Management of Object-Oriented Action-Based Distributed Programs

Luiz Eduardo Buzato

Ph.D. Thesis

Department of Computing Science
University of Newcastle upon Tyne

October 1994

◇

À Carla;
Luzia e René;
Silvana e Ronaldo;
Antonia, Maria José e
Ruth; em memória de
Joana e Miguel.
Pelo amor e
paciência.

◇

Contents

BLANK PAGE
IN
ORIGINAL

List of Figures

List of Tables

BLANK PAGE
IN
ORIGINAL

List of Classes

BLANK PAGE
IN
ORIGINAL

BLANK PAGE
IN
ORIGINAL

List of Programs

BLANK PAGE
IN
ORIGINAL

Abstract

This thesis addresses the problem of managing the runtime behaviour of distributed programs. The thesis of this work is that management is fundamentally an information processing activity and that the *object model*, as applied to *action-based* distributed systems and *database systems*, is an appropriate representation of the management information. In this approach, the basic concepts of classes, objects, relationships, and atomic transition systems are used to form object models of distributed programs. Distributed programs are collections of objects whose methods are structured using atomic actions, i.e., atomic transactions. Object models are formed of two submodels, each representing a fundamental aspect of a distributed program. The structural submodel represents a static perspective of the distributed program, and the control submodel represents a dynamic perspective of it. Structural models represent the program's objects, classes and their relationships. Control models represent the program's object states, events, guards and actions—a transition system. Resolution of queries on the distributed program's object model enable the management system to control certain activities of distributed programs.

At a different level of abstraction, the distributed program can be seen as a reactive system where two subprograms interact: an application program and a management program; they interact only through sensors and actuators. Sensors are methods used to probe an object's state and actuators are methods used to change an object's state. The management program is capable to prod the application program into action by activating sensors and actuators available at the interface of the application program. Actions are determined by management policies that are encoded in the management program. This way of structuring the management system encourages a clear modularization of application and management distributed programs, allowing better separation of concerns. Managemental concerns can be dealt with by the management program, functional concerns can be assigned to the application program.

The object-oriented action-based computational model adopted by the management system provides a natural framework for the implementation of fault-tolerant distributed programs. Object orientation provides modularity and extensibility through object encapsulation. Atomic actions guarantee the consistency of the objects of the distributed program despite concurrency and failures. Replication of the distributed program provides increased fault-tolerance by guaranteeing the consistent progress of the computation, even though some of the replicated objects can fail.

A prototype management system based on the management theory proposed above has been implemented atop Arjuna; an object-oriented programming system which provides a set of tools for constructing fault-tolerant distributed pro-

grams. The management system is composed of two subsystems: Stabilis, a management system for structural information, and Vigil, a management system for control information. Example applications have been implemented to illustrate the use of the management system and gather experimental evidence to give support to the thesis.

Acknowledgements

Professor Santosh Shrivastava likes examples and often uses them to reify his ideas. Therefore, during our discussions he was always motivating me to explain my ideas using concrete but simple examples. During his interpretation of the examples, Santosh became himself an example of professionalism tempered with wisdom. I thank him for his criticisms and encouragement during every stage of the development of my work: from helping me to decide which research topic to pursue to carefully proof-reading drafts of papers and of this thesis; I thank him for fruitful years of supervision.

I am grateful to many people who are now or have been connected with the Arjuna team and have helped me with my work. I owe a special debt to Mr. Alcides Calsavara, who has been my partner in the implementation of Stabilis. We have learnt that debugging layers upon layers of distributed software is not easy and is very time consuming. Dr. Graham Parrington helped me a lot with the steering of Arjuna's stub generator; he also helped me with the debugging of Arjuna, Stabilis, and Vigil. Dr. Stuart Weather implemented distributed atomic actions in Arjuna. So, he had to endure long sessions of questions on the use of atomic actions, and on how they are implemented in Arjuna. Also, I thank Stuart for his readiness in installing new versions of Arjuna when I needed them. Replication is Dr. Mark Little's area of specialization, during the design of Vigil we had several discussions on how replication could be used to Vigil's advantage. Additionally, I thank all the members of the team for their collaborative role in the creation of a friendly research environment.

I would like to say "Muito obrigado" to Ms. Shirley Craig for having helped me to navigate our libraries during these years.

I would like to thank the members of the Department of Computing Science of The University of Campinas for their co-operation. In special, I would like to thank Drs. Ricardo Anido, Hans Liesenberg, and Ricardo Dahab for having managed so well my leave. I am grateful to Hans for his immense friendship; he took over my lecture assignment for the second semester of 1994, so that I could commit this work.

Financial support for much of the work presented in this thesis was provided by CNPq (Consellho Nacional de Desenvolvimento Científico e Tecnológico, Brazil) under grant number 200410/88-1, and from the project BROADCAST (Basic Research On Advanced Distributed Computing: from Algorithms to SysTems) under grant number 6360.

BLANK PAGE
IN
ORIGINAL

Introduction

Motivation

Our society is increasingly and irreversibly dependent upon computing systems, and consequently, on their reliability. Computing systems are being used in all types of applications, ranging from mass-produced televisions to products that are produced in small numbers such as nuclear power stations.

Reliability of computing systems is a very important requirement to all these products. For mass-produced products, and service products like banking and telephony, reliability is a monetary issue. If mass-produced products prove to be unreliable, then their producers are liable to suffer monetary losses. Similarly, if a bank has its services disrupted because of unreliable computing systems, then monetary losses are usually inevitable. For systems like nuclear power stations, railway control systems, and airspace control systems, unreliability of computing systems can cause serious human and/or monetary losses. Therefore, reliability is a key issue in the development of computing systems, in particular, distributed systems.

Distributed systems became possible due to coevolution of computing and communication systems. Co-operation between components of a distributed system to execute a task and maintain some distributed shared data readily available to its users is the major benefit obtained from distributed computing. However, reliable and useful operation does not come without costs. Because distributed systems are composed of many components that can fail independently, the task of writing distributed programs that behave dependably in the presence of partial failures is very difficult. Programmers of distributed systems have to stay in control of not only the standard system activities when all components are well, but also of the complex situations that can occur when some components fail. With the ever increasing dependence of our society upon computing services, the demand for reliable distributed systems is likely to increase. Consequently, the size and complexity of distributed applications is bound to increase and make the programming of distributed systems even more difficult. In such scenario, it is crucial, for reliability purposes, to have a programming environment able to

provide simple and well-integrated programming tools for implementation and *management* of distributed systems.

Objective

The aim of this thesis is the implementation of a novel environment for management of object-oriented action-based[1] distributed programs. Object-oriented action-based distributed programs are composed of interacting objects that are instances of classes whose methods have been structured using atomic actions. Management involves the processing of dynamic information concerning a distributed program, as the program executes, and the enforcement of changes upon it to make it conform to a desired management policy.

The thesis of this work is that management is fundamentally an information processing activity and that the *object-oriented* model, as applied to *action-based* programming systems and database systems, is an appropriate representation of the management information. In this approach, the basic concepts of object, classes, relationships, states and atomic transitions are used to form an object model of distributed programs. The resolution of queries on object models enable the management system to control certain activities of distributed programs.

The management thesis proposed above entails the separation of information about the distributed program being managed, i.e., metainformation, from information concerning the distributed program's intrinsic function. There are two categories of metainformation: structural metainformation and control metainformation. An object model is composed of two submodels, each representing one of these two categories of metainformation. Classes and their relationships form the structural submodel; this submodel represents a static perspective of the distributed program. A control submodel is a representation of a transition system; in the control submodel the distributed program's object states and transitions between states are represented. States in the control submodel represent states of objects of a distributed program. In the control submodel edges between states represent transitions from an object state to other object state. These transitions are atomic and can only be traversed when their corresponding guards are satisfied. Transitions in the control submodel are implemented as guarded actions. The control submodel represents a dynamic view of a distributed program. The metainformation represented by the object model is dynamic, it reflects changes in the relationships between the objects of a distributed program. It is the processing of such metadata that makes possible the management of distributed programs.

[1] In this context, *action* is a synonym with *transaction* or *atomic action*.

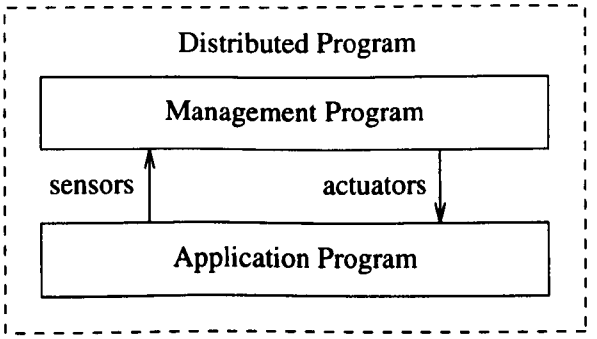

Figure I.1: Distributed program seen as a reactive system.

At a different level of abstraction, the distributed program can be seen as a reactive system where two subprograms interact: an application program and a management program; they interact only through sensors and actuators (Figure I.1). Sensors are methods used to probe an object's state and actuators are methods used to change an object's state. The management program is capable to prod the application program into action by activating sensors and actuators available at the interface of the application program. Actions are determined by management policies that are encoded in the management program. This way of structuring the management system encourages a clear modularization of application and management distributed programs, allowing better separation of concerns. Management concerns can be dealt with by the management program, functional concerns can be assigned to the application program.

A prototype management system has been built to gather experimental evidence to give support to the thesis. The prototype is composed of two subsystems: Stabilis and Vigil. Each subsystem manages information related to one of the two submodels identified above. Stabilis is the subsystem responsible for the management of structural information. The name of the subsystem reflects its role, Stabilis is a Latin word synonymous with stable, firm. Vigil is the subsystem responsible for the management of control information. Vigil is a Latin word synonymous with guard, sentinel. Sentinels are constantly aware of the changes in the surrounding environment with the sole objective of timely reacting to relevant events. Both subsystems have been implemented atop of Arjuna; an object-oriented programming system which provides a set of tools for constructing fault-tolerant distributed programs. A layered view of the systems is shown in Figure I.2; the programming interface allows access to any of the tools provided not only by the management system but also by Arjuna. Atomic actions and replication are two of the tools provided by Arjuna that are important to the implementation of Stabilis and Vigil. Fault-tolerant management is only possi-

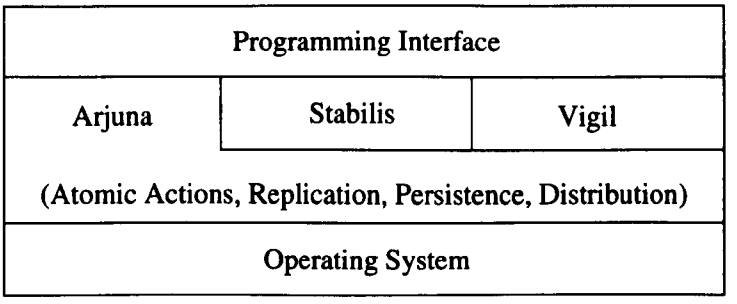

Figure I.2: Layered diagram of the management system.

ble because Stabilis and Vigil can rely on atomic actions and replication. The programming language adopted by the three systems is C++; it has not been adapted or extended.

The first step a programmer has to follow to be able to use the management system is the creation of an object model of a distributed program. Next, the object model is captured and represented as a database schema. The information stored in the database schema is used to generate part of the C++ code of both application and management programs. Subsequently, both programs are compiled and their object code is linked with code in the libraries of Arjuna, Stabilis, and Vigil (Figure I.3). During the execution of the distributed program, the objects of the application program are constantly monitored by the objects of the management program. Changes in the state of the objects of the application program generate stimuli that trigger the execution of guarded actions, that is, reactions, in the management program. Reactions cause further state changes in the application program. Stimulus-reaction cycles are intrinsic to management systems.

The work described in this thesis is synthetic in nature, bringing together techniques and insights from several branches of computing science. Throughout the work, object orientation is presented as a unifying paradigm that can be used to integrate ideas coming from distributed operating systems, database systems, and distributed programming systems. Management systems can benefit from this relative unification.

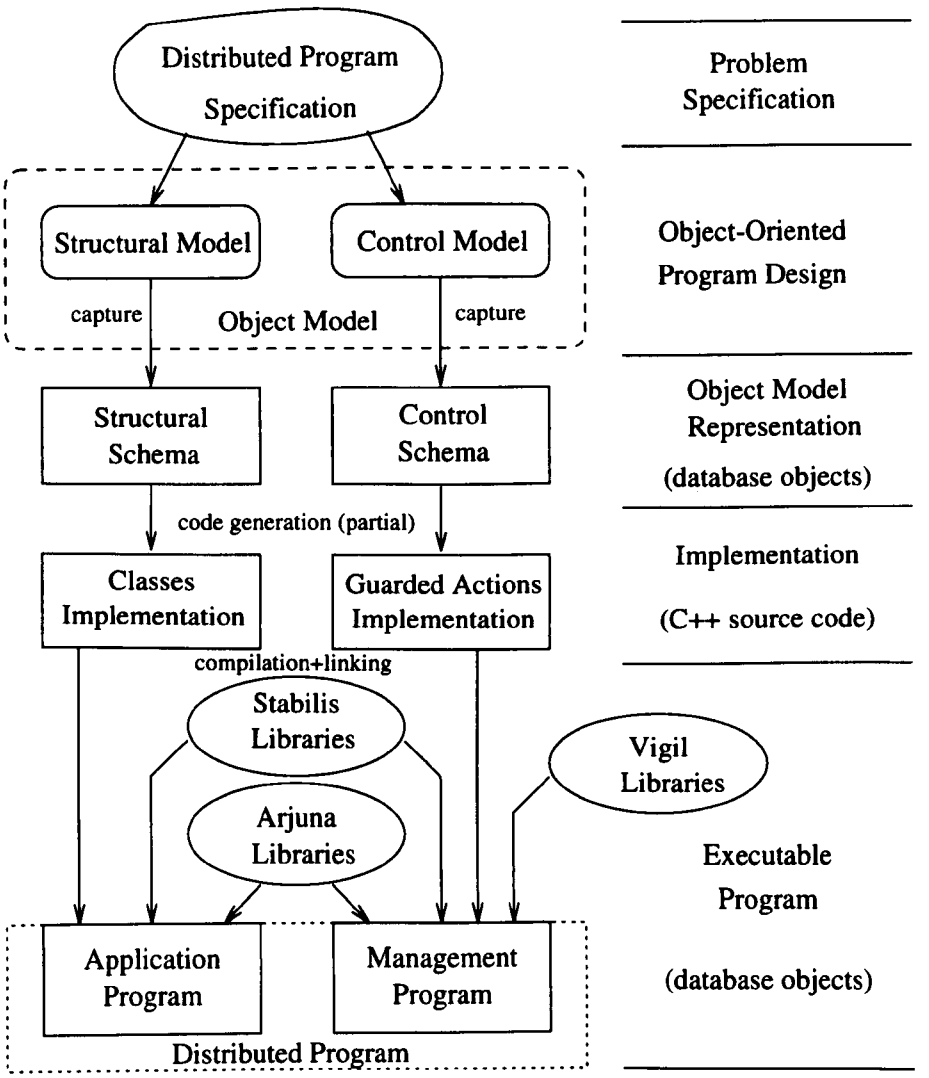

Figure I.3: Steps followed to implement distributed programs.

Thesis Organization

Chapter 1 introduces fundamental concepts about distributed systems, computational models and management of distributed programs. The main reasons for the inclusion of a Chapter on fundamentals of distributed computing at this point are to make the thesis self-contained and more readable. Chapter 2 presents a review of research work that have had influence in the way Stabilis and Vigil have been designed and implemented. Chapter 3 presents the design and implementation of Stabilis, the prototype system for management of structural information. Chapter 4 presents the design and implementation of Vigil, a prototype system for management of control information. Chapter 5 demonstrates the use of the management system through examples. Finally, in the Conclusions we ponder on the results obtained so far, suggesting corrections and pointing out directions for future research.

Chapter 1

Distributed Systems

For our purposes, a *system* [8, pages 6-13] is a set of components which interact to deliver a service. The components of a system are themselves systems. The interaction among the various components of a system is controlled by an *algorithm*. An *atomic* system is a system whose internal structure can be ignored.

A system interacts with its *environment*, responding to stimuli at an interface between the system and the environment. An *interface* is simply a place of interaction between two systems. Consequently, the environment must be another system. The external behaviour of a system can be described in terms of a finite set of *external states*, together with a function defining transitions between states. The environment provides stimuli to its system and perceives it only through discrete transitions from one external state to another.

The *external behaviour* of a system is the manifestation of activity within the system, and for a non-atomic system this activity can be observed. The *internal state* of the system is defined to be a tuple composed of the external states of the components of the system. A function maps internal states to external states. The pattern of internal state transitions is specified and controlled by the algorithm of the system. The *reliability* of a system is defined as its ability to deliver its services, as dictated by its algorithm.

The definitions above have been made general enough that they can serve two purposes. First, they are applicable to distributed systems and management systems. Second, they can be applied at many different levels in such systems.

1.1 A Definition

An interesting definition of a distributed system is due to Leslie Lamport:

> *"You know you have one [distributed system] when the crash of a computer you have never heard of stops you from getting any work done."*

Lamport's definition implies the existence of some machines, other than the ones he knows about, whose crash have prevented him from doing the job he had been doing in his own machine. From this scenario one may infer that a distributed system is composed of at least two interconnected computers. The actual architecture of the system is immaterial to this stage of discussion. What is important is that Lamport's job depended on the co-operation of these computers to deliver services that were critical to the accomplishment of his work. Furthermore, these computers had to be interconnected to each other. Otherwise, the crash of one computer would not have interfered with the functioning of another. From this, one can infer that a distributed system has at least three attributes:

- *Multiple Computers*: A distributed system contains more than one physical computer each consisting of CPUs, local memory, possibly some stable storage and I/O paths to connect it with the other computers of the system.

- *Interconnections*: Some of the I/O paths will interconnect the autonomous computers. The interconnections can vary in their specification from twisted pairs of wire to coaxial cables, or bus-like I/O paths. Different communication protocols may be used to transfer information among machines.

- *Shared State*: The computers of a distributed system co-operate to maintain some shared state available to their users, usually through the execution of distributed co-operative programs.

The motivations for building distributed systems are various and one of the most important is *independence of failure*. Because there are multiple computers in a distributed system, when one breaks down others may still be operational. Ideally, users of a distributed system should expect to continue working even when some computers are not working. Increased availability and reliability, achieved through replication of components, are very important benefits offered by distributed systems. Such benefits can be easily nullified if measures are not taken during the design and implementation of the system to minimize unwanted dependencies. Again, Lamport's definition might set us thinking about independence of failure and the disastrous consequences its absence can cause to the operation of a distributed system.

Distributed systems are built by interconnecting computers through communication links. On the one hand, that means that distributed systems, unlike centralized ones, can grow incrementally and can, potentially, deliver increased computing power at lower costs. On the other hand, communication links may not work correctly all the time. Links may be unavailable, messages may be lost or corrupted. In short, one computer cannot rely on being able to communicate with another, even if both are operational. Maybe Lamport's distributed system stopped working because a lightning struck a public telephone line which was part of a communication link used by the distributed system he was using. *Unreliable communication* is a source of complexity when programming distributed systems.

Communication among subsystems of a distributed system can occur at very different speeds. Differences in communication speeds complicate the design and implementation of distributed programs. For example, interconnections among computers usually provide lower bandwidth, higher latency, and higher communication cost than that available between independent processes within a single machine.

An important benefit of distributed systems is analogous to that offered by a public telephone network but in an amplified way—information sharing. Like a

public telephone network, a distributed system allows physically distant users to share information, only in a more integrated and faster way. Telecommunication and computer industries all over the world have recognized this analogy. They are investing large sums of money in creating distributed computing systems which will gradually substitute the public telephone network. In order to be shared, computing resources have to be found, and to be found they have to be identified, named. So, the assignment to and processing of names of resources in distributed systems is important.

Distributed systems act as a double-edged sword when *security* is considered. Recent advances in cryptography have lent distributed systems an apparent greater advantage over centralized systems because, unlike centralized systems, breaking into one computer does not compromise the entire system. The down side of security in distributed systems is that the potential number of points vulnerable to attack is enormous.

Finally, distributed system management can be considered as the adjustment of system state by a manager, be it a management program or a human manager. There are certain aspects of a system that cannot be defined only in terms of the function of the system itself because they require the observation of the environment where the system is embedded. For example, whether a replicated object will help improve the performance of the distributed system as a whole depends on its internal construction, its function, and where it is placed in the distributed system, its management. If the function of the component system, as seen at its interface, is appropriate to its environment, or vice versa, then the system will serve its intended purpose. Thus, if an object is *close* to its clients, then it might help improve the performance of the system as a whole. Otherwise, if it is not *close*, the management program may have to act by moving it closer to where it is needed. This division between functional and non-functional, or managemental aspects of systems is very important to the implementation of distributed system managers. When the design and implementation of a component fails to recognize this natural division of concerns then functional and managemental aspects become entangled and the distributed system as a whole becomes very difficult to be managed. Reconfigurations become very costly. Thus, distributed systems should provide programming tools which help in the design and implementation of components where functional and managemental concerns are carried out at the right level. Management of distributed systems is an important property that software engineers know least well how to achieve [100].

This discussion about the advantages and disadvantages of distributed systems can set us pondering how the designers of distributed systems choose computation models and mechanisms adequate to the requirements of the distributed systems they are building. Transparency is an important requirement designers of distributed systems have to consider. Independence of failure and fault

tolerance are also very important. The next Section lists various transparency requirements for distributed systems, setting the foundation for our revision of distributed systems carried out in the next Chapter.

1.2 Transparency

The Oxford Advanced Learner's Dictionary [72] defines *transparency* as "being the state of transparent" and *transparent* as "allowing light to pass through so that objects behind can be seen clearly." The word *transparency* has a semantic element of *no interference*. It is this aspect of the definition of the word transparency that we have to explore to understand it as an attribute of distributed systems.

