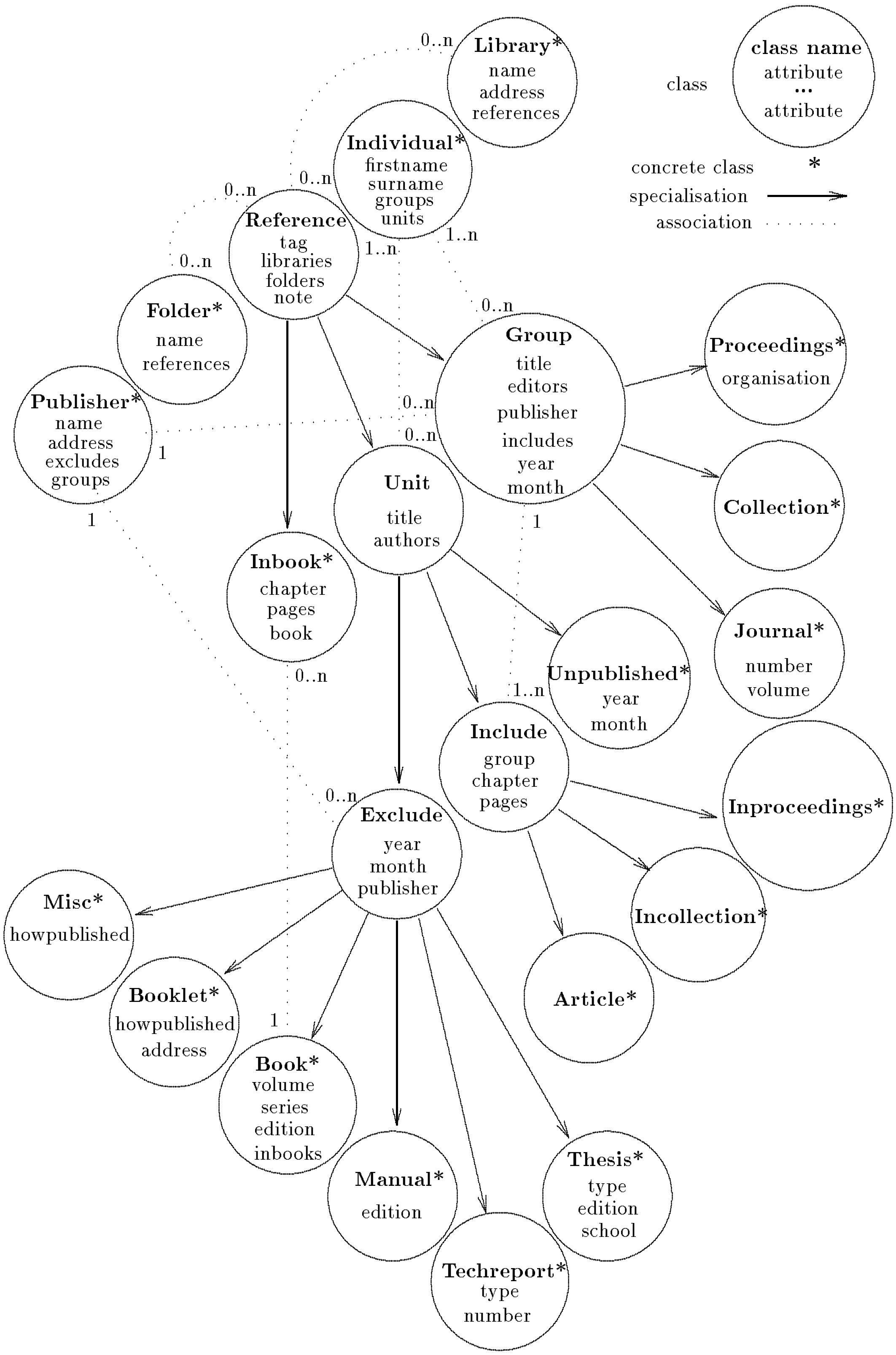

Figure 8: An object-oriented model for Dbib.

```
root class   Reference;
begin attributes
    tag:      SearchKey;
    note:     String;
    libraries: Relational
      (Library, references, 0, -1);
    folders:  Relational
      (Folder, references, 0, -1);
end attributes;

class   Unit;
parent  Reference;
begin attributes
    title:    String;
    authors: Relational
      (Individual, units, 1, -1);
end attributes;


class   Include;
parent  Unit;
begin attributes
    group:    Relational
      (Group, include, 1, 1);
    pages:    String;
    chapter: String;
end attributes;


concrete class   Article;
parent     Include;
begin attributes
end attributes;
```