Let us recall that before distributed systems existed programmers were used to writing programs for single-processor computing systems. For instance, when writing a program for a single-processor computer we usually do not have to be concerned about sending messages across a communication channel to execute a procedure. In single-processor machines, procedures are located in the same address space and the compiler can take care of generating the right code needed to execute them. But, if we are coding the same program for a distributed system, then the procedures can be located in different machines and programmers may have to be aware of sending and receiving messages across machines to get procedures executed. Thus, from the point of view of a programmer used to writing programs for a single-processor system, the communication mechanism is not **transparent**. It is not "allowing light to pass through", making blurred the once "clearly seen" procedure call abstraction. In yet other words, the communication abstraction is *interfering* with the procedure call abstraction. What the programmer wants is *transparent access* to procedures regardless of where they are located. In the context of distributed systems *transparency* means being able to provide mechanisms to the programmer in a form that minimizes the differences between programming centralized and distributed systems. We would like a distributed system to be capable of deceiving its users into thinking that the collection of machines *is* a centralized system. Transparency is probably the single most important issue to be considered when designing distributed systems. Besides access transparency there are several other mechanisms whose transparency has to be considered [138, pages 385-387]:

- *Naming transparency* mechanisms conceal from the user the location of hardware and software resources such as CPUs, printers, files, and data bases. Users should be able to identify resources by location-independent names.
- *Concurrency transparency* mechanisms ensure that concurrent accesses to resources do not interfere with each other.

- *Replication transparency* mechanisms conceal from the user the number and placement of copies of resources or services. Users of a replicated service should normally not be aware that multiple copies of a service exist, to them the replicated services can be identified individually when they request operations to be carried out. Although requests for operations may be carried out concurrently by all copies of the replicated service, replication transparency guarantees that the users of the service only receive back one set of results.

- *Failure transparency* mechanisms take advantage of the replication of resources, i.e., redundancy, to try to mask failures and trigger recovery actions wherever possible.

- *Security transparency* mechanisms conceal from the user the details of how resources are protected.

The list of transparency mechanisms can probably be extended indefinitely because for each new feature added to a distributed system there will most certainly be a correspondent transparency mechanism trying to conceal from the user how the feature is implemented. For example, the advent of portable computers has created the possibility for mobile computing. Disconnected operation has become reality. How can we make the reunion of disconnected systems simple to programmers? A possible answer is to make the reunion process transparent to programmers and users of the computing system. This answer shows how transparency requirement lists may have to grow to accommodate new system features.

1.3 Fault Tolerance

In day-to-day conversation there is a tendency to treat words like "error" and "failure" and "fault" almost as synonyms, with their actual meaning being inferred from the context where they are being used. Unfortunately, this can be a cause of confusion when these words are used to characterize the operation of systems. Thus, a more precise meaning has to be assigned to each of these and other words when they are used in technical discussions. The definitions found next follow the terminology introduced by Anderson and Lee [8, pages 6-13], Melliar-Smith and Randell [105, pages 143-153].

A *fault* is a cause of an error. Various kinds of imperfections in the hardware (e.g., electric, electronic or mechanic) or software (e.g., algorithmic) of a computing system are faults. Also, imperfections in the environment where the

computing system is placed (e.g., human errors[1]) are faults. An *error* is a part of the state of a system which is "incorrect" or erroneous. An erroneous state is an internal state of a system such that further normal processing may lead to a failure. A *failure* is an event that happens when a system does not perform according to its algorithm or specification.

There are two distinct approaches to the provision of reliable systems:

- *Fault prevention* tries to ensure that the system does not or will not contain any faults. Fault avoidance techniques are used to avoid introducing faults into the system, e.g., through the use of improved methods of design and proof. Fault removal techniques, e.g., testing and validation, are used to improve the overall reliability of the system by removing faults present in the system. Unfortunately, it is difficult (and for complex systems, practically impossible) to remove all faults, so fault tolerance techniques are called upon to deal with any residual faults.

- *Fault tolerance* techniques try to prevent faults from causing failures. When faults manifest themselves then erroneous states appear in the system so the first step towards fault tolerance is *error detection.* The second step is to try to evaluate the extent of the damage to the state of the system. The complexity of the *damage assessment* is function of design decisions made regarding the containment of damage. The two last steps in the fault tolerance process are *error recovery* and *fault treatment.* Error recovery techniques are concerned with promoting the system to a error-free state from which normal operation can be resumed. *Fault treatment* techniques aim at making sure that the manifested fault does not recur immediately. Locating the fault and repairing it is the best solution. However, when this is not possible then steps are taken to reconfigure the system to avoid the fault.

1.4 Object Orientation

The concept of object was introduced by the designers of Simula [45] in the late sixties. Today, almost twenty five years after its introduction, object orientation has become an important computing paradigm. Despite its relevance and wide application, object orientation is still settling down and, thus, invites a discretionary approach in its application. Different areas of computing science can interpret differently an object-oriented concept. Thus, this Section delineates object orientation in the context of this work.

[1] Error really means fault in this case.

An *object* is an entity that encapsulates a *data structure* and has a behaviour which is determined by its operations. At a higher level of abstraction, objects are grouped into *classes*. If objects exhibit common behaviour, then they are grouped into the same class. Thus, each class definition specifies the attributes and operations, i.e., the programming interface, that all objects belonging to that class will display. Each instance of a class—an object—has some variables, its instance variables, which are determined by the specification of its class. The operations of an object have access to these instance variables and can thus modify the internal state of the object.

Among the attributes of a class there are attributes that express relationships. An important type of relationship between classes is the *generalization/specialization relationship*[2]. Using the generalization/specialization relationship designers can create hierarchies of classes. Classes higher in the hierarchy specify operations that are inherited by all classes below and are thus common to all subclasses of the referent class. The inheritance mechanism is fundamental for the creation of class hierarchies; several object-oriented programming languages implement it. In order to create superclasses the designer finds classes that have common operations and tries to see if such common operations can be effectively grouped in a superclass. For example, it is possible to define a class, say **plant**, that represents a higher-level abstraction of a number of subclasses such as **epiphyte**[3] or **aquatic-plant**. Each of these two classes can be specialized further into subclasses such as **lichen** and **orchid** or **seawater-plant** and **riverwater-plant**. A detailed study of other important object-oriented concepts is made in Chapter 3 and 4 where object orientation is viewed in the context of Stabilis and Vigil. The generalization/specialization relationship allows Arjuna, Stabilis and Vigil a certain flexibility to programmers which can augment and specialize interfaces of these systems according to their needs. In Arjuna, persistent objects can be created. In Stabilis, new database classes can be created and, in Vigil, sensors and actuators can be refined.

1.5 Computational Reflection

Reflection involves the capacity to process information *about* a certain domain of events. A simple example is given by tools for performance tuning found in many operating systems. An operating system is a program whose problem domain is the direct management of computational resources. A performance tuning tool is a program whose problem domain is the management of the operating system. These tuning tools keep and process statistical *information about* the operating

[2]The generalization/specialization relationship is also called the "is-a" relationship.

[3]Epiphyte is a plant that lives on another plant but is not a parasite.

system to allow system managers to *reflect* upon the operating system's behaviour and *react* aiming the optimization of its behaviour. The whole performance tuning process involves three agents: an operating system, a tuning tool, and a system manager. The system manager follows policies which are enforced through the mechanisms of the tuning tool. Suppose for a while that the tuning tool and system manager have been merged into one atomic system. The resulting system has a two layer architecture where the tuning system plays a reflective role in relation to the operating system. The behaviour of the operating system is constantly observed, traced through, and changes in its behaviour trigger reactions by the tuning system. A circle of causality has been established: changes in the behaviour of the tuning system imply changes in the behaviour of the operating system, and vice-versa. Let us now look at the resulting system, tuning system plus operating system, as an atomic system; what we have is a system that has the attribute of *reflection.* The tuning system monitors the operation of the operating system using sensors and acts upon this operation using actuators. Thus, reflective computation does not directly contribute to solving problems in the external domain of the system. Instead, it contributes to the internal organization of the system or to its interface to the external world. Its purpose is to guarantee the effective and smooth functioning of the system it is reflecting upon.

We can now define computational reflection as the behaviour exhibited by a reflective system, where a reflective system is a computational system which has information about itself in a causally connected way [97].

A reflective system is a system which incorporates structures *representing* (aspects of) itself. We call the sum of these structures the self-representation of the system. This self-representation makes it possible for the system to answer questions, that is, resolve queries, about itself and support actions on itself. Because the self-representation is casually connected to the aspects of the system it represents, we can say that:

- The system always has an accurate representation of itself.
- The status and computation of the system are always in compliance with this representation. This means that a reflective system can actually bring modification to itself by virtue of its own computation.

An example of reflective computation comes from configurable distributed systems. Configuration managers compute which configuration to pursue next by reflecting upon the current state of the system.

1.5.1 What is a reflective architecture?

Capacity of information gathering and interpretation are basic requirements for reflective systems. Thus, we can view a reflective system as an interpreter with the following characteristics:

- The interpreter has to give access to data representing aspects of the program itself to any program running under the control of the interpreter. Programs implemented in the environment provided by the interpreter then have the possibility to perform reflective computation by including code that prescribes how these data may be managed.

- The interpreter also has to guarantee that the casual connection between these data and aspects of the program they represent is fulfilled. Consequently, the modifications these programs make to their self-representation are reflected in their own status and computation.

It is reasonable, in principle, to design a reflective system as a two-layered architecture because there are two main components in the system: an interpreter and an interpreted program. The interpreter is assigned to the reflective layer and the interpreted program to the application layer. The task of the interpreted program is to solve problems about an external domain, while the task of the interpreter is to solve problems and queries about the program's computation.

Maes [97] has explained how reflective facilities can be incorporated into object-oriented languages. Alternatively, the Apertos [146, 144] reflective operating system proposed the use of a multi-layered architecture containing meta-objects for the implementation of an operating system well-adapted to an open and adaptive computing environment. Choices [96] adopted reflective concepts to simplify and enhance the implementation of many of the operating system functions. This work uses reflective principles in a selective manner, specifically to simplify the architecture and implementation of Stabilis and Vigil.

1.6 Engineering a Reliable Distributed System

The engineering of reliable distributed systems is subject to many requirements and there are many alternative options a software engineer can follow to possibly fullfil these requirements. This Section reviews some of the basic mechanisms and tools software engineers have at their disposal during the design and implementation of distributed systems.

1.6.1 Communication

In distributed systems the absence of shared memory means that most of the communication between objects is carried out using messages sent across a network.

Client-Server Model

In the *client-server* communication model the distributed system is organized as a group of co-operating objects. Some of the objects play the role of service providers, or *servers*, while others have the role of customers, or *clients*. The communication pattern is very simple, a client sends a message to a server asking for a service to be done, the server executes the service and sends the results back. The advantage of this model is that it allows an overall simplification of the underlying communication protocols. An important issue is whether the communication primitives used to implement the client-server model, e.g., **send_message** and **receive_message** are synchronous or asynchronous.

The client-server model simplifies communication protocols between objects but it is not good as a programming abstraction. Programmers usually work with programming languages were the main abstraction for communication between objects is the procedure call. The problem designers are faced with is how to combine these two abstractions in order to get the best out of both.

Remote Procedure Call

An interesting and simple solution to this problem was proposed by Birrel and Nelson in 1984 [27], the mechanism proposed is called *remote procedure call* (RPC). The idea behind the mechanism is simple: use the procedure call mechanism regardless of whether the procedure called is local or remote in relation to the caller. When an object on node A calls a procedure on node B, the calling process is suspended, and the execution of the called procedure takes place on node B. Information is transported from the caller to the callee as procedure parameters, and can come back as a procedure result. In this scheme, no message passing is visible to the programmer.

In programming environments for distributed computing, remote procedure calls usually are automatically generated by a *stub generator*. A stub generator translates definitions of client-server *interfaces* into the appropriate code needed for the activation of procedures using remote procedure calls. An *interface* contains a list of method signatures, that is, their names and the types of their parameters.

1.6.2 Threads

Recently, we have seen the development of the concept of *lightweight processes* or *threads* [138, pages 507-519]. Threads are processes within processes, they are the result of recursively applying the concept of process to the process itself. Because of that threads share many of the characteristics of a traditional process. Threads have their own program counter and stack, run strictly sequentially, can spawn child threads, and share processor cycles exactly as processes do. Thus, the analogy, thread is to process as process is to machine, holds in many ways [138, page 508]. The critical difference is that threads cost little to create and destroy, can communicate efficiently (shared memory) and can be used to explore concurrency within a process, all because they run in the same address space as the process that starts them. The importance of threads to distributed system designers is that they can use the same process abstraction to structure different levels of the system. Finally, threads and RPCs are being combined to create faster and cheaper software components to implement servers. Object-oriented systems can benefit from threads by mapping objects onto these software components.

1.6.3 Atomic Actions

The notion of *atomic action*, transaction, originated in the database systems community and was first formalized by Eswaran [58], Gray [62], and Lomet [94]. Atomic actions have three important properties that help reduce the complexity of programming concurrent systems:

- *Serializability*: This property ensures that concurrent execution of programs that access some common data is free from interference, i.e., a concurrent execution can be shown to be equivalent to some serial order of execution.

- *Failure Atomicity*: the second property ensures that a computation can either be terminated normally (*committed*), producing the intended results or it can be *aborted* producing no results. This abortion property may be obtained by the appropriate use of backward error recovery, which is invoked whenever a failure that cannot be masked occurs.

- *Permanence of Effect*: the third property ensures that any state changes produced, i.e., data modification produced during the action, are recorded on *stable storage*, a type of storage that can survive system crashes with high chance.

An atomic action, once started, either *commits* and produces the desired effect or *aborts*, having no effect at all. All data touched by an aborted transaction is

guaranteed to be in the state it was exactly before the beginning of the transaction; regardless of concurrency.

The transaction model presented above is suitable for conventional database applications but does not provide much flexibility for structuring complex programs. Extensions to the standard transaction model have been proposed to overcome the limitations of the *flat* transaction model.

Nested Action

Nested actions were introduced by Moss [111] to increase flexibility in the structuring of distributed applications. Transactions start subtransactions which are themselves implemented by transactional operations. The transaction nesting structure forms a forest, that is, a graph with a number of nodes (representing the top-level transactions) as roots, each with several children (representing subtransactions) and with the subtransactions themselves having children [95, p. 15]. At the leaf level of the tree are the transactions that actually affect change to the objects of the application.

Nested transactions are useful mainly for two reasons:

- They allow controlled concurrency within a transaction: one transaction can run many subtransactions in parallel, with the guarantee that they will appear to run sequentially, and each will appear to happen either completely or not at all.
- They provide means for more flexible protection against failures, by permitting nested recovery within a transaction. When a subtransaction aborts, its parents can still continue and may be able to complete its task by initiating an alternative subtransaction.

Nested action is only one of many generalizations of the original transaction model. All generalizations aim at the relaxation of one or more of the properties of the original transaction model.

The transaction model became so widely accepted that distributed system designers decided to make it an explicit tool which programmers could use to structure distributed programs. Atomic actions are a basic element of this thesis. Atomicity is explored by Stabilis to organize its object management functions and indexing. Vigil relies on atomic actions to guarantee the consistency of transitions between external states of a system.

1.6.4 Replication

Replication is used mainly to increase availability and fault-tolerance in distributed systems. If many copies of a computing resource, say an object, are

available, then it is possible for different programs to change them in parallel, causing inconsistencies. Replica consistency protocols are therefore necessary to ensure that copies remain in a mutually consistent state.

There are two ways to replicate computing resources, using passive or active replication. In the passive, or primary-backup, approach to replication, update requests are gradually propagated between replicas. In the active approach, update requests are sent to all replicas in the same order using broadcast communication.

In the passive replication scheme a replica group of server objects is created. One server object is designated as the *primary* and the others as the *backup* servers. No more than one server can act as primary at any time. Each client knows one server and requests services from it by sending messages to it. If a client request arrives at a server that is not the current primary, then that request is ignored by the server. During normal operation the primary processes service requests and ensures that all replicas are mutually consistent by sending snapshots of its state to the backup servers. If the primary fails, then the backups elect a new primary, make it known to the clients and resume processing. Timeouts on the duration of the service requests are used by clients to discover that a primary has failed. Probably the earliest protocol for passive replication to appear in the literature is due to Alsberg and Day [7].

In active replication the behaviour of the members of the replica group changes. Now, all members of the group are active processing requests sent by clients. When one member receives a request it is broadcasted to all the other members of the group for processing. Provided all members of the replica group receive and process all requests in the same relative order as they have been made then mutual consistency is guaranteed [121].

1.6.5 Persistence

A typical computer system can have four or five types of storage components ranging from fast volatile caches to slower, persistent disks. This lack of uniformity is visible in programming languages, the language constructs for manipulating volatile data stored in main memory are different from the facilities used for manipulating data stored in persistent storage. The use of special input/output constructs is necessary if the transient data manipulated by a program is to survive the program's activation. The need to map transient data, which could be quite complex data structures, into persistent data (databases or files) increases the complexity of programming. Therefore, the elimination of this dichotomy between volatile and persistent modes of data has long been of concern to programming language implementors.

Atkinson [9, 11] defines persistence as "the period of time for which the data exists and is usable." Research into *persistent programming languages* such as

PS-Algol [10] and Napier [110] has extended traditional programming languages to present persistence as orthogonal to the programming language constructs that determine the way data is maintained. Orthogonality of persistence to data types and input/output allows programmers to use complex data structures without having to be concerned with mapping their in-memory representation into a database representation explicitly, and vice-versa.

More recently, object-oriented persistent programming languages have been designed and implemented, as exemplified by languages like Galileo [5], Trellis/Owl [113] and E [117, 118]. Some systems such as Arjuna [52, 53] and NIH [61] favor the adoption of already existent object-oriented languages and the use of class libraries to add support for persistence.

Object Stores

Research on persistence has provided us with results that strongly encourage a uniform treatment of transient and persistent objects. Some modern distributed systems combine the concepts of persistence and object to create a new abstraction for information storage: object stores. Object stores are useful for the deployment of persistence transparency because their operations can be seemingly integrated into other components of the distributed system such as the object manager and transaction manager. Object stores also play a key role in the unification of distributed databases and distributed programming environments. Later, in the next Chapter, we review examples of systems where the concept of object store is used extensively. These systems include distributed operating systems, database systems and distributed programming systems.

1.6.6 Programming Interface

Research in programming languages for distributed systems has already witnessed the development of more then one hundred different programming languages [13]. Despite all their differences, all these languages have basically to deal with three important issues:

- The use of multiple processors to allow concurrent execution of objects;
- The communication and synchronization among multiple objects;
- The potential for failure.

These three issues sum up most of the issues addressed in this Chapter. Thus, a good programming interface should provide flexible, extensible and selective use of transparency mechanisms for concealing from the programmer the details of how the underlying distributed system solves the problems related to these

three major issues. From now on we concentrate on object-oriented languages for distributed programming not because languages based on the other paradigms are less suited but because object orientation is one of the primary concerns of this work.

Object-oriented languages have a great potential as programming languages for distributed systems. Classes and inheritance mechanisms can be used to develop a very flexible and extensible interface.

Object-oriented programming languages are been designed and implemented to reflect the different views of the object-oriented paradigm, e.g., Simula [45], Smalltalk [60], Guide [78], Eiffel [106, 107], Arche [20], and C++ [137]. These languages implement different interpretations of concepts such as encapsulation, inheritance delegation, object activation, and others. Arche, for example, supports a type hierarchy and a class hierarchy, reflecting the designer's view that types are not equivalent to classes. In Arche there are two hierarchies: a type hierarchy and a class hierarchy; C++ assumes that types and classes are equivalent.

Once again, the difficulty in the design of a good object-oriented language lies in the careful selection of a subset of object-oriented concepts which are finally incorporated in the language. In the end, the distributed programming environment as perceived by application programmers is the result of a combination of features of the programming language, of the underlying operating system, and the hardware used.

1.6.7 Management

Support for configuration management varies considerably in distributed operating systems, from systems with little support, such as Amoeba [112], to systems designed to facilitate the implementation of management mechanisms, such as Apertos [144].

Management involves the extraction of dynamic information concerning a computational process and the enforcement of changes upon it to make it conform to a desired, pre-determined control policy. A *computational process* is simply anything that can be said to compute. In the context of this work, a computational process is the activity resulting from the execution of object-oriented action-based distributed applications. Control policies specify how the objects that take part in the computational process change their relative configuration over time. Below, we discuss a criterion that is used as a guideline for the design of management systems.

Organization and Control

Management has two complementary aspects: organization and control. Organization refers to the relative arrangement, or structure, of the various objects that compose distributed applications. For a given application this information is static, it does not change after the application has been developed. In the proposed management system, Stabilis is the subsystem responsible for the implementation of the static aspects of a distributed application.

The control aspect raises the problem of processing dynamic information; information that changes in time. Monitoring changes in the application's state and timely reacting requires the provision of control mechanisms, such as sensors and actuators, to instrument the application and allow the management system access to the dynamic information. Vigil is the subsystem of the management system associated to the dynamic aspects of distributed applications.

Usually, to represent management information, management systems make use of configuration and control languages. The configuration language is used to represent the structural aspects of a distributed program. The control language is used to represent the dynamic, managemental, aspects of the distributed program. A good benchmark for management systems is the relative complexity of their configuration and control languages. In this work, the confluence of object-oriented action-based programming systems and object-oriented database systems is explored to design and implement a management system with simple configuration and control languages.

1.7 Conclusions

This Chapter has surveyed some of the problems, models and techniques normally employed by software engineers in the construction of distributed systems, with the intent of setting up the framework of concepts within which this work has been developed.

We have used Lamport's definition of a distributed system because programming large distributed systems and distributed applications is still very hard and prone to errors, despite all the advances made in in this field. We are still striving to find designs that combine most of the concepts put forward in this Chapter into a dependable, friendly and easily maintainable distributed system.

We have seen that the notions of transparency and fault tolerance are critical to the development of reliable distributed systems. These notions have to be at the top of the list of any project in distributed programming and have to be dealt with from the very beginning of the design.

We have introduced fundamental concepts on object orientation, such as, class, object and relationships. These concepts play an important role in the design and implementation of the management system we have developed.

A reflective system is a system that incorporates structures that represent aspects of itself. This concept has played an important role in the design and implementation of Stabilis and Vigil. Both systems maintain information about themselves and about the distributed programs they manage.

Finally, the Section entitled "Engineering a Reliable Distributed System" has explored issues related to the client-server model, atomic actions, replication, persistence, management, and programming interfaces, from a software engineering point of view. We have also seen that management of distributed programs deals with to complementary aspects: organization and control. We are going to see that these two notions have played an important role in the design and implementation of Stabilis and Vigil.

Chapter 2

Related Work

Chapter 1 introduced principles used in the design of distributed systems such as: transparency, object orientation, communication paradigms, atomic actions, persistence and replication. This Chapter reviews work which has influenced directly or indirectly the design and implementation of Stabilis and Vigil. It looks at particular features of distributed operating systems, database systems and programming systems in order to make our design considerations more concrete and point out how they have influenced the design of the management system proposed here. Our review of distributed operating systems aims at evaluating how they have approached the problem of management and what of their built-in characteristics can be used to the advantage of management systems. In the second part of the Chapter, we see that research in database management systems and operating systems have increased their areas of intersection due to the adoption of object-oriented technologies. Finally, we evaluate some distributed programming systems and management systems to see how they have used object-oriented techniques. During our "visit" to each of these systems our concern is always with the factorization of techniques that can be used in the design and implementation of an object-oriented management system.

2.1 Distributed Operating Systems

2.1.1 Eden and Emerald

Eden [6, 30, 85] is a distributed object-oriented operating system developed at University of Washington, USA from 1980 to 1986. Eden was one of the first systems to consider tight integration of operating system and programming language important to simplify the implementation of distributed applications. The Eden Programming Language (EPL) was designed to provide access to facilities of the Eden system in an integrated and easy-to-use fashion. Programming experience with the Eden system gave positive feedback towards the implementation of object-oriented operating systems and action-based programming environments based on synchronous communication primitives, that is, remote procedure calls [29].

The project introduced the concept of a concrete Edentype. A *concrete Edentype* is a description of the state machine that defines the behaviour of an object, i.e., those patterns of invocation that it accepts and the effect of each invocation upon the object. In practical terms, a concrete Edentype is data about a user defined type, metadata, which is managed by the runtime system of EPL. This metadata is used to synchronize concurrent operations of user-defined types during runtime. EPL is syntactically similar to Pascal, with extensions for concurrent programming. EPL supports both synchronous, remote procedure call, and asynchronous communication primitives.

Objects in Eden are active and mobile; they can atomically checkpoint their state to stable storage; Eden implements a simple object store called a Eden Permanent Object Database. Objects in Eden are named using unique object identifiers and made secure through the use of capabilities. Capabilities are visible at user programming level. Eden has no support for construction of reliable applications.

Emerald [28, 74, 116] is an object-based programming language and system for programming distributed applications. Emerald can be seen as evolving and complementing concepts first introduced in Eden/EPL. Similar to EPL, Emerald offers a programming interface with support for active objects, with focus on object mobility. Objects can move freely within the Emerald system to take advantage of distribution and dynamically changing environments. Emerald uses monitors for concurrency control, does not support inheritance, and does not address fault transparency. Emerald provides some mechanisms for management of distributed programs. For example, the mechanisms for object migration can be used to implement basic reconfiguration services; objects can be moved following a failure or a recovery or prior to scheduled downtime [74].

2.1.2 Amoeba

Amoeba [112] is a distributed operating system developed at the Vrije University, Amsterdam; the project started in 1984. In terms of software architecture, Amoeba is based on the client-server model, with objects communicating via remote procedure calls or broadcast-based asynchronous protocols [75].

Amoeba is implemented using microkernels, meaning that most of the system resources are managed by programs run as user-level programs. Objects have ports whose identifiers are globally known. To execute an operation on a server object the client object first finds a server using the server's port identification and then sends a service request to it. Global port and object identifiers are managed transparently, including location, access and migration transparency. Capabilities are used to name objects and control access to their ports. At application level, Amoeba does not provide any direct support for reliability and availability. Although Amoeba support objects, it does not use object-oriented concepts either in its implementation or in its programming interface. Further, Amoeba does not provide any direct support for management or reconfiguration of applications at its programming interface.

Amoeba's programming interface resembles that of a Unix system. Usually applications are programmed in C with support of stub generators. The project has also developed an object-based language for parallel programming called Orca [14]. Orca uses shared data-objects for communication and synchronization between objects. In the runtime system created for Orca, several parallel

programs share objects and can perform the same set of operations on them, as defined by the object's type. Changes to the object made by one program are visible to other programs, so a shared object acts as a communication channel between programs. In Orca, each operation is applied indivisibly to each shared object. Operations are indivisible, but not atomic because Orca's protocols for operation invocation do not provide recoverability. Indivisibility guarantees object consistency in the presence of concurrent accesses, but only if failures do not occur. In Orca [12], the primitives for synchronization between objects are integrated with methods and allow them to block. Blocking is achieved using guarded commands. Each method of an object has a list of guarded commands associated to it. When a method is invoked it initially blocks and waits until at least one of the guards becomes true. Next, one true guard is selected nondeterministically, and its sequence of statements is executed. Vigil uses guarded atomic commands to guarantee that management actions are executed only when the application being managed is at the state specified by the management program.

Let us compare the computation model used by Orca to that of persistent objects structured using atomic actions. Objects in Orca can be directly mapped into instances of classes. Indivisible operations can be mapped to atomic actions. We can say that the models are similar in their fundamental aspects, but when we compare them in greater detail we verify that there is a difference regarding persistence of objects. The abstraction of shared objects provided by Orca [14] differs from the abstraction used by Arjuna, Stabilis and Vigil: object sharing in Orca is limited to volatile objects that are related to each other only during runtime. Such restriction does not arise in Arjuna as a result of persistence.

2.1.3 Chorus/COOL

Chorus [119] is a distributed operating system based on the concepts of actors, messages, and ports. The Chorus project run from 1979 to 1986 as a research project on distributed systems at the Institut National the Recherche en Informatique et Automatique (INRIA), France. From 1986, a company called Chorus Systèmes became responsible for the system.

Chorus has an architecture similar to Amoeba's architecture, based on microkernels and message passing. It aims at supporting emulation of Unix, something Amoeba does not do, so Amoeba does not provide binary compatibility with Unix. In Chorus the equivalent of an Amoeba object (process) is an actor; actors are multi-threaded processes. Resources are identified by ports and port rights, the equivalent of capabilities in Amoeba. Port identifiers and capabilities can be constructed and managed at user-level. The similarity is first broken when actor and object management is considered. Actors can be dynamically loaded into the kernel address but Amoeba's objects cannot; all Amoeba's objects are

user-level processes. A Chorus resource can be migrated dynamically from one server to another using port migration primitives, but resource migration is not transparent.

COOL (Chorus Object-Oriented Layer) [65, 86] is a software layer designed to provide a set of generic services to reduce the gap between operating system level abstractions and programming language abstractions. COOL provides the following abstractions to programming languages: object, virtual object memory, and clusters of objects. The virtual object memory supports creation, dynamic link/load, transparent object invocation, including location on persistent storage and mapping into clusters. Classes are structured in modules that are application-defined clusters of associated objects. When an instance of a class is created in a cluster, the class descriptor is saved in the cluster. This class descriptor is used to retrieve the appropriate module and therefore the appropriate class when a cluster of objects is remapped in another address space. Language specific constructions are mapped into the COOL layer using preprocessors. For example, COOL++ is a preprocessor that supports a extended version of C++ adapted to run upon the COOL layer. COOL does not provide any facility for management of distributed programs, but it has features that could be used to implement an object-oriented management system.

Chorus/COOL is a good example of a hybrid architecture. The operating system kernel is based on the message-process computation model and the operating system interface, provided by COOL, is object-oriented in the sense that it supports classes and objects.

2.1.4 SOS

SOS (SOMIW Operating System) [122] is a project of INRIA's Distributed Object-Oriented Systems research group. Secure Open Multimedia Integrated Workstation (SOMIW) is an Esprit umbrella project under which SOS has been developed. SOS is a distributed object-oriented operating system supporting distribution transparency including location, access and migration transparency. SOS does not provide transactions but has an object store which provides persistence. The main goal of SOS is to build an object management support layer common to all applications and languages. SOS is programmed in C++ and has been prototyped upon Unix.

SOS is based on the following concepts: elementary objects, fragmented objects and services. An elementary object is a state and a set of methods, a fragmented object is a group of elementary objects which can be located in several address spaces (contexts), on different sites. A fragmented object is an aggregate whose component objects can be distributed. Component objects of a fragmented object can break encapsulation and access each other states directly using basic

communication protocols. Elementary objects can be of three kinds: servers, proxies and providers. Services are implemented using these three types of elementary objects. A server is an object which is able to serve service requests. A proxy is an elementary object located at the same address space as the client which represents the service. Each client which wants to access a service must have a proxy of this service in its context. A provider is in charge of creating proxies on client request. For clients, proxies are the only interface to the service. A proxy can process requests locally, or forward them to a remote server. Clients have a reference to a reference, the proxy. Thus proxies are the equivalent of an extra level of indirection in dynamic data structures, a well-known programming principle has been reused to add extra flexibility to the traditional client-server model of computation. Management systems can profit from the migration flexibility offered by the use of proxy mechanisms. Stabilis uses mechanism similar to the proxy mechanism to organize its object manager and indexing structures. Key attributes, attributes that can be used to search a database, point to proxy objects instead of pointing to the objects themselves.

In summary, the implementation of SOS is based on four subsystems:

- communication: where remote procedure calls and multicast communication protocols are implemented.
- storage: the system's object store provides services for manipulating persistent objects. The representation of fragmented objects is handled by the storage manager;
- naming and binding: the subsystem responsible for name resolution and binding. References to proxies are managed by this subsystem.
- acquaintance: this is the distributed object manager of SOS. It deals with localization, migration of objects, in cooperation with the other three subsystems.

These four pre-defined services are very similar to those found in some object-oriented distributed database systems. For example, O_2 [49] also has an object manager, a communication subsystem based on remote procedure call, and an object store. This convergence in architecture and functionality is partially due to the use of object-oriented technologies.

2.1.5 Clouds/Aeolus

The goal of the Clouds [46, 87] project from the Georgia Institute of Technology, USA, is the implementation of a fault-tolerant distributed operating system based on the notions of objects and actions. Clouds started in 1985. The programming

interface is represented by Aeolus, a programming language that provides access to the synchronization and recovery mechanisms of Clouds. Aeolus and Clouds are object-based, that is, the concept of object as a unit of encapsulation and recovery is explored but the concept of class hierarchy is not. In Clouds objects are passive entities, in contrast with Eden/Emerald where objects are active. Clouds differs from Amoeba and Chorus in its adoption of an object-oriented activation style and in its use of threads. Clouds is the first operating system in this review that conceives objects as persistent and passive. The concept of threads is transparent, they are simply the activity responsible for the execution of methods. Clouds takes an architectural approach that is symmetric to that taken by Amoeba and Chorus—persistent objects and atomic actions are the main programming abstractions instead of active objects (processes), messages and persistence through state checkpointing.

While checkpointing, as implemented in Eden and Amoeba can be used to construct processes or objects with an acceptable degree of fault tolerance, it has an essential weakness in that it is oriented toward dealing with a single object or a single process at a time. Maintaining data integrity in the presence of interactions between objects is an essential problem programmers of distributed applications face. If checkpointing techniques are used, then programmers have to be aware of the detailed patterns of object interactions. Conversely, distributed transactions make the checkpointing of object states transparent to programmers. Therefore, projects such as Argus, Camelot, Arjuna, and Clouds have proposed that reliability in a distributed system be based on atomic actions, an extension of the transaction concept used in distributed database systems.

In terms of programming interface, it is interesting to note the evolutionary path followed by Clouds. In 1985, the project used Aeolus as the main programming language [87]. The emphasis then was on the development of a new programming language for distributed programming. Later, in 1991, the project had attracted new members, namely from Arizona State University and Siemens-Nixdorf and the approach shifted to the use of already existent programming languages [46]. Consequently, Clouds developed three programming environments based on extensions of C++, Eiffel, and Clide; the extensions are called DC++, Distributed Eiffel, and CLiDE respectively.

2.1.6 Guide

Guide [15, 78] is an object-oriented distributed operating system based on the concepts of jobs, persistent objects and nested transactions. Guide was started in 1986 as a joint research project of Bull Research Centre and Laboratorie de Gènie Informatique (IMAG), University of Grenoble, France.

In Guide, a job is an execution unit, it provides an object space with multiple concurrent activities. Object invocation and communication is achieved via a system primitive called ObjectCall. This primitive is similar to a remote procedure call but in ObjectCalls dynamic binding and *callbacks* are possible.

The Guide language [78] has support for classes, single inheritance, and exception handling. Guide is strongly typed and distinguishes class hierarchies from type hierarchies. There is no support for selective persistence; all objects are persistent by construction. One of Guide's objectives is the provision of aggregate objects, called composite objects in Guide. The concept of composite objects used by Guide is related to the concept of aggregate objects used by database systems. Guide has support for flat transactions on persistent objects.

Guide has a management service [54] designed using the same management principles introduced by Conic [100]. These principles include the principle of clear separation between management, dynamic reconfiguration, and application programming through the adoption configuration and control languages. The implementation of the management service is facilitated because Guide has built-in mechanisms, such as composite objects and location transparent object stores, both facilitate object migration. Additionally, Guide has a set of system calls that allow easy creation and deletion of objects, and suspension of activity. Guide's transaction system is used in the implementation of the management mechanisms. Guide's management service is object-based, that is, the only object-oriented concept used is the concept of object. In contrast, Stabilis and Vigil make use of a richer group of object-oriented concepts

2.1.7 Choices

Choices [37, 38] is an object-oriented system written in C++; it is being developed at the University of Illinois, Urbana-Champaign, USA. The project started in 1987. It supports an object-oriented application interface based on objects, inheritance, and polymorphism. Objects in Choices can be made persistent [39]. Choices and Arjuna (Section 2.3.4) share some design principles such as the adoption of a general-purpose object-oriented language for their implementation and use of object-oriented mechanisms to provide system interfaces that can be augmented and specialized according to the needs of the application programmer. Properties of the system can be selectively inherited by classes programmed by users. Unlike Clouds and Guide, Choices does not provide mechanisms for reliable software construction.

Choices main influence on Stabilis and Vigil is its uniform application of object-oriented concepts. Objects are used to model both the hardware interface, the application interface, and all operating system concepts, including system resources, mechanisms, and policies. The team responsible for Choices has devel-

oped a methodology for the design and implementation of object-oriented operating systems that has many aspects in common with the methodology adopted in in the development of Arjuna, Stabilis and Vigil. Choices bases the design of its components on the concepts of entity-relationship diagrams and class hierarchies, i.e., a structural model, and control flow diagrams, a form of control model [37].

2.1.8 Apertos

The Apertos [144] object-oriented operating system is being developed at Sony Computer Science Laboratory, Japan. This project was initiated at Sony in 1988 with a different name: Muse [146, 145]. Apertos is being developed to meet the characteristics required of a distributed operating system that has to offer services for a open and mobile computing environment.

The framework used to structure the operating system is based on reflection; an architecture based on objects/metaobjects and object migration is defined. The metaobject layer defines the semantics of certain methods of the objects. The association between objects and metaobjects change over time, these changes define how objects behave and certain object properties. Migration is the mechanism used to change the group of metaobjects an object is associated with. In the Apertos terminology, an object metaspace is a group of metaobjects that define a set of protocols of the operating system. Objects migrate from metaspace to metaspace. For example, for an object to acquire persistence, it migrates to a metaspace that supports persistence. The metaobject layer is viewed as a virtual machine that can be adapted to the needs of the group of objects it is managing. Further, each metaspace implements part of the functionality required of the operating system. There are intersections among metaspaces.

Apertos uses sensors and actuators in the same sense as Vigil does, that is, to inspect/change the state of objects. In Apertos, it is possible to change a policy of object management such as, what to do when an object is in a host that is overloaded. If the object wants to change the load management policies used, then association, migration, to a metaspace that implements the desired load balancing policy is possible. In summary, what Apertos call object migration is, in more general terms, object management implemented through dynamic association of objects to metaobjects, using a reflective architecture. Tasks like memory management, scheduling, communication, etc, are implemented by distinct metaspaces; metaspaces are specialized managers that trap, interpret and react to information generated by objects and by themselves. In such a reflective architecture management becomes a primary concept and, as a consequence, we have a system that is highly reconfigurable. The system is programmed in MC++, an extension of C++ that takes into account the existence of metaspaces.

2.1.9 Discussion

In this Chapter, we communicate a lesson we have learnt during the realization of this survey. This lesson is that a degree of convergence has been achieved by systems that have adopted object-oriented techniques in their design and construction. As we continue our survey more evidences are found of this convergence; management systems can certainly benefit from this unification.

In operating systems like Amoeba and Chorus, objects are secondary entities which are accessed by an active entity, a process. Chorus can be seen as a hybrid design due to the use of different abstractions to implement the kernel and the Chorus object-oriented layer (COOL). But, as we have seen, operating systems are evolving towards object-oriented implementations where objects are seen as the primary entities, as is the case with Eden, Guide, SOS, Choices, and Apertos. Among these systems there are systems adopting the abstraction of distributed object stores upon which language runtime systems are implemented. Furthermore, we have seen the introduction of atomic actions, originally developed for relational database systems, as an effective way of deploying mechanisms for fault-tolerant programming. Atomic actions evolved from being hidden from the user and automatically controlled by monolithic database kernels to mechanisms whose interface is visible and controllable by application programmers. Distributed systems started to provide transactions as standard built-in services, Guide is an example of such evolution of transaction systems. Apertos and Choices consider object orientation and reflection as key concepts that should be used to obtain systems that are scalable and flexible. Apertos takes object-orientation and reflection a step further by creating an architecture where mechanisms for reconfiguration management are embedded in the operating system. We have seen that object-oriented techniques are being adopted because they encourage modularization, increase reusability and maintainability, and give application/system programmers a single unified perspective of a system. Table 2.1 summarizes the main characteristics of the systems reviewed so far.

Our discussion has shown that Apertos [144] and Guide [15] have considered management of runtime behaviour of distributed programs an important requirement. Consequently, these two systems have object stores and communication mechanisms that offer better support to implementors of object-oriented programming environments and management systems. In contrast, the other distributed systems reviewed have a smaller set of mechanisms available for runtime management of distributed programs. For example, few of them have features that allow distributed programs to select and reconfigure dynamically the mechanisms provided by the distributed system such as:

- the naming conventions adopted by these systems do not allow the specification of names based of attributes of objects. Attributive names are important

Characteristics	Amoeba	Apertos	Choices	Clouds	Chorus	Eden	Guide	SOS
object-oriented interface	√	√	√	√		√	√	√
object-oriented design		√	√					
reliability support				√			√	√
persistence transparency		√		√			√	√
distribution transparency	√	√	√	√	√	√	√	√
open architecture		√	√					
reflective architecture		√	√					
selective property support		√	√					
management service		√					√	

Table 2.1: Distributed Systems: Summary Table.

to build objects that can be used in different contexts. For example, the use of attributive names allows objects to be used in different systems whitout requiring recompilation.

- little or no runtime interface checking is supported to ensure compatibility of interconnected objects.

We proceed with our review and show that that the use of object-oriented technologies can bring database management systems closer to operating systems and programming environments.

2.2 Database Management Systems

In this Section, we provide a survey of operational object-oriented database management systems that have had some influence on the design of Stabilis and Vigil. At this time we estimate that more than thirty object-oriented database systems are under development in commercial vendors, industrial research laboratories, and universities. Sufficient details about these systems is available in public domain only for a relatively small number of these systems. Therefore, we do not attempt an exhaustive survey but concentrate on those systems that have gained more visibility and, consequently, have had greater impact on the way next-generation database systems are developed.

Shortly after its appearance in 1970, Codd's relational model [44] became, and still is, the dominant model for the design and implementation of database systems. The relational model is centred upon the concept of *data* structured as sets over which *relations* are defined. A relation on n sets of a database, say $S_1, S_2, \ldots, S_n$ (not necessarily distinct), is a set of n-tuples each of which has its first element from S_1, its second element from S_2, and so on. These concepts of set and relations over sets form a relational algebra which provide integral data management capabilities for most applications. Unfortunately, they are not adequate for applications such as computer-aided software engineering, scientific and medical applications, graphics representation, office automation, etc. This class of applications demand procedural data, encapsulation, a more complete type system and other extra capabilities. Once again object orientation seems to be a reasonable option and, in fact, several object-oriented database management systems have already been developed to attend the demands posed by these new classes of database applications. Additionally, there are other models being explored by the database community such as the extended relational and functional models [31, pages 166-173].

2.2.1 Object-Oriented Database Systems

Object-oriented database systems lie at the convergence of research in database management systems, operating systems, and programming languages [51, pages 1-10][147, pages 1-32]. There are two approaches to developing object-oriented database systems that are being actively pursued by the research community. The two approaches differ in their starting points: one approach seeks to extend existing database management system concepts with data and procedural abstractions; the other approach embellishes existing programming languages with persistence and sharing. The approach taken in the design of Stabilis and Vigil benefit mainly from ideas generated in the latter approach.

The features of object-oriented database management systems might be considered as combining the best features of traditional database systems and object-oriented programming languages. The systems described in the next sections can be grouped into two broad categories, depending on whether they evolved from programming-language or database-system architectures:

- evolving from database programming languages: O_2 from O_2 Technology [49], ObjectStore from Object Design [83], GemStone from Servio-Logic [35], ODE from Bell Labs [3], and ORION from MCC/CDC Systems [77, pages 259-282].

- evolving from extended relational database management systems: Postgres from University of California at Berkeley [136], and Starburst from IBM Almaden Research Centre [93].

There are trade-offs between database architectures in terms of compatibility with relational systems, convenience of access from host programming languages, and design of the query languages. Concerning query languages, Starburst is based on extensions to SQL. Postgres is based on extensions to INGRES QUEL. Both of these systems produce relations designed to represent logical objects that are represented internally as tuples. The query languages of ObjectStore and GemStone bear some resemblance to the programming languages on which they were originally designed—C++ and Smalltalk, respectively. Both query languages are designed to select objects from collections of objects. The extended relational query languages in Postgres and Starburst are more powerful, supporting complex joins and views. An O_2 query can result in objects, tuples, lists, sets, or any type of data value. Thus, when considering query languages, we can coarsely divide object-oriented database management systems into those that provide support for SQL-like queries and those that do not. Table 2.2 summarizes the historical language influences these database systems have had.

Database Category	Database System	Query Language Syntax Basis	Original Programming Language Basis
Database Programming Languages	ORION	Lisp-like	Common Lisp
	ObjectStore	C++-like	C++
	ODE	C++-like	C++
	GemStone	Smalltalk-like	Smalltalk
	O_2	Hybrid Algebra	O_2C
Extended Database Systems	Postgres	Quel extensions	-none-
	Starburst	SQL extensions	-none-

Table 2.2: Database systems and programming languages.

Designers of relational database management systems usually claim that the separation between host programming languages and query languages bring advantages to the programmer. For example, the capacity to debug database queries separately from application programs, and the possibility of having only one query language which can be interfaced to many programming languages, e.g., Cobol, Fortran, APL, C, etc. Designers of programming languages and persistent programming systems argue that usually the distinctions between the languages used in these two modes of programming place an unnecessary learning and memory burden on those users who have to work in both modes. Thus, in database man-

agement systems that have originated from programming languages, there are good reasons to try to integrate as much as possible the query language into the host programming language. Ideally, only one programming language should suffice all needs of the application programmer, allowing him to query databases and program complex algorithms in a uniform programming environment. Stabilis follows the latter approach.