(a)

```
class  Reference: public  DatabaseObject
{ protected:
    String tag;
    String note;
    SetOfObjects libraries("Library", "references", 0, -1);
    SetOfObjects folders("Folder", "references", 0, -1);
    virtual   bool restore_state(ObjectState&);
    virtual   bool save_state(ObjectState&);
    virtual   void relations(RelationTable&);
  public:
    Reference();
    ~Reference();
    virtual   String get_SearchKey();
    String get_tag();
    SetOfObjects get_libraries();
    SetOfObjects get_folders();
};
    ...
class  Article:   public  Include
{ public:
    Article(Uid&, bool&);   // constructor for creation
    Article(Uid, Lockmode, bool&);    // for access
    ~Article();
    bool put_attributes(String new_tag, ...
                  SetOfObjects new_libraries, ...
                  String new_title, ...
                  String new_chapter);
};  ...

bool Article::put_attributes(String new_tag, ...
                SetOfObjects new_libraries, ...
                String new_title, ...
                String new_chapter)
{ AtomicAction A;
  A.Begin();
      tag = new_tag; ...  libraries = new_libraries;...
      title = new_title; ...
      chapter = new_chapter;
  bool result = FALSE;
  if    (make_permanent() == COMMITTED)
        result = (A.End() == COMMITTED);
  else A.Abort();
  return result;
}  ...
```

(b)

Figure 9: (a) Path P coded in the graph description language. (b) C++ code for path P.

Acknowledgements

Our thanks are due to Professor Santosh K. Shrivastava and Dr. Stuart M. Wheater for their critical comments on this paper. The Arjuna team also deserves our gratitude; they have been very supportive throughout the development of Stabilis.

References

[1] Shrivastava, S.K., Dixon, G.N., Parrington, G.D. “An Overview of the Arjuna Programming System”, IEEE Software, **8**(1):66-73, Jan. 1991.

[2] Shrivastava, S.K. “Robust Distributed Programs” In Resilient Computing Systems, T. Anderson (editor), Collins, London, 1985, ch. 6, pp. 102-121.

[3] Weihl, W.E. “Using Transactions in Distributed Applications” In Distributed Systems, S. Mullender (editor), ACM Press, New York, 1989, ch. 11, pp. 215-235.

[4] Stroustrup, B. “The C++ Programming Language”, Addison Wesley, Reading, Massachusetts, 1986.

[5] Shrivastava, S.K. and Wheater, S.M. “Implementing Fault-Tolerant Distributed Applications Using Objects and Multi-Coloured Actions”, Proceedings of the Tenth International Conference on Distributed Computing Systems, Paris, France, May 1990, pp. 203-210.

[6] Parrington, G.D. “Reliable Distributed Programming in C++ : The Arjuna Approach”, Second Usenix C++ Conference, San Francisco, USA, Apr. 1990, pp. 37-50.

[7] McCue, D.L. and Shrivastava, S.K. “Structuring Persistent Object-Oriented Systems for Portability in a Distributed Environment”, Fourth ACM SIGOPT Workshop, Bologna, Italy, Sep. 1990.

[8] Mullender, S. “Introduction to Distributed Systems” In Distributed Systems, S. Mullender (editor), ACM Press, New York, 1989, pp. 3-18.

[9] Wu, C. T. “Benefits of Object-Oriented Programming in Implementing Visual Database Interfaces”, Journal of Object-Oriented Programming, **2**(6):8-16, Mar./Apr. 1990.

[10] Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F. and Lorensen, W. “Object-Oriented Modeling and Design”, Prentice-Hall, Englewood Cliffs, New Jersey, 1991.

[11] Kim, W. “Architectural Issues in Object-Oriented Databases”, Journal of Object-Oriented Programming, **2**(6):29-38, Mar./Apr. 1990.

[12] Linton, M.A., Vlissides, J.M. and Calder, P.R. "Composing User Interfaces with InterViews", IEEE Computer, **22**(2):8-22, Feb. 1989.

[13] Lamport, L. "The Bibliography Database" In "LaTeX: User's Guide and Reference Manual", Addison Wesley, Reading, Massachusetts, 1986, pp. 140-147.

[14] Wheater, S. M. "Constructing Reliable Distributed Applications Using Actions and Objects", PhD thesis, The University, Computing Science Department, Newcastle upon Tyne, Jun. 1990 (Technical Report no. 316).