O_2

O_2 [49, 88] was initially designed and developed within the Altaïr research consortium, France, funded by INRIA, Siemens-Nixdorf, Bull, the CNRS and the University of Paris XI. Altaïr was a five year project which began in September 1986. Its goal was to design and implement a next-generation database system. A throw-away prototype was ready in December 1987. From 1988 to 1990 the Altaïr consortium produced and tested a final version of the database system. At the end of 1990, a commercial company, O_2 Technology, was created. Commercial shipment of O_2 began in June 1991.

The kernel of the O_2 database management system, called O_2Engine, is capable of storing structured and multimedia objects. It handles disk management, distribution, transaction management, concurrency, recovery, security, and data administration.

O_2Engine can support two types of interfaces: programming language interfaces, C and C++, and the O_2 environment. Language interfaces allow a C or C++ program to benefit from the services of O_2Engine by declaring O_2 schemas and populating O_2 databases. Alternatively, the user can benefit from the complete O_2 environment. This environment includes: a query language, O_2Query; a user interface generator, O_2Look; an object-oriented fourth generation language, O_2C; a graphic programming environment with a schema and database browser.

O_2 objects are identified by unique object identifiers. Persistence is achieved by associating an object type definition to a persistent "root" type. A persistent "root" type is a type in the schema that has a name associated with it. When an object is created, the programmer is not forced to decide if it will be persistent from the beginning. Objects may be made persistent after they are instantiated by associating them with an instance of a persistent "root". The data model supports multiple inheritance. Class definitions can be shared among schemas.

O_2Query is an SQL-like query language extended to deal with objects. It is a subset of O_2C, but may be used independently as an ad hoc interactive query language or as a function accessible from C or C++.

This brief review of O_2 is sufficient to highlight the similarities and differences with object-oriented operating systems and distributed programming environments. There is an functional overlap between the various systems which

make O_2 and the systems developed for object-oriented operating systems and distributed programming systems. For example, all these systems rely on the abstraction of an object store.

ObjectStore

ObjectStore [83] is an object-oriented database management system that provides a tightly integrated language interface to database management system features of persistent storage, transaction management, distribution, and associative queries.

The programming interface of ObjectStore is an extended version of C++. Objects of any C++ data type can be allocated transiently or persistently. Persistence is not an inherited attribute. Instances of the same class may be persistent or transient within the same program.

The motivations the ObjectStore team puts forward for making ObjectStore closely integrated with the programming language are:

- Ease of learning. ObjectStore is designed so that a C++ programmer would have to learn a little more in order to use ObjectStore's facilities. In particular, there is no need to learn a new type system or a new way to define objects.
- Persistence Orthogonality. Persistence is orthogonal to the representation of objects. The designers of ObjectStore wanted to save the programmer from having to write code that translates between the disk-resident representation of date, and the memory-resident representation. The way ObjectStore handles persistence is very similar to that of PS-Algol [10].
- Conversion. Easy conversion of pure C++ programs to ObjectStore applications is achieved by guaranteeing that the syntax and semantics of C++were not changed. In particular, variables should not have their type declaration changed when persistent objects were used.
- Designed for CPU-bound applications. ObjectStore's focused on making the access time to objects very low by integrating the management of the persistent object store with the virtual memory management system of the machine.
- Transactions. As in Camelot and Arjuna, transactions are visible to the database programmer.

The data model of ObjectStore supports only the association relationship. There is support for sets and lists of objects. There are two possible ways of expressing queries in ObjectStore. For example, suppose that **all_employees** is a set of **employee** objects: `os_Set<employee*> all_employees;`

The following statement uses a query against **all_employees** to find employees earning over $100,000, and assign the result to **overpaid_employees**:

```
os_Set<employee*>& overpaid_employees =
all_employees[: salary >= 100,000 :];
```

Expressions enclosed by [: :] are queries. Stabilis takes a similar approach to introduce queries in C++ but does not extend the language to achieve that goal. Consequently, a new C++ front-end did not need to be implemented.

ObjectStore has support for a simple form of version control. Users are allowed to create versions of objects but there is not any consistency protocol to guarantee that the copies remain consistent. The implementation of consistency protocols is left to the implementor of the database application that uses versions.

The object manager of ObjectStore relies on the virtual memory management system of the machine to fetch/write objects from/to the object store.

GemStone

GemStone [35][76, pages 200-202], a commercially available object-oriented database management system developed by Servio Corporation. GemStone supports a client-server architecture in which a number of clients may connect with GemStone engines running on a central server. Thus, GemStone cannot be considered fully distributed. GemStone provides a programming interfaces to three programming languages: C, C++, and Smalltalk. The basic programming interface of GemStone is called OPAL; it extends Smalltalk with schema definition and data manipulation capabilities.

GemStone's object model is an extension of the object model defined by Smalltalk. It supports only single inheritance and attributes whose values can be sets of heterogeneous objects.

Persistence is implemented as an association to a persistent 'root' object, in an approach consistent with that of Smalltalk. The user cannot explicitly delete any object, as long as the object is referenced by any other object; a persistent object is automatically deleted only if it is no longer reachable from the 'root' object.

GemStone supports concurrency control, crash recovery, dynamic schema modification, transaction management, and queries. The GemStone approaches to queries and indexing, and concurrency control are unique among database systems.

The basic components of the GemStone architecture are the Gem server and the Stone monitor [35]. The Gem server is where object behaviour specified in GemStone's data manipulation language is executed. Query evaluation occurs

within the Gem server. The Stone monitor allocates object identifiers and coordinates transaction commit activities.

Concurrency control is optimistic and implemented via shadowing. With optimistic concurrency control, objects need not be locked. At commit, read/write conflict detection with other Gems that have committed since the last commit by the Gem server is performed. If no conflict is detected, the the updates performed by the Gem are committed to the database. Otherwise, the commit fails and the Gem has the option of aborting and beginning a new transaction. Later, in GemStone version 2.0, object locking was introduced [73]. Version 2.5 of GemStone introduced version control of objects.

The GemStone C++ interface is implemented with the use of a stub generator. In addition, a class library is provided, giving the programmer a standard set of definitions for sets, arrays, etc, as well as functions for managing GemStone objects. C++ classes are captured as GemStone classes by submitting them to a class registration utility that stores the classes definitions as part of a GemStone schema.

GemStone's data definition language/data manipulation language is a variant of Smalltalk that is interpreted by Gem servers. GemStone has modified the Smalltalk environment substantially: classes that implement transaction control, authorization, replication and index control have been added. The class hierarchy is extensible.

The tools provided by GemStone include a visual schema designer that allows the user to capture data models. The relationships provided are: generalization/specialization, aggregation, and association.

ORION

ORION [77, pages 259-282] is an object-oriented database management system developed at MCC, USA. ORION's features include: persistence, atomic actions, versions, aggregate objects, dynamic schema evolution, queries, and multimedia data management. ORION is implemented in Common Lisp [133] and is not distributed.

The object manager of ORION provides high-level functions, such as schema evolution, version control, query optimization, and multimedia information management.

The object store subsystem provides access to objects on disk. It manages the allocation and deallocation of segments of pages on disk, finds and places objects onto pages, and moves pages to and from the disk. It also manages indexes on attributes of a class to speed up the evaluation of associative queries.

The transaction subsystem provides a concurrency control and recovery mechanism to protect database integrity while allowing the interleaved execution of

multiple concurrent transactions. Concurrency control uses a locking protocol, and a logging mechanism is used for recovery from system crashes and user-initiated aborts.

ODE

ODE (Object Database and Environment) [3] is being developed at AT&T Bell Laboratories, USA, since 1987. ODE's approach to the development of object-oriented database management systems is the closest we have in this revision to the approach adopted by Stabilis and Vigil. ODE offers an integrated data model for both database and general purpose manipulation. Databases are defined, queried and manipulated in the database programming language O++ which is based on C++. O++ [4] borrows and extends the class object definition facility of C++. ODE/O++ provides facilities for creating persistent objects with version, defining sets, and iterating over sets and clusters of persistent objects. ODE also provides facilities to associate constraints and triggers with objects as in active databases [59]. ODE also has a simple graphical interface [2].

Some distributed programming systems have atomic actions available at the programming interface just as ODE. Arjuna [126] and Camelot [56] are examples of systems where atomic actions are available explicitly at the programming interface.

Starburst

Starburst [93] was initiated in 1985 at IBM Almaden Research Centre, USA. Starburst is written in C. Starburst is based on the relational model and on extensions of a "standard" database manipulation language (SQL) that allow users to exploit relational database management system technology, with provision of facilitates to port existing applications to Starburst.

A basic principle of the Starburst extensible database management system is that the conflicting goals of application-specific facilities and information integration can best be satisfied by database management systems that support the addition of domain-specific extensions in the context of a common data model. This hypothesis distinguishes Starburst from other work in extensible database management system Both the Genesis and the EXODUS projects have built automated database management system toolkits or generators for building a database management system with a customized data model, storage system, query processing algorithms, etc, that best suit a particular application domain. However, this approach may make it difficult to share and integrate data from several diverse applications.

Postgres

Postgres [136, 135] has been under construction since 1986, it is a sequel to Ingres. Postgres is implemented in C. Postgres is oriented toward access from a query language, POSTQUEL. Independently, Postgres has a navigational query interface. Postgres takes a multilingual approach to its programming interface. The team believes that tightly coupling a programming language to the database system restricts the freedom programmers have in choosing alternative programming interfaces. So, they consider that when desired, it is easier to create preprocessors to integrate languages such as C, C++, and Lisp to Postgres. The disadvantage of this approach is that it inevitably produces several extended languages whose runtime system must be well adapted to Postgres's object management layer, otherwise performance suffers.

Postgres recognizes the importance of rules to simplify the programming of non-traditional database applications. In Postgres rules are used to specify integrity constraints and to derive results that are not directly stored in the database.

Commercial relational database management systems are oriented toward efficient support for data processing applications where large numbers of instances of fixed format records must be stored and accessed. The traditional query facilities for this application area are called **data management** by Postgres's team. The team defines two other management dimensions that have to be supported: **object management** and **knowledge management**. Object management entails efficiently storing and manipulating complex data types such as bitmaps, icons, etc. Knowledge management entails the ability to store and enforce a collection of **rules** that are part of the semantics of an application. The Postgres's team believes that most applications being developed now are three dimensional, requiring data, object and knowledge management services.

2.2.2 Active Databases

The research community in database systems has long recognized the need for integrating rules and facts in a database management system context. It has been argued that the lack of a rule facility can place a significant burden on the database management system application programmer. For example, in order to support general integrity constraints in the absence of such a facility, every transaction that updates the database for a given application has to be augmented with code to check the constraints and to take an appropriate action if a constraint is violated. Thus, the condition checking activity and the action become integral parts of users' transactions. In recent years various approaches have been suggested for adding active capabilities to database systems in order to integrate

rules and facts and thus simplify the application programming task. Next, a brief revision of the major approaches to creating active databases is made.

The addition of active capabilities to database systems was first considered in order to support specific database management system functions such as view maintenance and integrity constraint enforcement. It was proposed in [57] and [58] that "triggers" be added to System R in order to enforce integrity constraints or "assertions". Triggering mechanisms of different types have also been suggested to support the maintenance of materialized views, snapshots, and derived attribute values [3, 32, 34, 66, 89, 109]. More recently, it has been proposed that generalized active data management capabilities be added to database systems in order to provide a unified mechanism to support a variety of applications, such as those requiring rule-based inference.

The Postgres project [134] proposes a general mechanism to support alerters, triggers, and rules in the context of an extended relational data model.

Research in active databases [48] has contributed ideas on how to solve control problems by developing object models and query languages that are capable of expressing configuration and control.

2.2.3 Database System Generators

Database generators or extensible database management systems are database management systems designed as set of modules which can be expanded, adjusted and integrated in various ways which allow the creation of database management systems customized to particular applications. Rapid prototyping, flexibility and extensibility are the main concerns of these systems. We analyze two extensible database management systems. Genesis [19] is an extensible database system developed at University of Texas at Austin, and Exodus [40] is an extensible database system developed at University of Wisconsin.

Genesis

The basic argument the implementors of Genesis put forward to justify their system is that traditional database management systems can be customized in one of two ways: systems are developed from scratch, or existing systems are enhanced. Batory [19] argues that both approaches are costly and not always successful. Consequently, there is a definite need for tools that simplify and aid the development of database applications. The team proposes a theory for the factorization and modularization of the various functions of database management systems so that their most basic components are revealed. These basic components are: simple files and file structures; link-sets (structures used to assemble links between database records), and elementary transformations, mappings, from conceptual

models to internal database representations. The thesis defended by them is that the storage and retrieval architectures of any commercial database management system can be explained in terms of compositions of these basic building blocks. Thus, their theory says that any database application can be built by combining a number of basic extensible modules. Genesis is a prototype system built upon this theory. Genesis can be reconfigured into a database management system that stores and retrieves data according to a specified storage architecture. Extension and customization to new database applications is accomplished by synthesizing the target database management system from a library of software modules that correspond to the basic components of the theory. The library of modules is extensible, so new modules can be added as needed.

Exodus

The goals of Exodus are very alike the goals of Genesis. Exodus [41, 40] aims at facilitating the fast development of high-performance, application-specific database systems. Exodus provides certain kernel facilities, including a flexible store manager. It also provides an architectural framework for building application-specific database systems; tools to help automate the generation of such systems are provided, including a rule-based query optimizer generator and a persistent programming language; and libraries of generic software components (e.g., access methods) that are likely to be useful for many application domains.

2.2.4 Discussion

This analysis of object-oriented database management systems reveal several interesting facts regarding the convergence of object-oriented systems. An important fact is that the core object-oriented concepts are shared by almost all systems; that is, the data models supported include notions of object identity, instances, classes, class hierarchies and inheritance, and message passing. All systems support persistence of objects, although a few different notions of persistence have been adopted with respect to explicit deletion of persistent objects. The concept of object store is prevalent, and most of the systems support a client/server architecture, although some are not truly distributed, e.g., O_2.

Many of the systems have attempted a seamless integration of a programming language and a database system by extending an object-oriented programming language with database commands. The integration of database management system and programming system usually has involved the development of an object manager to perform memory management and to negotiate the consistency of memory-resident objects and disk-resident persistent objects.

To Stabilis and Vigil, the relevance of the research on extensible databases is

Characteristics	Stabilis	GemStone	ObjectStore	O2	ORION	Starburst	POSTGRES	ODE	Genesis
object-oriented interface	√	√	√	√	√		√	√	√
tightly integrated prog. lang.	√	√	√	√	√			√	√
object-oriented design	√			√					
object identity	√	√	√	√	√	√	√	√	√
object-oriented data model	√	√	√	√	√		√	√	√
relational model support		√		√		√	√	√	
collections (sets, lists)	√	√	√	√	√	√	√	√	√
active database						√	√		
aggregates	√	√	√	√	√	√	√	√	√
object versions								√	
navigational query	√	√	√	√	√	√	√	√	√
reliability support	√	√	√	√	√	√	√	√	√
distribution	√								√
object manager	√	√	√	√	√	√	√	√	√

Table 2.3: Database Systems: Summary Table.

the concept of customization and extensibility. In Stabilis, a programmer can add new types to the system and in Vigil the interpretation of guarded actions can be customized. Extensible databases have tools for automatic generation of code, similarly, Stabilis and Vigil have been designed to be able to generate parts of a distributed program from the application's object model.

Further, most systems support concurrency control, recovery, and transaction management; some do so through the traditional mechanisms of relational database systems, while others provide these mechanisms explicitly, as primitives that are visible at the programming interface. Table 2.3 summarizes characteristics of the database systems reviewed.

2.3 Distributed Programming Systems

2.3.1 ISIS

ISIS [22, 23, 24, 25, 26] is a toolkit for distributed programming based on group communication (process groups) and group communication tools. Fault-tolerance is based on a tool for creating checkpoints and on a a set of multicast protocols that ensure conservation of causality and atomicity in the delivery of messages exchanged among process groups. The computation model underlying all the structure of the system is the virtual synchrony model [25]. Virtual synchrony is good for applications were forward recovery is essential, for example, process control applications. As the focus of this thesis is on action-based systems, it is worth to say that action-based systems centre their concerns in the isolation of concurrent transactions, persistent data and rollback mechanisms. ISIS is mainly concerned with direct co-operation between members of process groups, as such, it does not provide atomic transactions. Persistence of data is important in database systems, but much less so in ISIS. For example, the commit problem is a form of reliable multicast, but a commit implies serializability and permanence of effect of the transaction being committed, while delivery of a multicast in ISIS provides much weaker guarantees [24]. Despite the differences between non-transaction based and transaction-based systems, there is ongoing research trying to implement atomic actions using families of protocols similar to those implemented by ISIS. The Raid [21] project is an example of a distributed database system where atomic actions rely on multicast protocols and virtual synchrony for their implementation.

Meta [143] is a system for building fault-tolerant reactive control applications. It consists of a layer for instrumenting a distributed application with sensors and actuators. Meta defines a number of built-in sensors, e.g., to return the load or the set of users of a machine. User-defined sensors and actuators extend this initial set.

The "raw" sensors and actuators of the lowest layer are mapped to *abstract* sensors by an intermediate layer, which also supports a simple database-style interface and a triggering facility. This layer supports an entity-relation data model and conceals many of the details of the physical sensors, such as polling frequency and fault tolerance. Sensors can be aggregated, for example by taking the average load on the servers that manage a replicated database. The interface supports a simple trigger language, that will initiate a pre-specified action when a specified condition is detected.

Running over Meta is a distributed language for specifying control actions in high-level terms, called Lomita. Lomita code is embedded into the Unix `csh` command interpreter. At run time, Lomita control statements are expanded into

distributed finite state machines triggered by events that can be sensed local to a sensor or system component; a process group is used to implement aggregates, perform these state transitions, and to notify applications when a monitored condition arises.

2.3.2 Darwin/Regis (Conic)

Conic [79, 100] was probably the first project in the area of management of distributed systems to address dynamic configuration, it was started in the early eighties at Imperial College of Science, Technology and Medicine, England. The Conic system has four subsystems: (i) a configuration language (Darwin), (ii) a programming language (an extension of Pascal), (iii) a distributed operating system, and (iv) a configuration manager.

Conic has been designed upon the assumption that a distributed system should be programmed using two a two-level abstraction. For this purpose, two programming languages are used: one for programming the several modules of the application and other for specifying the configuration of the modules which constitute a logical computing unity (node) of the distributed system. The language for programming the modules is Pascal, extended with message passing and module primitives. Conic does not have any built-in support for persistence and atomic actions. The configuration language has primitives for describing a distributed application in terms of the relationships among their basic modules. The REX (Reconfigurable and EXtensible parallel and distributed systems) project [82, 101] is the sequel to Conic; it was started in 1989. The main changes in relation to Conic are that now the individual modules of an application can be programmed in any imperative programming language (C, Pascal, Fortran, etc) and subsequently interfaced to each other using a common Interface Specification Language (ISL). The configuration language, Darwin [98], is an evolution of Conic's original configuration language. Darwin has a rich set of constructs for the specification of binding and message passing modes among modules. Finally, there is a graphical tool, ConicDraw, to help with the specification of reconfigurable applications [81].

Regis [99] is the result of the coevolution of Conic, REX and their respective configuration languages. Regis is a programming environment aimed at supporting the development and execution of parallel and distributed programs. Darwin/Regis represent an extension of Conic/REX in two major areas: (i) separation of communication from computation, and (ii) support for dynamic program structures.

It is interesting to observe that all systems mentioned above base their design and implementation on the same principle: "a separate, explicit structural (configuration) description of distributed programs is essential for all phases of the software development process, from system specification as a configuration of

component specifications to implementation as a set of interacting computational components. [99]"

The Conic system was restricted to a single programming language, Pascal, extended with a fixed set of communication primitives for defining computational components. REX permitted the use of varied programming languages (C, Pascal, and Modula-2) for programming computational components, however, only a fixed set of communication primitives was available. Regis allows users to specify their own communication primitives. Thus, Regis separates configuration, computation, and communication while its predecessors considered computation and communication as integral [99].

Dynamic system configuration is the ability to modify and extend a system while it is running. Incremental changes are introduced to the system as they are needed. Kramer and Magee [79] show that dynamic configuration of systems should occur at the programming-in-the-large scale of software where atomic systems are treated as autonomous components which can be reconfigured as the circumstances demand. In their approach a *configuration specification* is written in a *configuration language* that describes *both* the behaviour and the structure of a system composed of a set of atomic subsystems.

Let us review the original requirements a management system should meet according to Conic [79].

Programming Language

- Modularity—The language must provide software modules which can be written and compiled independently from the configuration in which they will run, i.e., *context independence*. In object-oriented systems, classes aggregates are the equivalent of modules.

- Interconnection—The direct naming of other modules or communication entities restricts the logical configuration flexibility since change would involve modifying these names in the program text and thus require recompilation. *Interconnection independence* is the key property in separating module programming from configuration. This requirement is met by having attributive names, that is, the names used to find objects are specified in terms of properties and not in terms of fixed identifications.

- Interfacing—All the information passing into and out of a module must be an interface which specifies both the type of the information and the mechanism by which it is to be transferred. This requirement is met by encapsulation in object-oriented systems.

- Distribution—Distribution transparency ensures that intermodule communication is perceived always in an uniform way independently of the fact that modules are located in the same or in different address spaces.

- Resource requirements—It is desirable to know the maximum physical resource requirements of a module. This allows the configuration manager to check that a station can provide these requirements and that the module will not fail once it has started because, for example, it cannot acquire enough store. In an object-oriented system, objects can have this information represented as an attribute of them. Thus, when needed the management program can determine what is needed in terms of system resources.

Software systems can be conveniently described, constructed and managed in terms of their configuration, where configuration is the system structure defined as the set of constituent software components together with their interconnections [81, 82]. In the management system created in this thesis, Stabilis manages structural metainformation and Vigil manages control metainformation.

Configuration Language The configuration language is used to specify two aspects of a system: its structure and changes it can go through (control).

- Context definition—The configuration language must specify the set of module types from which the system is constructed. In an object-oriented approach, this is equivalent to classes specified in the structural model.

- Instantiation—It must also specify the instances of module types which are to be created in the system. In object-oriented systems, the instantiation relationship in the structural model guarantees the consistency of the types of the modules (objects) instantiated.

- Interconnection—The configuration language must describe the way module instances are interconnected. In our approach, structural models are used to specify configuration of instances of classes, that is, objects.

- Declarative configuration language—It is desirable that the configuration specification be descriptive (declarative). The structural and control models adopted in this work are used to build declarative specifications of distributed programs.

Operating System The distributed operating system is responsible for modifying the running system in response to commands from the configuration manager.

- Module management—The operating system must provide the ability to load/delete the code for module types into station(s). Additionally, it must allow the configuration manager to control the execution of modules and to query the state of the system.
- Connection management—The operating system must provide facilities to establish and delete connections between modules.
- Communication support—The system must support intermodule communication.
- Real-time modification—The time taken by management operations should be such that they can be used for on-line real-time reconfiguration.
- Logical interconnection—Logical interconnection between components of the system simplify the specification task.
- Flexible operating system configuration—The operating system should itself be capable of flexible configuration to enable it to be used in different systems and to allow it to be changed in the same way as the application system it supports.

  The major constraint on the operating system is that the facilities it provides must be integrated with the programming language and configuration language for an efficient and safe system.

The Validation Process

- Interconnection—Ensure type compatibility between communicating modules. Stabilis and Vigil do not violate any of the type checking features of C++, consequently, type compatibility in the management system is as good as can be guaranteed by C++.
- Specification/system consistency—It is essential to ensure that the configuration of the actual system is given by and satisfies the current specification. The fact that metaobjects are related to objects guarantees this form of consistency.
- Allocation—The logical-to-physical mapping should be checked to ensure that the resources required by modules can be provided by the underlying system. The use of metainformation on the system can help the maintenance of such mappings.
- Behaviour—It should be possible to perform some semantic checks on the behaviour of the configuration based on information provided on the behaviour of each module. Once again, the management of metainformation on a program provides the necessary infra-structure for semantic checks.

Darwin/Regis	Stabilis/Vigil
module (process)	class (object) (Stabilis)
port defintion	class interface definition (Stabilis)
provide/require	relationships (Stabilis)
bind	query resolution and relate/unrelate (Stabilis)
forall	ObjectSet and for (Stabilis)
when do	guarded action (Vigil)
loop select	guarded action scheduler (Vigil)

Table 2.4: Darwin/Regis compared to Stabilis/Vigil.

Stabilis and Vigil meet the requirements above but are not as flexible and complete as Darwin/Regis; specially because Darwin/Regis allow its user to specify varied communication protocols while our current implementation of the management system provides only remote procedure calls; the implementation of reconfigurable communication protocols is planned [114]. Table 2.4 shows a one-to-one mapping between each main feature of Conic/REX/Regis/Darwin and Stabilis/Vigil[1]. Table 2.5 brings a summary of the characteristics of the management systems reviewed so far. Darwin/Regis allow the specification of relationships between objects at the level of methods (provide/require), Stabilis/Vigil allow such specification at the level of class using relationships.

2.3.3 Argus

Argus [92, 90, 91] is an early distributed object-oriented language which was developed at MIT during the mid 1970s. Argus is based on CLU and was designed specifically for supporting fault tolerance. The main features of Argus are guardians and actions. Guardians are Argus modules which contain data objects and methods for manipulating those objects. Guardians are active entities and they control the synchronization and recovery properties of objects. Guardians encapsulate and control access to a collection of data objects to which access is possible only by invoking operations exported by the guardian.

Communication in Argus is through remote procedure calls. Parallelism is supported by allowing guardians to execute in parallel and by allowing processes to execute in parallel within guardians. Instances of guardians are created by sending messages to creator objects.

[1] We were not able to find published information that would have allowed us to make a better comparison between Stabilis/Vigil and Meta.

Characteristic	Vigil	Darwin/Regis	Meta
computational model	object and action	message and process	message and process
object-oriented	yes	object-based	no
persistence	yes	no	no
communication	RPC (fixed)	async. msgs and RPC (configurable)	ordered broadcast
programming language	C++	Pascal, Modula-2 C, C++	C
instrumentation	programmed	programmed	programmed
control language	state machines	Regis	Lomita
configuration language	object model (database schema)	Darwin	Lomita
composite objects	yes	yes	no
pre-defined sensors/actuators	yes	yes	yes
inherited sensors/actuators	yes	composition	no

Table 2.5: Vigil, Regis and Meta: a synopsis.

Argus distinguishes the problem of providing atomicity for a computation from that of supporting resilience. Actions are units of atomic activity. However the atomicity properties are provided only by *atomic objects* and atomicity is guaranteed only when all the objects shared by actions are atomic. Argus supports nested actions [111] as well as nested top-level actions. There are a number of built in atomic types as well as facilities to allow users to define new atomic types [142]. To achieve persistence, a guardian definition may specify a number of *stable variables*. Atomic objects reachable from a stable variable are *stable objects*, that is, a persistent object. Only the stable state of a guardian is recovered after a failure. Finally, Argus is a single language system with a tight coupling between system and language.

2.3.4 Arjuna

Arjuna is an object-oriented programming system which provides a set of tools for constructing fault-tolerant distributed applications [126]. 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 serializability. Locks on objects are acquired inside an atomic action, and are released only when the outermost atomic action ends (or aborts) [124]. 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.

Architecture

The main modules of the Arjuna system are shown in Figure 2.1. 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 objects. The Atomic Action module is the application-level interface. These modules are described further in the subsequent sections.

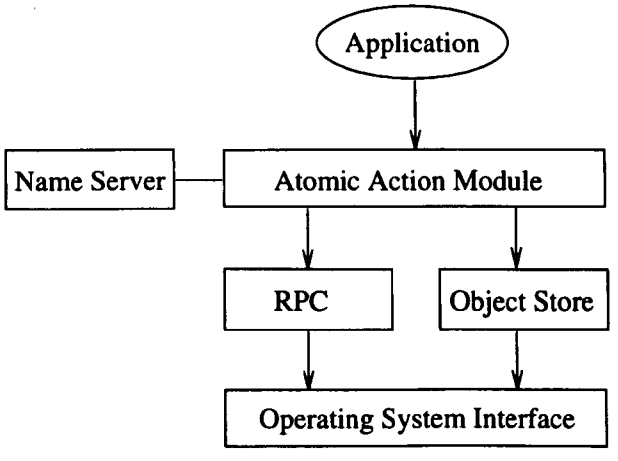

Figure 2.1: The architecture of Arjuna.

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 [115] 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.

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-supplied 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.

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 2.2 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**.

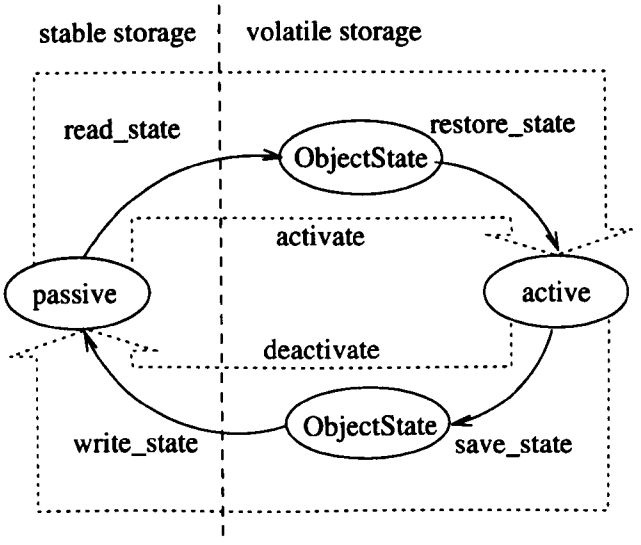

Figure 2.2: Object state transitions.

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 2.3. 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 organize 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 2.3.

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 2.3 implement most of the support operations of the Atomic Action module; further details about the class hierarchy of Arjuna can be found in [126].

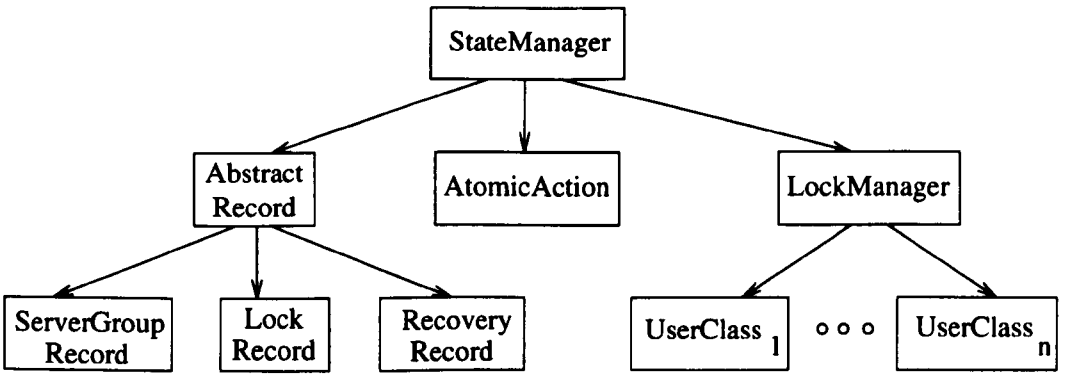

Figure 2.3: The Arjuna class hierarchy in the Atomic Action module.

Replication

Arjuna supports passive replication. Replication transparency mechanisms conceal from the user the number and placement of copies of resources or services. Users of a replicated service should normally not be aware that multiple copies of a service exist, to them the replicated services can be identified individually when they request operations to be carried out. Although requests for operations may be carried out concurrently by all copies of the replicated service, replication transparency guarantees that the users of the service only receive back one set of results. At the programming interface of replica services each replicated resource is represented by a *replica group.*

In Arjuna, groups are managed through the name server and with the help of a specialized database called the GroupView Database. Both name server and GroupView database maintain information pertaining replicated objects. To replicate an object a programmer has to create a replica group by entering the information concerning the replicas into both subsystems.

Failure Assumptions

Arjuna assumes 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.

2.3.5 Camelot and Encina

Camelot [56, 130] is a distributed transaction facility build atop of the Mach [1] distributed operating system and its programming tools; it is the successor of TABS [129]. Camelot was developed at Carnegie-Mellon University, USA. It implements the recovery, synchronization and communication mechanisms needed to support distributed transactions and objects.

In Camelot, object managers, called *data servers*, encapsulate code and data implementing different abstract data types (classes). Applications act as clients which can begin and end transactions and use remote procedure calls to request data servers to carry out methods on the data which they maintain. Data servers can be remote from an application. Accessing such a server results in a distributed transaction. A server may also act as a client and call another server. The transaction model allows nesting and parallelism inside a transaction, based on concurrent subtransactions. To the programmer of an application or data server, the Camelot services are available as libraries.

To reduce the effort required to construct reliable distributed applications a new programming language, called Avalon [71], has been created. Avalon encompasses compatible extensions to C++ and its compiler automatically generates code for calling Camelot and Mach facilities. Full linguistic support for atomic objects and transaction management is provided by Avalon.

Encina [139] is a commercial system being developed by Transarc, a company founded by some of the main designers of Camelot. Encina aims at supporting commercial distributed transaction processing programs in an open distributed computing environment. It is being developed to run on top of Open Software Foundation's Distributed Computing Environment (DCE). The Encina architecture conforms to the X/Open distributed transaction model. Encina provides a transaction monitor as well as a few specialized transactional resource managers (including record-oriented transactional file server, a specialized object store, and a transactional queueing system). Integration to relational database management systems is supported by use of the X/Open interface [141]. Interoperability with mainframe systems is provided by incorporating a multiplatform protocol interface.

At a lower level Encina offers services such as strict two-phase read/write locking, logging and two-phase commit, which can be used in building new transactional resource managers. The Encina transaction model supports nested transactions and parallelism within transactions.

To the programmer the Encina services are available as augmented C/C++ constructs mapped to library calls by a stub generator. The stub generator takes care of distribution transparency [140].

2.4 Monitoring Systems

Monitoring has provided insights into the problem of instrumentation. For instance, TQuel [127, 128] has played an important role in defining the use of sensors and data collecting mechanisms. TQuel uses the entity-relation model to structure the system being monitored. TQuel provides a way to monitor the state of a single component. The system provides a methodology for instrumentation, with facilities to enable and disable the collection of data. The collected data is viewed by the user as residing in a historical relational database, a database in which each tuple is tagged with the time in which the tuple was valid. A query language provides a way of analyzing the system state. TQuel is centralized and, being a monitoring system, lacks support for reaction to events.

2.5 Debugging Systems

A debugger provides a programmer with the means for stopping an application during its execution at specified points, called breakpoints, and examining the state of the stopped process. Breakpoints are typically written as some predicate on the state of the process; as soon as the state of the process satisfies the specified condition, the process is stopped. A distributed debugger allows the application to consist of separate, communicating processes. Handling breakpoints for a distributed application is much more complicated, since they are now predicates over a distributed state [64]. Ideally, the system should be halted as soon as the predicate is satisfied, with each application component being in the state it was at the moment the predicate was satisfied. Works on debugging systems [16, 17, 18, 103] have contributed with ideas on consistent detection of global states and consistent reaction. The work of [108] extends the Chandy-Lamport algorithm [43] to cause the system to stop when a specified condition is satisfied.

The work of Spezialetti [131, 132] defines three classes of predicates: a *monotonic* predicate: once true, forever true. A *dependent monotonic* predicate is one that under certain conditions behave as if it were monotonic. For example, the predicate "the disk is full" is dependent monotonic in the presence of a

disk space manager; the predicate remains true as long as the condition "corresponding corrective action taken" does not hold. The last type of predicate is the *non-monotonic* predicate; the fact that such a predicate held in some past state does not mean it will hold in any subsequent state. Dependent monotonic predicates—predicates that remain true until some corrective action is executed—are of particular interest to a management system, since for such predicates the manager is guaranteed that the predicate will still hold when the reaction takes place.

2.6 Conclusions

In this Chapter, we have described the changes that are gradually being introduced to the engineering of operating systems, database systems, and programming systems due to object orientation. In some systems, object orientation is applied only at application level, in other cases the object-oriented paradigm is adopted in the design and implementation at all levels of the system. Among operating systems, Chorus/COOL is an example of the former approach and Choices is an example of the latter. Postgres and Starburst are examples of database systems with hybrid architectures; GemStone and ORION are examples of homogeneous architectures.

Relational databases do not use unique identifiers to identify their entities (tuples), they use the concept of primary keys to identify them. In contrast, distributed systems use a combination of node addresses and processes identifiers to identify processes uniquely. This difference in identification kept distributed databases and distributed systems apart for a long time. The introduction of object-oriented technologies has almost removed this "identification gap" that once separated database systems from operating systems.

Storage systems are another interesting example that some degree of unification has been achieved through the adoption of object orientation by software engineers. As we have seen in this Chapter, the uniform use of object-oriented concepts in the implementation of the systems reviewed has evolved the abstraction of file systems and introduced the abstraction of object stores and object managers. Research in object-oriented persistent programming languages and environments has done much to reduce the differences in the treatment of volatile and persistent objects.

The use of inheritance has made systems much more easier to adapt and refine. Arjuna, Choices and Apertos are examples of systems that explore inheritance to create a flexible system where user-defined classes can selectively inherit properties from system-defined classes.

Originally atomic actions were used almost exclusively in database systems. Then, implementors of operating systems and programming systems realized that the transaction concept could evolve by becoming visible to application programmers. Atomic action modules were built on top of operating systems and integrated with programming languages, resulting in an environment where all applications could benefit from the properties of atomic actions. Within this framework, it became possible for programming environments to provide persistent objects and allow programmers to structure their distributed programs as collections of transactions acting upon these persistent objects. Argus, Arjuna, Camelot, and Guide are examples of systems that adopt this programming paradigm. Later, object-oriented database management systems adopted a similar approach. ObjectStore and ODE are examples of such systems. Object orientation and explicit transactions has made these systems much more alike.

Reflection allows the implementation of a system to be exposed in a controlled way, at the right level of abstraction, to a programmer. Reflective techniques allow the interception of a message before it is executed, making possible the redefinition of its behaviour and, consequently, the behaviour of the environment. When programmers have access to this interception facility, through reflection, they can adapt the behaviour of the environment without having to learn about implementation details on which they are not interested. Apertos has used these concepts to create an object-oriented system where the gathering and interpretation of information about an active program is used to adapt the system to the needs of the program. Consequently, it is easier to implement management systems in a system like Apertos, than it is in a more conventional system.

It is within this framework of gradual convergence among software systems that Stabilis and Vigil have been designed. We have studied operating systems, database systems and programming systems and have tried to filter concepts that seemed effective to the implementation of a management system for object-oriented action-based distributed programs. During our survey, we have found that most of the systems used for programming distributed programs lack support for runtime reconfiguration of distributed programs, including some object-oriented action-based systems

BLANK PAGE
IN
ORIGINAL

Chapter 3

Stabilis

This Chapter focuses on the architecture and functionality of Stabilis. The application of object-oriented and reflective techniques resulted in the simplification of the architecture of Stabilis. Object-oriented techniques allow the management system to be incrementally extended. Reflective techniques make the behaviour of the system adaptable.

Unlike most database management systems, Stabilis is not intended to be a complete database management system with, for example, provision of query operators comparable to those found in relational database management systems. Rather, it is intended to be a tool that can be easily adapted to satisfy the needs of different areas of application.

3.1 The Architecture of Stabilis

We are going to present the architecture of Stabilis in stages. In the first stage, we define the structural model [1] supported by Stabilis, using the structural model defined by C++ as our referential. The second stage shows how layers of the architecture are created through a simple example involving the instantiation of a structural model. The last stage presents the complete architecture of Stabilis and discuss some of the problems encountered during its design and implementation.

3.1.1 Structural Model

A *structural model* describes a *static* perspective of a system, the referent system. In our case the referent system is the distributed program the user is implementing. Structural models capture the static structure of a distributed program by representing its classes, attributes, and relationships. Structural models are intrinsically metadata, since they describe the system being modeled (rather than *being* the system); models of metadata are *metamodels.*

A structural model is represented as a *directed acyclic graph* where the nodes are classes, and the edges are relationships between them. A *structure diagram* is a graphic notation for the directed acyclic graph.

Classes

C++ and Stabilis are in accordance with respect to the concept of class, both define class as a user defined type, an abstract data type [137, page 134]. A *class diagram* is a notation for the nodes, classes, of the directed acyclic graph. A class diagram is represented by a rectangle subdivided into three smaller rectangles. The top rectangle brings the name of the class. The other two rectangles contain

[1] Also called a *database schema*, or *object model* in the literature.

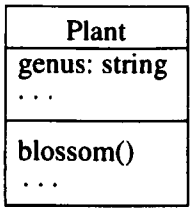

Figure 3.1: Class diagram.

lists of attributes and methods of the class respectively; they are optional. Figure 3.1 shows an example of a class diagram for a class called **Plant** with attribute **genus** of type **string** and method **blossom()**.

Relationships

In C++ relationships are represented as pointers between objects, with their meaning being determined by the programmer. Stabilis redefines the meanings of the references used to carry out relationships between database objects and shifts the responsibility for their management from program level to management level. In the structural model of Stabilis relationships between classes can be any of the following [120, pages 21-84]:

- Generalization/Specialization.

  A class, the *subclass*, inherits attributes and behaviour of another class, its *superclass*. A class with no superclass is called a *root class*. A class with no subclass is called a *leaf class*. *Multiple inheritance* exists when a class has more then one superclass. The notation for generalization/specialization is an arrow that starts at the superclass and ends at the subclass. Figure 3.2 shows an example of generalization/specialization relationship. In the example, the class **Plant** represents a higher-level abstraction of a number of subclasses such as **Epiphyte** or **AquaticPlant**. Each of these two classes can be specialized further into subclasses such as **Lichen** and **Orchid** or **SeawaterPlant** and **RiverwaterPlant**.

  Usually, the generalization/specialization relationship is interpreted in two different ways by designers of object-oriented systems:

  1. it is used as a means of inheriting or "borrowing" implementation parts from an existing class. In this case, inheritance is used mainly to increase code reuse.
  2. it is used as a means of inheriting *specification*, a subclass is designed by including the specification of its parent class as a subset of itself.

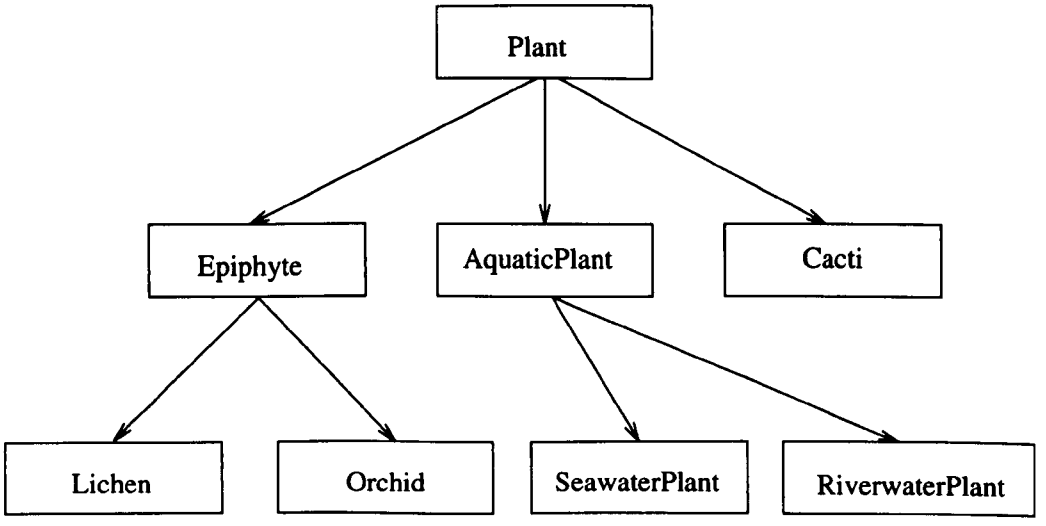

Figure 3.2: Generalization/Specialization.

In this work we adopt the latter way of using the generalization/specialization relationship with the consequence that each subclass contains the public interface of its parents as a subset of its public interface. Additionally, when using the generalization/specialization relationship the designer of the class hierarchy must guarantee that the *specification* of methods remains consistent. We can express this consistency requirement in terms of *method invariants* [63] for each method that is inherited by a subclass:

- The preconditions on a method in the subclass may only be weakened relative to the pre-conditions on the same method in the base class. The postcondition on a method in the subclass may only be strengthened relative to the post-conditions on the same method in the base class.

We can now state the requirement a designer must attend to when creating class hierarchies, it is called the *class invariant* requirement:

- The specification of a class must include, as a subset the specification of each of its base classes. Additionally, the method invariant must hold for every inherited method in the subclass.

These two requirements produce a class hierarchy in which the subclasses are subtype compatible with their parents and polymorphic substitution of subclasses for superclasses is possible. We call this use of the generalization/specialization relationship *strict inheritance.*

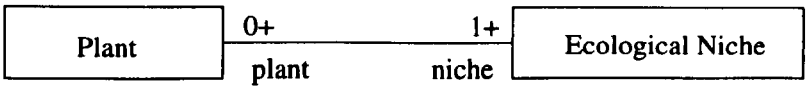

Figure 3.3: Association.

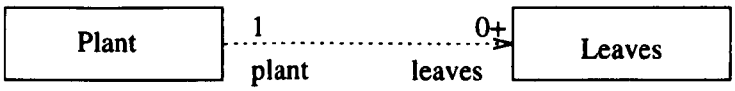

Figure 3.4: Loose Aggregation.

- Association. An association is a description of a conceptual connection between two classes. Figure 3.3 shows an example of association between the classes Plant and EcologicalNiche; it models a real-world relationship between plants and ecological niches. The notation for the association relationship is a line connecting two classes.

  The purpose of a class in an association specifies its *role* in the association. In the example, the class EcologicalNiche plays the role of being a *niche* for plants. The default name of a role is the name of its corresponding class; the annotation of default roles is optional. Roles are indicated in a object diagram by writing their names next the ends of the association line.

  The *multiplicity* of a class in an association specifies how many instances of this class may relate to an instance of the associated class. Multiplicity is symbolized in a class diagram as a non-negative integer or an interval superscripted next to the end of the association line. Examples of multiplicity marks are: "1+" (one or more), "3 – 5" (three to five, inclusive). An association line without multiplicity symbols indicates a one-to-one association. In Figure 3.3, the multiplicity "0+" means "an ecological niche has zero or more plants in it," and "1+" means "a plant must appear in at least one niche."

- Aggregation. An *aggregation* is an association where one class is in a "part-whole" or "a-part-of" relationship with the other class. The classes in the role "a-part-of" are denominated *component classes* and the class in the role of "whole" is denominated an *aggregate class.* Instances of a component class are *component objects*, and instances of an aggregate class are *aggregate objects.* The existence of a component object may depend on the existence of the aggregate object of which it is part; in this case the aggregation is termed a *tight aggregation.* When components and aggregate have independent existence, then the aggregation is termed a *loose aggregation.* The

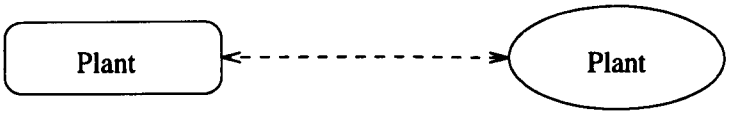

Figure 3.5: Instantiation.

notation for tight aggregation is a dashed arrow, and the notation for loose aggregation is a dotted arrow. The head of the aggregation arrow is at the component class and its butt is at the aggregate class. The multiplicity of aggregations is indicated in the same way as the multiplicity of associations. Figure 3.4 shows an example of aggregation where a plant is specified as a loose aggregation of leaves.

- Instantiation. The *instantiation* relationship relates a class to its instances. Explicitly showing the instantiation relationship is useful when both classes and instances have to be manipulated as objects. The notation for the instantiation relationship is a bidirected dashed arrow linking the class diagram to the instance diagram. The notation for an instance of a metaclass, a *metaobject*, is similar to the notation used for classes, except for the rounded corners. *Terminal instances*, instances that are not classes, are represented by an ellipsis with the name of the generator class and, optionally, the state of the object inscribed. Figure 3.5 shows a metaobject, instance of the metaclass **Plant**, linked to a terminal instance of the class **Plant**.

The analysis of the structural model defined by C++ carried out so far shows that C++ is capable of providing the functionality required of Stabilis except for:

R_1 The need to manage structural data about distributed programs in order to resolve queries. The structural model of C++ does not allow the management of information related to applications. For example, there is not any built-in mechanism for obtaining the class of an object during runtime; during compilation such information is available in the compiler's data structures, but it is not transferred to the executable code. The data maintained by Stabilis about the distributed program is the program's structural model and related indexing information.

R_2 The need for redefining the meaning of C++ references for database objects to allow the implementation of pre-defined reference semantics.

The requirements above induce the definition of an extended structural model, the structural model of Stabilis. In the following list, principles S_1 to S_4 and S_6

define the structural model of C++. Principle S_5 has been added in response to requirements R_1 and R_2.

S_1 A *class* describes a group of objects with similar properties (attributes), common behaviour (methods), common relationships, and common semantics.

S_2 Activation of objects is carried out only by message passing: a message specifies which method to execute and its actual parameters.

S_3 An *object* is an instance of a *class*. The current values of the attributes of an object define its *state*. The state of an object comprises the state of its instance variables, including relationships.

S_4 Every object belongs to a class that specifies its attributes and behaviour. Objects are dynamically generated from this model, they are denominated *instances of a class.*

S_5 A class is also an object, instantiated by another class, its *metaclass*. From (S_4) we have the association of each class of a structural model to a metaclass. Metaclasses specify the attributes and behaviour of a class as an object. In Stabilis, the original metaclass is the class **Class**.

S_6 A class can be defined as a subclass of one (or many) other class(es); subclasses are defined through the inheritance mechanism. In Stabilis, the class **Object** represents the most common behaviour shared by all objects; it is the original root class, that is, all classes in a structural model are, directly or indirectly, subclasses of the class **Object**.

3.1.2 An Example Program

In the second stage of the description of the architecture of Stabilis we rely on a simple example program to show how the structural model of Stabilis moulds the architecture of the management system.

Although understanding the architecture of Stabilis and its overall implementation is important, it is crucial to separate what belongs to the internals of Stabilis from what is perceived by its users. A useful metaphor for making this separation comes from the workings of an orchestra. We can think of the programming environment as being a *concert*. Users, who we think of as the *audience*, only get to see and hear what is played *on-stage*. We can think of the programming interface as being on-stage. The implementation of the concert, that is,

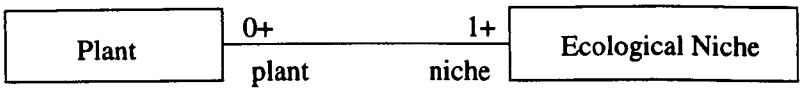

Figure 3.6: Niche structural model.

orchestration of the music, tuning of the instruments, and rehearsals are *backstage*: they support what happens on-stage, but the audience does not get to see them. Finally, implementors are the *maestro* and musicians: they get to see what happens both on-stage and backstage, and they are responsible for the concert. During the remaining of this work we are going to make reference to the orchestra metaphor as a navigational aid which helps the reader distinguish between what belongs to the internals of Stabilis from what is seen at its programming interface.

A simple Stabilis program consists of two autonomous sub-programs:

1. The *schema program* has a definitional role, when executed it generates objects that *define* a structural model: the classes, attributes, methods, and relationships of the structural model. Backstage, the generation of these objects creates an internal representation of the structural model. The internals of Stabilis use information stored in these objects, in fact metaobjects, to create terminal instances and to access their state and methods. In general, the schema program is run only once to define the structural model which is going to be used as a basis for the execution of several application programs.

2. The *application program* has a querying role. When executed it generates and/or retrieves database objects which are instances of the classes defined by the metaobjects created by the schema program. The objects created by the application program belong on-stage, they are the objects of the user's distributed program.

To make things concrete, we analyze the execution of a Stabilis program (schema and application sub-programs) for the structural model shown Figure 3.6; let us call it the Niche structural model. The structure of the schema program (Program 3.1) mirrors the Niche model directly. Lines 1 and 2 of the schema program (Program 3.1) have the function of creating instances of the class Class for the classes EcologicalNiche and Plant of the Niche model. Recalling the concert metaphor, the instances created by lines 1 and 2 are part of backstage, they are metaobjects. The creation of the metaobjects like the ones created by lines 1

and 2 is determined by principles S_3 and S_4: without metaobjects (classes) it is impossible to create objects (instances of a class). The existence of the metaclass Class, determined by principle S_5, allows the creation of classes.

We still have to create the metaobjects that define the attributes of the classes EcologicalNiche and Plant, lines 3 and 4 of the schema program (Program 3.1) do exactly this for the class EcologicalNiche. Line 4 is responsible for creating the relationship between the metaobject that defines the class EcologicalNiche and the metaobject that defines its attribute **name**. In a similar way, lines 5 and 6 create metaobjects that define the attribute **genus** of the class Plant. Finally, line 7 carries out the creation of the metaobject that defines the association between the two classes of the Niche model. Note that the variables representing the classes of the Niche model are passed as parameters to the constructor of the class Association which guarantees the creation of the necessary relationships between the metaobjects involved.

Program 3.1 Schema for Niche structural model.

```
main() {
 (1) Class* enclass = new Class("Class(name = 'EcologicalNiche')",...);
 (2) Class* pclass = new Class("Class(name = 'Plant')",...);
 (3) StringAttribute*
 enname = new StringAttribute("StringAttribute(name = 'name' &&
 key = 1)",...);
 (4) enclass→relate("Attribute", enname);
 (5) StringAttribute*
 pgenus = new StringAttribute("StringAttribute(name = 'genus' &&
 key = 1)",...);
 (6) pclass→relate("Attribute", pgenus);
 (7) Association*
 en_p = new Association("Association(left_role = 'plant' &&
 left_min_card = 0 && left_max_card = -1 && right_role = 'niche'
 && right_min_card = 1 && right_max_card = -1 && key == 1)",
 enclass, pclass,...);
}
```

We can now concentrate on the execution of the application program (Class 3.1 and Program 3.2). The header file (Class 3.1) has been automatically generated by Stabilis using Niche's database schema. Note the inheritance from the class Object, determined by principle S_6. Also, in the code fragment we can see that

the constructor of class EcologicalNiche takes a query expression as its first formal parameter. Every class definition fit to execute under the control of Stabilis has in its definition a family of constructors that accept query expressions as their parameters. Later, we are going to see how these query expressions are interpreted. Principle S_4 is materialized as the data structures and algorithms which allow query resolution. There is a similar class definition for the class Plant. Lines 1 and 2 of the application program (Program 3.2) show how the kernel of Stabilis is activated, the name of the user-defined structural model is passed as a parameter to Stabilis. Finally, the program creates an instance of each of the classes EcologicalNiche and Plant, relates them and terminates (Program 3.2, lines 3, 4, and 5).

Class 3.1 Class EcologicalNiche.

```
/* Automatically generated code */
class EcologicalNiche: public Object {
public:
 EcologicalNiche(String query_expression, ...);
 ~EcologicalNiche();
protected:
 String name;
};
```

Program 3.2 Application for Niche structural model.

```
main()
{
 (1) String name("EcologicalNicheStabilis");
 (2) Stabilis* stabilis = new Stabilis(REINCARNATION, ..., name);
 (3) EcologicalNiche*
 niche = new EcologicalNiche("EcologicalNiche('name = "tundra"')", ...);
 (4) Plant* plant = new Plant("Plant('genus = "Riviea"'", ...);
 (5) plant→relate("niche", niche);
}
```

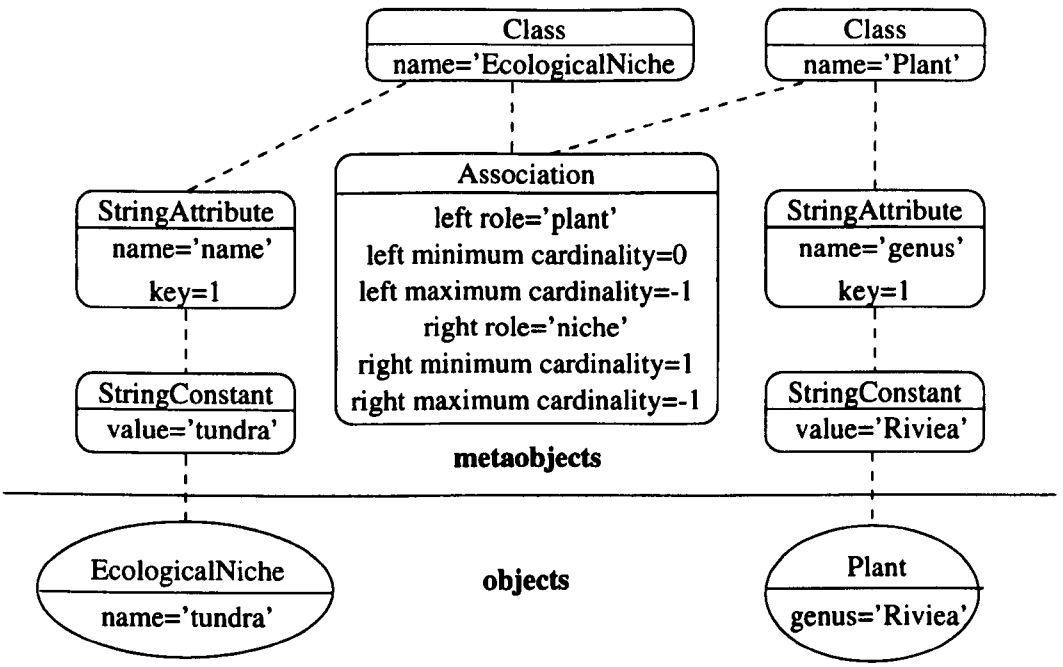

Figure 3.7: A first hint of the architecture.

The execution of both programs caused several new objects to be created and connected as shown in Figure 3.7. They can be divided in two groups: metaobjects (classes) and objects (terminal instances). In the former group, are the objects that form part of the indexing structures of Stabilis. Objects at this level of the architecture compose a bidirectional graph upon which the algorithms for query resolution operate. These *backstage* objects represent a Stabilis user program rather than the domain of it. In fact, everything that happens inside the implementation of Stabilis is considered to be at "meta" level with respect to the user program; i.e., *about* the user program itself, rather than about whatever the user program happens to be about. As we have seen, principles S_3 to S_5 were used in the instantiation of this layer of Stabilis' architecture. If we were not looking into the inside of an implementation we might not even have noticed that there were such metaobjects; being part of backstage they are normally hidden from the programmer. In the latter group, the two objects (which are part of on-stage) are terminal instances and the connections they have to their respective metaobjects (classes), principle S_4, is what makes them subject to the management algorithms implemented by Stabilis.

Given this overall picture, the remaining of this Chapter will detail further the architecture and operation of Stabilis.

3.2 Implementation

The previous example has shown how a schema program determines the creation of part of the backstage machinery used by Stabilis to manage structural information related to user programs. It has also shown that terminal instances are managed by algorithms resident inside their corresponding classes (metaobjects). The architecture of Stabilis, as it stands, is formed by two layers: an object layer and a metaobject layer. Is this two-layered architecture sufficient to the operation of the system? The answer to this question is based on a uniform application of principle S_5 which states that *classes are objects instantiated by other classes, their metaclasses.* If we have classes (metaobjects) in the system, then we must have metaclasses. Consequently, the answer to the question is *no.* This negative answer takes us into the subject of this Section: the formation of the third layer of the architecture of Stabilis.

Stabilis' third layer implements the basic algorithms for management of structural information, this layer is the *kernel* of Stabilis. In the example program the metaobject layer was created by the execution of a schema program. Similarly, the kernel of Stabilis, the meta-metaobject layer of the system, is created by the execution of a special schema program which encodes the structural model of Stabilis itself. A detailed discussion of the special schema program is beyond the scope of this work. Instead, we resort to the structural model of Stabilis, as shown in the object diagram of Figure 3.8, to explain the creation of the meta-metaobject layer.

At first sight, the object diagram (Figure 3.8) looks complex because of the number of classes, attributes, and relationships represented. In fact, this apparent complexity disappears when the graph is seen as a design of the structural model defined in 3.1.1 (page 58). Every class, attribute, and relationship represented in Figure 3.8 is an interpretation of a concept laid down during the definition of principles S_1 to S_6. For example, the subtree whose root is the class **Relationship** has been designed as so because of the definitions of Section 3.1.1. There we say that a relationship can be specialized into subclasses: **Generalization/Specialization**, **Aggregation**, **Association**, and so forth. The representation of the class **Class** as a tight aggregation of objects reflects principles S_1 to S_4. The other classes and relationships of the object diagram have been designed in a similar way. The interpretation of the object diagram can be eased further by dividing it into two intersecting subdiagrams:

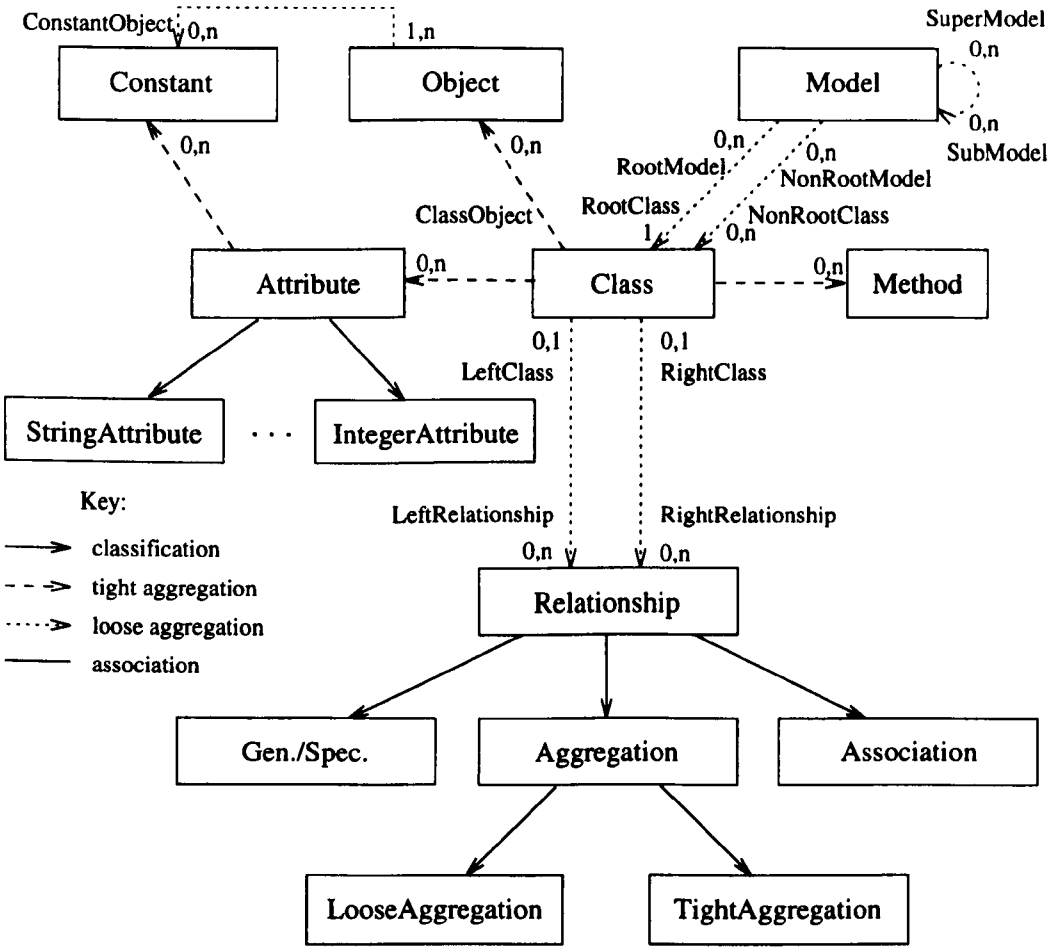

Figure 3.8: The structural model of Stabilis.

1. one is formed by the classes **Class**, **Attribute**, **Method**, **Object**, **Constant** and those of the class hierarchy whose root class is **Relationship**. The instantiation of the classes represented in this diagram in terms of themselves form the kernel of Stabilis.

2. the other is formed by the classes **Model** and **Class**. This diagram show how structural models are represented in Stabilis. A structural model has always an instance of the class **Model** associated to its root class. Instances of the class **Model** are the entry points to navigate database schemas.

We are going to study each of the subdiagrams in turn and show how their instantiation determines the architecture and functionality of Stabilis.

3.2.1 Representing Classes

On-stage, the Stabilis programmer begins the design of his distributed program with the creation of a structural model where the definition of classes play a central role. It is natural, therefore, to begin our study of the kernel of Stabilis by examining how classes and metaclasses are represented internally.

The first metaclass to be represented is the metaclass Class. It is created through the instantiation of a special constructor of the class Class. This meta-metaobject is the root of the instantiation graph formed by objects, metaobjects, and meta-metaobjects. Its existence guarantees that any other class or metaclass of the model can be created; the metaclass Class is the only class in the system that can instantiate itself.

The representation of the metaclass Class is not complete with the creation of an instance of the class Class for itself. Next, we must represent the attributes of the metaclass Class. For example, the representation of the attribute name of the metaclass Class requires the creation of a metaclass Attribute and its initialization with the name of the attribute of the metaclass Class. In the structural model of Stabilis (Figure 3.8) we can see that the class Attribute is modeled as a tight aggregation of classes Constant[2]. Therefore, the representation of the metaclass Attribute implies the creation of the metaclass Constant. The representation of the kernel of Stabilis is not complete until we have instantiated the whole structural model of Stabilis using metaclasses. The booting up of Stabilis is carried out by a special schema program whose purpose is the creation of these metaclasses. Figure 3.9 shows the three layers of the architecture. In the **meta-metaobjects** layer of the figure we can see, at the top right, the instances of the metaclasses Class, StringAttribute, and three instances of the metaclass StringConstant. Together, they represent a snapshot of part of the kernel, that is, the meta-index, of Stabilis. These meta-metaobjects index the metaobjects, **metaobjects** layer, that in their turn index the database objects, **objects** layer.

[2]The name of the class *Constant* is unfortunate, since instances of Constant are not constants at all; they are **values** of attributes. The name Constant has been retained for practical reasons.

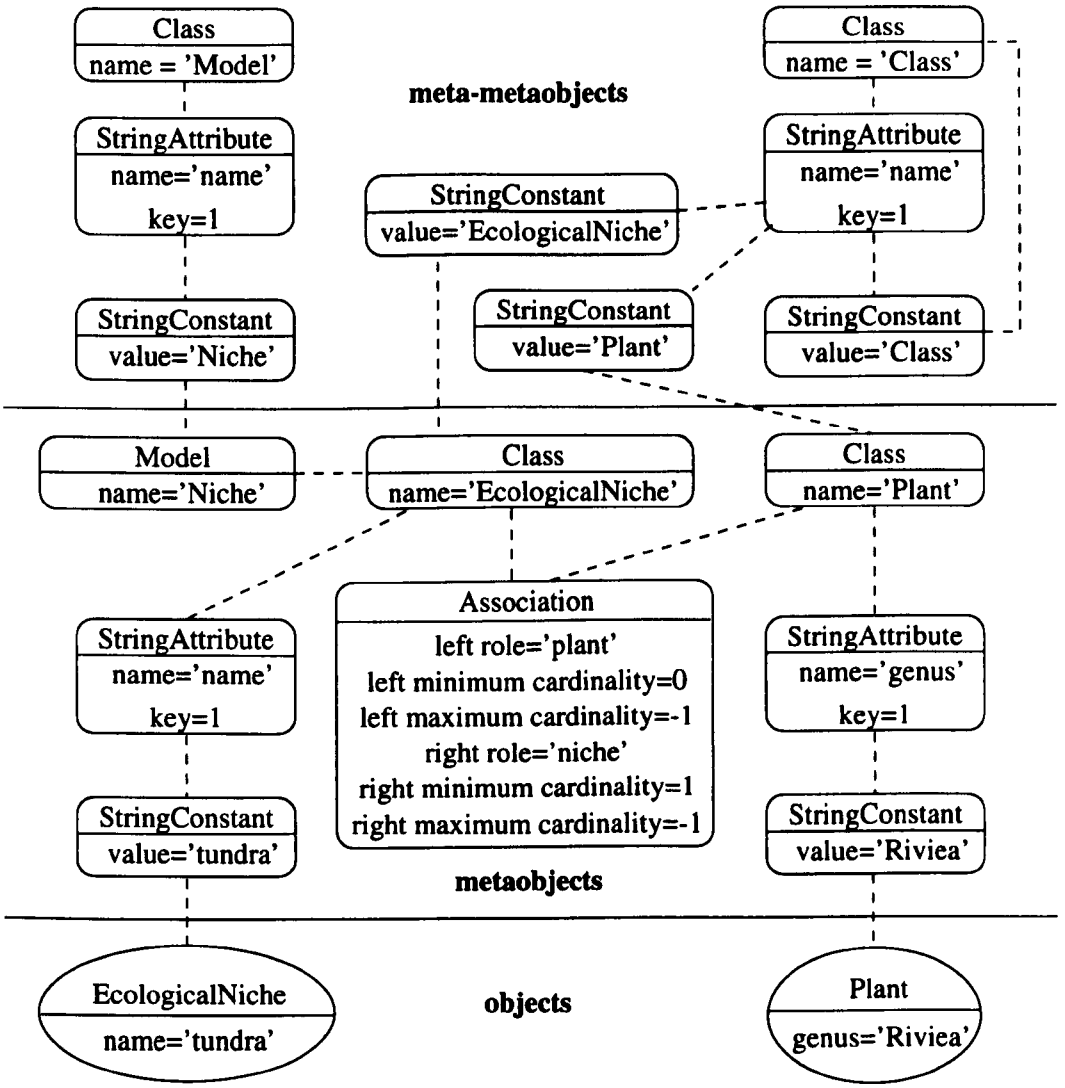

Figure 3.9: A hint of the three layers.

The metaclass Class indexes all metaclasses of the structural model of Stabilis. For example, we can traverse the index formed by metaclasses Class, Relationship, and Association (Figure 3.8) and reach the classes (metaobjects) which represent associations between any two classes of any stored structural model.

3.2.2 Representing Models

Understanding the representation of models is simpler once we have seen how to represent the metaclasses of the kernel of Stabilis. In the kernel of Stabilis, classes

are represented as a graph: objects are nodes and relationships are edges. Models are represented as a graph superimposed on the kernel graph. The structural model of Stabilis (Figure 3.8) shows us how the superimposition is achieved. In the structural model, the class **Model** has three relationships, two with the class **Class** and one with itself. The two relationships with the class **Class** indicate the intersection of the representations. They are loose aggregations, meaning that structural models are formed of aggregations of classes that can be changed. In a loose aggregation the components are autonomous objects in relation to the aggregate object; components usually outlive the aggregate. The loose aggregation relationship between **Model** and itself (Figure 3.8) allows for the representation of hierarchical models.

An instance of metaclass **Model** is created by the special schema program which indexes all structural models represented in Stabilis. Structural models are represented as graphs whose internal nodes are classes **Model** and whose leaf nodes are classes **Class**. Models are to Stabilis what database schemas are to traditional database management systems.

3.2.3 On Circularity

Principle S_5 (see page 63) of the structural model states that classes are objects, instantiated by another class, denominated metaclass, represented by a metaobject. In practice, the circularity expressed by this principle implies a recursive process of instantiation. The general step of recursion is provided by the instantiation of the structural model supported by Stabilis (Figure 3.8) *for* a given *target* structural model. The architecture of Stabilis is thus determined by successive applications of this step. Let us fix the metaobject layer of the architecture as our referential. Reasoning forwards takes us to the layer of meta-metaobjects, this level results from the the application of principle S_5 to the structural model of Stabilis *itself*. In theory, one could carry on with the regression infinitely[3]; in practice, the depth of the instantiation tree is kept at three. Reasoning backwards and using the foregoing layers as support, takes us to the base of the recursion, to the layer where terminal instances are found.

As already written, heuristic considerations determine the recursive instantiation of structural models to be stopped at meta-metaobject level. The ending of the recursion poses a chicken-and-egg problem: an object cannot be created until its class exists, but this class metaobject needs to be an instance of itself. In Stabilis, the most important chicken-and-egg problem occurs in the creation of an instance of the class **Class** to represent itself, a meta-metaobject. The removal

[3] A metaclass is a class which instantiates a class, a meta-metaclass is a class which instantiates a metaclass, a meta-metametaclass is ...

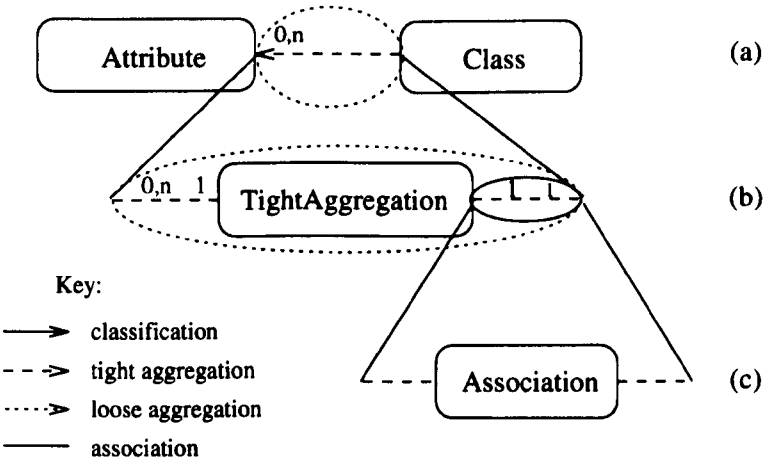

Figure 3.10: Circularity in relationship representation.

of the class **Class** circularity involved the programming of special code used only during the initialization of Stabilis:

- the class **Class** has a autonomous primary constructor,
- creation of a volatile-only version of the kernel code of Stabilis which uses the special constructors of class **Class** and of others classes of the structural model of Stabilis. This volatile-only version of Stabilis' kernel has its generation controlled by compilation flags, meaning that two variations of Stabilis exist: a variation that is used only to boot the kernel and a variation that is used during Stabilis' normal regime of operation.

Relationships

There is another circularity in the kernel of Stabilis (Figure 3.8). How do we represent (implement) the relationships between the classes of this model? Let us take as an example the tight aggregation relationship existent between the class **Class** and the class **Attribute** (Figure 3.10a). It is possible to implement this relationship through the creation of an instance of the class **TightAggregation** (Figure 3.10b). Now, the representation of the kernel is composed of instances of metaclasses **Attribute**, **TightAggregation**, and **Class**. The link between these metaclasses associations. Again, it is possible to implement these associations using metaclasses **Association** (Figure 3.10c), and then implement the associations between these **Association** metaclasses using metaclasses **Association**, and so on. We stop the recursion at the level shown in Figure 3.10b by not implementing the associations in terms of **Association** metaclasses. Instead, we use instances of

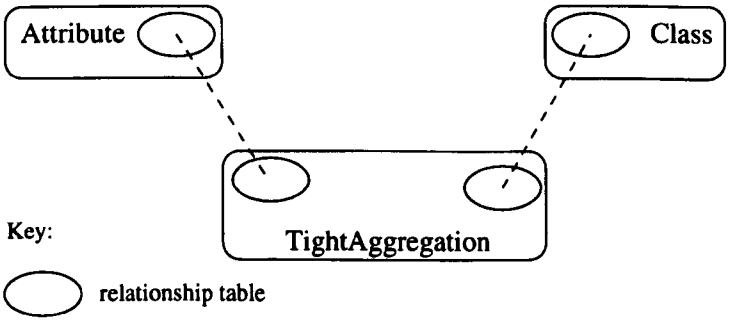

Figure 3.11: Relationship table.

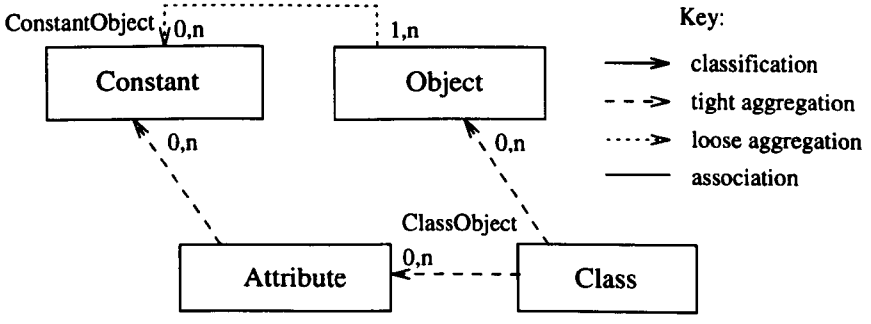

Figure 3.12: Part of the structural model of Stabilis.

the class hierarchy whose base class is the class RelatioshipTable. The links shown in Figure 3.10b are implemented by instances of RelationshipTable; the resulting configuration of objects is shown in Figure 3.11. Every database object has a relationship table as its attribute.

The class RelationshipTable has as its attribute a list of relationship table entries. Relationship table entries are implemented as classes and there is a one-to-one mapping between the class hierarchy of relationships and the class hierarchy of relationship entries. For example, we have a class Association to represent the association relationship and we have a class AssociationEntry to represent links of type *association* between objects.

3.3 Classes Interfaces

Our analysis of the protocols implemented by the kernel of Stabilis starts with a brief description of the interfaces of some of the classes involved in the implementation of the protocols. The classes chosen are represented in Figure 3.12. They

Class Name	Class Definition	lines
Class	3.2	1
Attribute	3.3	1
Constant	3.4	1,2
Object	3.5	15,16

Table 3.1: Family of primitive constructors.

Class Name	Class Definition	lines
Class	3.2	2,3,4
Attribute	3.3	2,3,4
Constant	3.4	3,4,5,6
Object	3.5	17,18,19

Table 3.2: Family of query constructors.

have been selected because they represent a significant cross-section of the methods used in the implementation of the protocols of Stabilis' kernel, especially the protocols related to query resolution. We hope that the grouping of methods into families provides a less fragmented view of the classes interfaces. Later, during the discussion of queries, we detail further how some of these methods are used.

3.3.1 Constructors

The constructors of classes of the kernel can be divided into two families. The first family comprises the constructors that are used only during the booting process (Table 3.1). These constructors are primitive in the sense that they do not depend on the previous existence of metaclasses for their instantiation.

The second family of constructors (Table 3.2) is used during the normal operation of the system. Once the metalevels of the architecture have been created then objects can be manipulates using query expressions. Three parameters are of interest: query expression, mode of instantiation, and operation history. A *query expression* is an expression involving classes and their key attributes.

The parameter *mode of instantiation* can be one of:

- **Birth**: meaning that the creation of a database object is mandatory. For example, if a program contains a statement like "Plant* plant = new Plant(" Plant('genus = "Riviea" ')", ..., Birth, ...)", then Stabilis creates an instance of **Plant** and initializes its attributes with the values supplied in the query expression. In this simple example, attribute **genus** is set to "Riviea".

- Reincarnation: If a database object which satisfies the query expression is found then it is returned, that is, restored from the object store into the object manager and made available for processing.

- Provide: If a database object which satisfies the query expression is found, then return it, otherwise, create it.

Class 3.2 Class Class.

```
class Class : public Object {
public:
 (1) Class(Context*, Birth, OpHistory*); /* Class for Class */
 (2) Class(String sexpr, Context*, Birth, OpHistory*);
 (3) Class(String sexpr, Context*, Reincarnation, OpHistory*);
 (4) Class(String sexpr, Context*, Provide, OpHistory*);
 (5) ~Class();
 (6) String get_name();
 (7) OpHistory* index();
 (8) OpHistory* traverse(AttrExpr* attr_expr, String attribute_name);
 (9) OpHistory* traverse(Class*& meta, String& meta_name, String role);
 (10) OpHistory* insert(String class_name, Class*);
 (11) OpHistory* locate(String class_name, Class*&);
 (12) OpHistory* insert(String attribute_name, Attribute*);
 (13) OpHistory* locate(String& attribute_name, Attribute*&);
 (14) OpHistory* update(String& attribute_name, Constant*, Object*);
 (15) OpHistory* adjust(String& attribute_name, Object*);
 (16) OpHistory* gen_code(String path, unsigned version, StringList& classes);
 (17) virtual ostream& print(ostream&);

private:
 (18) CheckPointingList<MetaEntry> metas;
 (19) CheckPointingList<AttributeEntry> related_attributes;
 (20) Constant* first_constant;
 (21) Attribute* search_attribute;
};
```

The third parameter is an *operation history*, class OpHistory. It is used to store lists of messages, warnings, and error codes. At any moment a programmer can poll operation histories to determine the result of an operation executed by Stabilis. All systems implemented in Stabilis adopt the same mechanism to communicate results of operations. The use of an operation history has made the system much more easy to program and debug.

Class 3.3 Class Attribute.

```
class Attribute : public Object {
public:
 (1) Attribute(String a_name, KeyFlag a_key_flag, Context* context,
                        Class* meta, Birth, OpHistory*);
 (2) Attribute(String sexpr, Context*, Birth, OpHistory*);
 (3) Attribute(String sexpr, Context*, Reincarnation, OpHistory*);
 (4) Attribute(String sexpr, Context*, Provide, OpHistory*);
 (5) Attribute(Pip*, Context*, Reincarnation, OpHistory*);
 (6) OpHistory* index(ParentTightAggregationEntry* objects_entry);
 (7) OpHistory* update(Constant* new_constant, Object*);
 (8) OpHistory* adjust(Object*);
 (9) OpHistory* relate(Constant*, ParentTightAggregationConstantEntry*);
 (10) OpHistory* unrelate(Constant*, ParentTightAggregationConstantEntry*);
 (11) OpHistory* first(Constant*&);
 (12) OpHistory* next(Constant*&);
 (13) String get_name();
 (14) Boolean is_key();
 (15) unsigned operator==(Attribute&);
 (16) virtual ostream& print(ostream&);
 (17) virtual OpHistory* gen_code_header(ofstream&);
 (18) virtual OpHistory* gen_code_cc(ofstream&, String class_name);

protected:
 (19) String name;
 (20) KeyFlag key_flag;
};
```

Class 3.4 Class Constant.

```
class Constant : public Object {
public:
 (1) Constant(Birth, OpHistory*);
 (2) Constant(Uid&, Reincarnation, OpHistory*);
 (3) Constant(String sexpr, Context*, Birth, OpHistory*);
 (4) Constant(String sexpr, Context*, Reincarnation, OpHistory*);
 (5) Constant(String sexpr, Context*, Provide, OpHistory*);
 (6) Constant(Pip*, Context*, Reincarnation, OpHistory*);
 (7) OpHistory* relate(Object*);
 (8) OpHistory* unrelate(Object*);

 (9) virtual Boolean operator== (Constant*) = 0;
 (10) virtual Boolean operator≠ (Constant*) = 0;
 (11) virtual Boolean operator< (Constant*) = 0;
 (12) virtual Boolean operator> (Constant*) = 0;
 (13) virtual Boolean operator≤ (Constant*) = 0;
 (14) virtual Boolean operator≥ (Constant*) = 0;
};
```

Class Name	Class Definition	lines
Class	3.2	7-15
Attribute	3.3	6-15
Constant	3.4	7-14

Table 3.3: Family of indexing methods.

3.3.2 Indices

Indices are search structures used by database systems to resolve queries. The family of methods related to indices and query resolution is listed in Table 3.3. Next, we give a brief description of each of these methods.

- **index**: These methods are responsible for the construction of the initial indices of Stabilis (Classes 3.2 line 7 and 3.3 line 6), that is, they are responsible for building the graphs that implement indices. They are executed only during the start-up of the system. First metaclasses are created and related to each other then, close to the end of the start-up process, each metaclass receives a **index()** message and that causes the initialization of all the indices of metaobjects and meta-metaobjects.

Class 3.5 Class Object.

```
enum ObjStatus { UNDEFINED_STATUS, NORMAL, MODIFIED,
TO_BE_MADE_PERMANENT, FAILED_TO_MAKE_PERMANENT,
TO_BE_MADE_VOLATILE, FAILED_TO_MAKE_VOLATILE, POSSIBLY_STALE };

class Object : public LockManager {
public:
 (1) String get_class_name(); Boolean is_a(String class_name);
 (2) OpHistory* put(String sexpr);
 (3) OpHistory* relate(String related_object_role, Object* related_object);
 (4) OpHistory* cross_relate(String related_object_role, Object* related_object);
 (5) OpHistory* unrelate(String related_object_role, Object* related_object);
 (6) OpHistory* cross_unrelate(String related_object_role, Object* related_object);
 (7) virtual ~Object();

private:
 (8) Boolean clustered; ObjStatus activation_status;
 (9) String object_name; Uid cache_uid;
 (10) Class* meta; Expression* assign_expression;

protected:
 (11) ObjStatus object_status; String host_name;
 (12) Uid* object_uid; ObjectState* object_state;
 (13) ClassPath class_path;
 (14) RelationshipTable relationships; AttributeTable attributes;
 (15) Object(Birth, OpHistory*);
 (16) Object(Uid&, Reincarnation, OpHistory*);
 (17) Object(String sexpr, Context*, Birth, OpHistory*);
 (18) Object(String sexpr, Context*, Reincarnation, OpHistory*);
 (19) Object(String sexpr, Context*, Provide, OpHistory*);
 (20) OpHistory* activate();
 (21) OpHistory* make_permanent();
 (22) OpHistory* make_volatile(LockMode mode = READ);
 (23) virtual void make_abort();
};
```

- **traverse** and **locate**: Once the initial graph has been constructed it is possible to traverse it using these methods. For example, during the resolution of a query subexpression it may be necessary to go from metaclass Class to metaclass Attribute, through a certain Relationship, in order to locate a database object. The graph traversal algorithms, that is, query algorithms, used by the indices are implemented by methods **traverse**, **locate**, and their auxiliaries. For example, method **traverse** (Class 3.2 line 9) is used to traverse from one metaclass, the origin, to another, the target, given the name of the role the target metaclass plays in relation to the origin.

- **insert**, **adjust**, and **update**: These methods are used to traverse and update indices. For example, method **update** (Class 3.2, line 14) is used during the update of an attribute value, given the name of the attribute and the object from which such value is obtained.

Indices and Relationship Tables

The relationship between Constant and Object is very important to the resolution of queries. Ultimately, it is through this relationship that database objects can be found. Let us take as an example the C++ statement "Plant* plant = new Plant(" Plant('genus = "Riviea" ')", ..., Reincarnation, ...)". This statement tells Stabilis to find an instance of class Plant whose **genus** is "Riviea". Using the traversal methods Stabilis first locates the metaclass Plant. Next, it traverses from metaclass Plant to the metaclass Attribute responsible for indexing attribute **genus**. The metaclass Attribute for **genus** indexes all instances of Plant whose attribute is **genus**. The query resolution algorithm reaches metaclass Constant and selects those *constants* with value "Riviea". Finally, these constants point to the objects we are looking for; the last arc to be traversed is the arc between Constant and Object.

During the traversal of this index path we have had to go from one node to another, that is, from one metaobject to another. If the indices and objects were in the same address space then traversing the graph could simply be reduced to traversing an ordinary pointer. Inevitably, our indices are distributed, meaning that such traditional implementation of an index is impossible.

Before proceeding any further, let us recall that in our architecture relationship tables are responsible for implementing pointers between objects (Section 3.2.3). Thus, in Stabilis, the implementation of pointers to distributed objects is carried out by relationship tables and their auxiliary classes.

Relationship tables are implemented as dynamic lists of relationship table entries (Class 3.6) and the entries (Class 3.7), in their turn, are implemented as dynamic lists of pointers to database objects. Fundamentally, what these classes implement is a graph, using adjacency lists. There is one relationship table entry for each relationship held by an object. Finally, pointers to objects are implemented as instances of the class **Pip**[4] (Class 3.8).

The class **RelationshipTable** (Class 3.6) exports methods that allow an object to find which of its table entries indexes a given relationship, that is, an arc of the index graph. There are two useful ways of specifying arcs in this graph. In one way, we specify the name of the role played by the destination class, that is, destination node. In the other way, we specify the name of the role played by the origin class and the name of the related class. These two modes of specifying arcs of the graph are reflected in signatures of methods **find** (Class 3.6 lines 2,3). The methods **add_pips** (lines 4,5) are used to add a set of pips to a given relationship table entry. There is a method that allows an object to request the set of pips held by its relationship table (line 6). The method of line 7 allow a new relationship table entry to be added to the relationship table. Lines 8 and 9 bring the signatures of the **relate** and **unrelate** methods that allow the creation of a link between two objects. Finally, there are two methods that allow an object to test if its relationship table is contains a link to a given object or pip (lines 10 and 11). Essentially, these methods allow an object to ask whether or not it is related to another object. Finally, the method shown in line 12 allows an object to update its relationship tables with information extracted from the object passed as parameter.

[4]Pips and seeds encode information that allows them to grow into the structures that produce fruits of their kind, in short, pips are pointers to fruits. Thus, the analogy between pips and persistent pointers.

Class 3.6 Class RelationshipTable.

```
class RelationshipTable :
 public CheckPointingList<RelationshipTableEntry> {
public:
 (1) RelationshipTable();
 (2) RelationshipTableEntry* find(String related_object_role);
 (3) RelationshipTableEntry* find(String local_role, String related_object_class);
 (4) void add_pips(String related_object_role, Pips*);
 (5) void add_pips(String local_role, String related_object_class, Pips*);
 (6) OpHistory* get_related_pips(Set<Pip>*, String local_role,
                                 String related_object_class);
 (7) OpHistory* enter(RelationshipTableEntry*);

 (8) OpHistory* relate(String related_object_role, Object* related_object, String&
local_role);
 (9) OpHistory* unrelate(String related_object_role, Object* related_object, String&
local_role);
 (10) Boolean contain(String related_object_role, Object*);
 (11) Boolean contain(String related_object_role, Pip*);
 (12) Boolean adjust(Object*);
};
```

The class RelationshipTableEntry and its derived classes hold information used to determine if an arc can be traversed or not. They hold the name of the role played by the object, the names of the classes of the objects involved in the relationship, the cardinality of the relationships, and whether or not such relationship can be traversed. If a relationship is a key attribute, that is, it can be traversed during query resolution, then **key_flag** takes the value **key**, otherwise, it takes the value **nonkey**.

The interface of the class RelationshipTableEntry and its derived classes has methods similar to those found in RelationshipTable.

The class Pip holds information concerning the location of an object in the distributed system. It holds the class of the object, its unique identifier (UID), the host where it resides, a flag saying if the object is clustered inside a container object, and a C++ pointer to the object. This pointer can be used only when the object is made active in the same address space where the pip is.

A query is resolved to a set of pips. Stabilis then can then obtain a handle to the objects of the set of pips by asking the pips to activate themselves (Class 3.8 lines 12 and 13).

Pips have a very flexible set of constructors that allow them to be built using the minimum information available at the time of creation (Class 3.8 lines 6 to

Class 3.7 Class RelationshipTableEntry.

```
class RelationshipTableEntry {
protected:
 (1)  String local_role;
 (2)  String related_object_class;
 (3)  String related_object_role;
 (4)  int local_min_card;
 (5)  int local_max_card;
 (6)  KeyFlag key_flag;
 (7)  Pips pips; /* set of pips */

public:
 (8)  RelationshipTableEntry();
 (9)  virtual ~RelationshipTableEntry();
 (10) RelationshipTableEntry(String& a_related_object_role);
 (11) RelationshipTableEntry(String& local_role, String& related_object_class,
                             String& related_object_role, int local_min_card,
                             int local_max_card, KeyFlag key_flag);
 (12) OpHistory* relate(Object* related_object, String& local_role);
 (13) OpHistory* unrelate(Object* related_object, String& local_role);
 (14) Boolean contain(Object*);
 (15) Boolean contain(Pip*);
 (16) Boolean adjust(Object* object);
 (17) virtual RelationshipTableEntry* copy();
 (18) unsigned is_key();
 (19) void add_pips(Set<Pip>*);
 (20) unsigned empty();
 (21) Pip* first();
 (22) Pip* next();
 (23) void merge(RelationshipTableEntry*);
 (24) String get_local_role();
 (25) String get_related_object_role();
 (26) String get_related_object_class();
};
```

10). There is also a family of access methods (lines 14 to 18). The method of line 19 is used by relationship table entries to verify the equality between pips during the traversal of the adjacency lists. Two pips are considered to be equal in contents when every attribute of both has equal values.

Class 3.8 Class Pip.

```
class Pip {
private:
 (1) String object_class;
 (2) Uid object_uid;
 (3) String host_name;
 (4) Boolean clustered;
 (5) Object* object;

public:
 (6) Pip();
 (7) Pip(Uid uid);
 (8) Pip(String a_object_class, Uid, String a_host_name, Object*);
 (9) Pip(String a_object_class, Uid, String a_host_name, Boolean a_clustered,
Object*);
 (10) Pip::Pip(Pip* p);
 (11) ~Pip();
 (12) Object* grow();
 (13) OpHistory* grow(Context*, Object*&);
 (14) String get_object_class();
 (15) Uid get_object_uid();
 (16) String get_host_name();
 (17) void set_host_name(String& a_host_name);
 (18) void adjust(Pip*);
 (19) unsigned operator==(Pip& key);
};
```

Now that we have had an overview of the classes interfaces that implement Stabilis we can concentrate on the workings of the object manager. The description of the object manager sets in place the last piece of information we need to understand in detail how queries are resolved, i.e., how Stabilis operates when considered from the point of view of an application programmer.

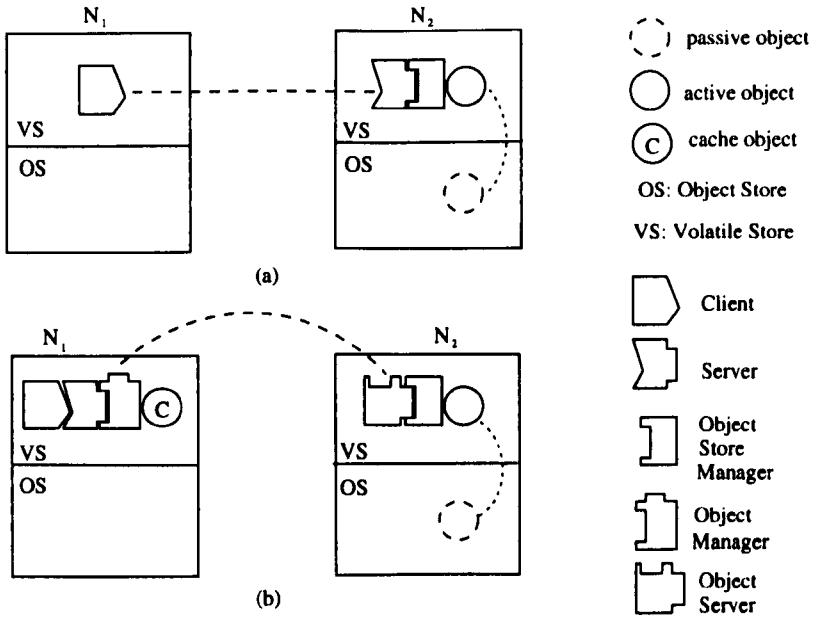

Figure 3.13: Architecture of the Object Manager.

3.4 Object Manager

The object manager provides an abstraction of database objects that exempts the use of the stub generator while giving the possibility of caching objects to improve the efficiency of a database application.

Several factors have been considered during the design 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. The current implementation of the object manager is very suited to control persistent objects that have a small size, are frequently accessed, have no CPU-bound operations and are generated from large structural models.

3.4.1 Architecture

Figures 3.13a and 3.13b will be used to explain how the object manager is organized. Figure 3.13a shows how database (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 3.13b 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 persistent object.

3.4.2 Consistency Issues

The consistency of the persistent object and cache object is guaranteed by the computational model of object and actions. When a database 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 database object, the cache object behaves as a hint or as a proper cache. When locked for read, this read lock is released as soon as the object state has been read, guaranteeing that the passive database object can be accessed by another client and, consequently, the cache object might become stale. On the other hand, a write lock guarantees that the cache object will be always up to date in relation to the passive database object.

3.4.3 Implementation

The classes that implement the two components of the object manager, namely the object manager and object server (Figure 3.13), are respectively the classes PlexManager, MultiPlex and Plex. Each object manager, an instance of PlexManager, can control the activities of several object servers which are instances of the class MultiPlex (Class 3.9, line 1). For a given structural model (distributed database), the attribute **multiplexors** lists all hosts where database objects can be maintained. The attribute **active_multiplexors** (Class 3.9, line 2) holds only the subset of **multiplexors** that are active. Object managers have references to Arjuna's name server (Figure 3.9, line 3). The connection between an object manager and a name server is established during the construction of the object manager, that is, during the instantiation of a PlexManager. Finally, an object server is capable of interacting with an object store server to activate/deactivate database objects. Database objects are encapsulated by instances of the class Plex. Plex is derived from the class LockManager of Arjuna. Instances of Plex hold states of database objects.

Class 3.9 Class PlexManager.

```
class PlexManager {
private:
 (1)  MultiPlexes multiplexers;
 (2)  ActivePlexes active_multiplexers;
 (3)  NameServer* name_server;

 (4)  OpHistory* provide_multiplex(MultiPlexEntry*&, String host_name);
 (5)  MultiPlex* active_multiplex(Uid&);
 (6)  MultiPlex* active_multiplex(Uid&, ActivePlex*&);
 (7)  OpHistory* packInto_envelope(ObjectState& os, ClassPath& cp,
                                   ObjectState& envelope);
 (8)  OpHistory* unpackFrom_envelope(ObjectState& envelope,ObjectState& os,
                                     ClassPath& cp);

public:
 (9)  PlexManager(OpHistory*);
 (10) ~PlexManager();
 (11) OpHistory* create(Uid& signature, ObjectState& os, Uid& plex_uid,
                        ClassPath& class_path,
                        String host_name, String object_name);
 (12) OpHistory* read(Uid& signature, ObjectState& os, LockMode mode,
                      const String& class_name, const String& object_name,
                      ClassPath& guard_class_path, Uid& plex_uid,
                      String& host_name);
 (13) OpHistory* write(Uid& signature, ObjectState& os, Uid& plex_uid);
 (14) OpHistory* setlock(Uid& signature, Uid& plex_uid, LockMode mode);
 (15) OpHistory* destroy(Uid& signature, Uid& plex_uid);
 (16) OpHistory* discard(Uid& signature, Uid& plex_uid);
};
```

Methods **provide_multiplex** and **active_multiplex** (Class 3.9, lines 4-6) are used to activate multiplexes on demand. Methods **packInto_envelope** and **unpackFrom_envelope** (Class 3.9, lines 7,8) are responsible for packing/unpacking database object states into a buffer whose representation is suitable for transfer through a network.

The public interface of PlexManager, that is, the interface of the object manager, has the following atomic methods (Class 3.9, lines 9-16):

- **create**: Creates a database object with name **object_name** and state **os** at the object store specified by **host_name**. The **plex_uid** returned by **create** is the unique object identifier assigned to the database object during its

creation. It is used to register the object created with the name server. The parameter **class_path** is used by the object manager to guarantee that the object created has an state consistent with the state specified in the structural model. A **class_path** is a flattened representation of the structural submodel used in the creation of the database object.

- **read**: Retrieves in **os** the state of a database object from the distributed database. Parameters **class_name** and **object_name** are used to locate the object store where the database object is stored, using the name server. The object retrieved has a lock set on it, as specified by parameter **mode**. The values returned in parameters **signature**, **guard_class_path**, **plex_uid**, and **host_name** are used to check the correction of the object state retrieved.

- **write**: Writes the state of a database object into the object store from which it was last recovered.

- **setlock**: The method **setlock** is used to change the type of lock set on a database object.

- **destroy**: The method **destroy** removes a database object from an object store. The total removal of a database object is only possible when it is no longer referenced by other database objects. In the current version of Stabilis a reference count is maintained to guarantee the correction of the **destroy** operation.

- **discard**: If the database object specified by **plex_uid** is the last object managed by this active multiplex then the multiplex is deactivated after the removal of the database object.

The interface of class **MultiPlex** is similar to the interface of class **PlexManager**.

The interface of class **Plex** exports the basic atomic operations needed to manipulate a persistent database object. The class **Plex** acts as a capsule to Arjuna's object states (Class 3.10, line 1).

Class 3.10 Class Plex.

```
class Plex : public LockManager {
private:
 (1) ObjectState plex;

public:
 (2) Plex(ObjectState& os, Uid&, Birth, OpHistory* oph);
 (3) Plex(ObjectState& os, Uid&, Reincarnation, OpHistory* oph);
 (4) ~Plex();
 (5) OpHistory* read(ObjectState& os, LockMode mode = READ);
 (6) OpHistory* write(ObjectState& os);
 (7) OpHistory* erase();
 (8) OpHistory* lock(LockMode mode);
 (9) Boolean save_state(ObjectState& os, ObjectType t);
 (10) Boolean restore_state(ObjectState& os, ObjectType t);
 (11) const TypeName type() const;
};
```

- **Constructors**: As with all constructors of classes of Stabilis, Plex uses parameters **Birth** to indicate that a new instance of Plex is to be created and **Reincarnation** to indicate that a already existent instance of **Plex** is to be fetched (Class 3.10, lines 2,3).
- **read**: Reads the state of the database object **plex** into **os**. Use **mode** to determine the type of lock to set on the database object. The default action is to set a read lock on it.
- **write**: Writes an object state **os** into the database object **plex**.
- **erase**: Erases the contents of the database object **plex**.
- **lock**: Changes the type of lock set on **plex**.
- **Arjuna**: The methods shown in lines 9 to 11 (Class 3.10) are virtual methods defined by the interface of the class StateManager of Arjuna and implemented by Plex. These methods are used by Arjuna to manipulate persistent objects.

3.5 Queries

In Stabilis queries are used for two purposes: creation and retrieval of objects of classes belonging to the structural model of a distributed program. Stabilis allows the use of multiple classes and their key attributes in query expressions.

Class attributes, including relationships, are flagged as key attributes during the creation of the structural model. At this point, we must recall that queries always return sets of pips as their result.

3.5.1 Query Language

Query expressions are C++-compliant expressions passed as a parameter to constructors of classes, or to constructors of sets of objects. Identifiers and constants, terminal symbols in this grammar, follow the rules of formation dictated by C++. Backus-Naur Form (BNF) notation is used to specify the query language accepted by Stabilis. All nonterminal symbols are enclosed between "⟨" and "⟩", and all terminal symbols are in smallcap type. A syntactic unit enclosed between "{" and "}" may be instantiated zero or more times. During the instantiation of a syntactic production composed of several syntactic constructions separated by "|" only one of the productions listed is chosen and instantiated. An "ϵ" matches an empty syntactic production.

Query Constructors and Set Specification

⟨class_name⟩ ⟨variable_name⟩ (⟨query_expression⟩, ..., ⟨mode⟩, ...);

ObjectSet ⟨variable_name⟩ (⟨query_expression⟩, ..., Reincarnation, ...);

⟨mode⟩ → { Birth | Reincarnation | Provide }

The first production of the query language specifies the general syntax for using query expressions that retrieve/create one database object. The second production rule shows the syntax used to retrieve sets of objects.

The subset of C++ accepted as a query expression is defined by the following grammar:

⟨query_expression⟩ → ⟨class_name⟩ (⟨attribute_clause⟩) |
⟨query_expression⟩ ⟨set_operator⟩ ⟨query_expression⟩ |
(⟨query_expression⟩) | ϵ

The class name determines the generalization/specialization subgraph of the metalevel of Stabilis that is visited during the resolution of the associated attribute clause. Each node of this subgraph, a metaclass, indexes key attributes and relationships for its class.

⟨set_operator⟩ → ⟨intersection_operator⟩ | ⟨union_operator⟩
⟨intersection_operator⟩ → &&
⟨union_operator⟩ → ||

⟨class_name⟩ → IDENTIFIER
⟨variable_name⟩ → IDENTIFIER

⟨attribute_clause⟩ → ⟨term⟩ |
⟨attribute_clause⟩ ⟨logical_operator⟩ ⟨attribute_clause⟩ |
(⟨attribute_clause⟩)

⟨logical_operator⟩ → ⟨and_operator⟩ | ⟨or_operator⟩
⟨and_operator⟩ → &
⟨or_operator⟩ → |

⟨term⟩ → [⟨relationship_path⟩] ⟨attribute_name⟩ ⟨relational_operator⟩
{ ⟨value⟩ | ⟨attribute_name⟩ }

⟨relationhip_path⟩ → ⟨role_name⟩ ⟨path_operator⟩ |
⟨relationship_path⟩ ⟨role_name⟩ ⟨path_operator⟩

⟨path_operator⟩ → ::

⟨role_name⟩ → IDENTIFIER

⟨relational_operator⟩ → ⟨equal_operator⟩ | ⟨different_operator⟩ |
⟨great_operator⟩ | ⟨greater_equal_operator⟩ |
⟨less_operator⟩ | ⟨less_equal_operator⟩

⟨equal_operator⟩ → { == | = }
⟨different_operator⟩ → ! =
⟨great_operator⟩ → >
⟨less_operator⟩ → <
⟨greater_equal_operator⟩ → >=
⟨less_equal_operator⟩ → <=

The associativity and precedence of operators is summarized in Table 3.4 presents the full set of query language operators in order of precedence. 5L reads as "precedence level 5, left to right associativity." The higher the precedence level, the greater the precedence of the operator.

Level	Operator	Function
6L	::	relationship path operator
5L	==, !=	relational operators
5L	>, >=, <=, <	
5L	=	assignment operator
4L	&&	logical AND
3L	||	logical OR
2L	&	set intersection (pips)
1L	|	set union (pips)

Table 3.4: Operator precedence and associativity.

In its simplest form an attribute clause is an expression involving a key attribute and a value (constant). Examples of simple attribute clauses are "year > 1984", "title == 'Flowering Plants"', and "genus == 'Riviea"'. Simple attribute clauses can be combined using any of the logical operators. In the next Section a series of examples the use of the query language and shows some of the querying capabilities of Stabilis.

3.5.2 Examples of Queries

We use the structural model developed for DBib [36], a distributed bibliography database, to discuss the querying capabilities of Stabilis. DBib's structural model (Figure 3.14) is based on the bibliographic entries defined in [84]. This model has 23 classes, 18 are leaf classes. Leaf classes can be instantiated as database objects.

Retrieving objects: a simple query

Suppose that we want to retrieve from the bibliography database a technical report whose number is TR400. The C++ statement that specifies such query is:

```
TechReport tr = new TechReport("TechReport(number == 'TR400')",
                               REINCARNATION, ...);
```

We use this simple example to discuss partially the query processing algorithm implemented by Stabilis.

1. **Construction of an object and parsing of query expression:** C++ constructs objects starting from the base class downwards. In this particular case, the order of instantiation is Object, Reference, Unit, Exclude, and

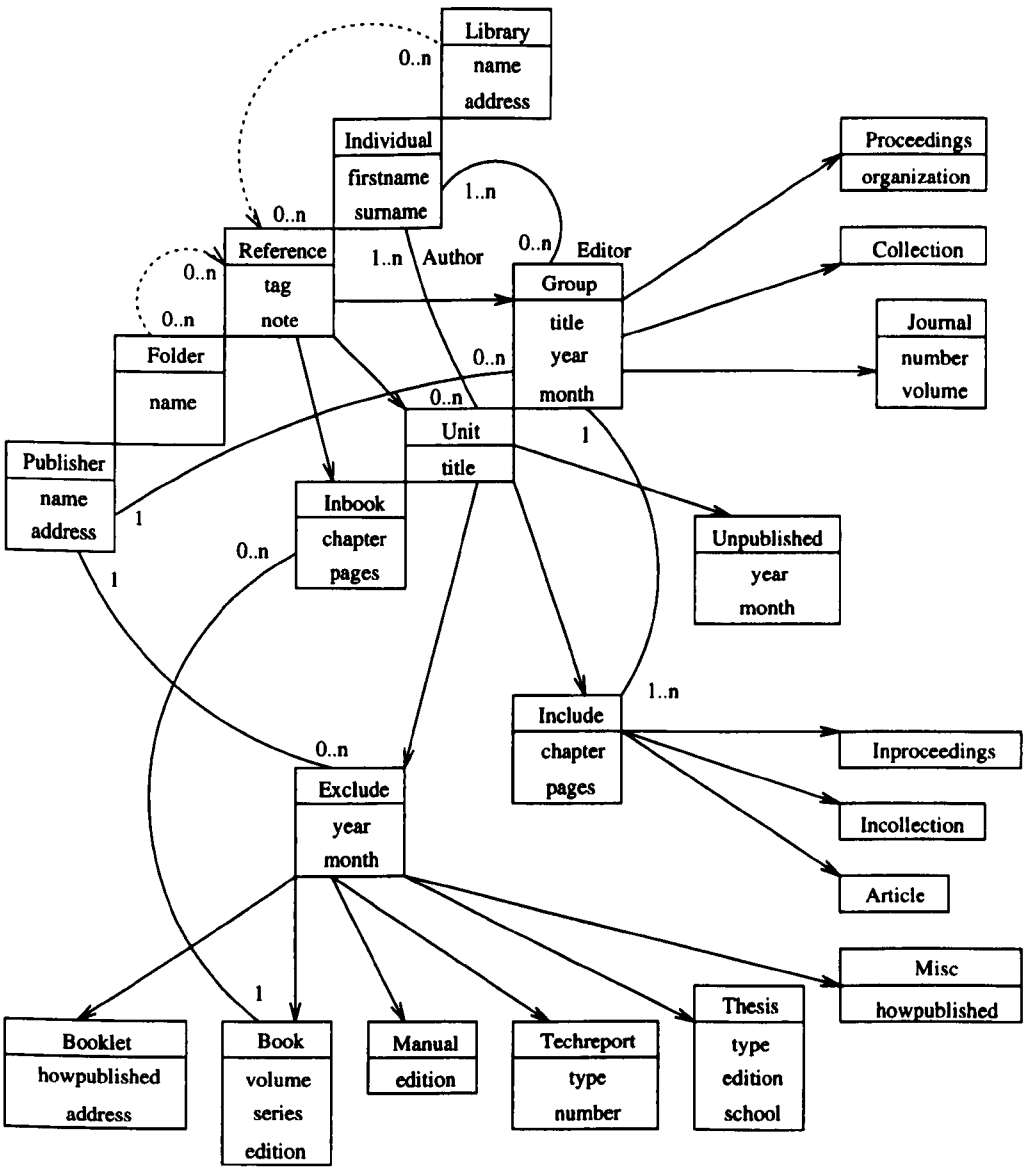

Figure 3.14: Structural model of DBib.

TechReport (Figure 3.14). During the construction process the text of the query expression is passed over to Object where it is parsed into a expression tree (Figure 3.15). Initially, a "hollow" object is built, that is, an object whose latest state has still to be retrieved from the database. If the query is successful, then the object's state is initialized with the state retrieved from the database. Otherwise, the variable **tr** is made **nil** and an operation history indicating the failure of the operation is returned.

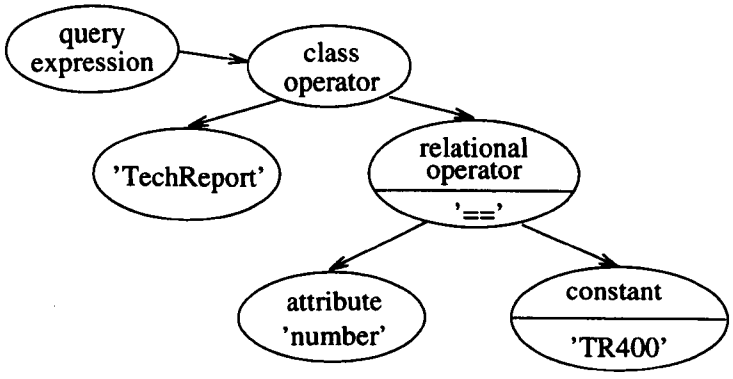

Figure 3.15: Parsed query expression (query tree).

2. **Resolution of class operator**: Object locates the metaclass contained in the class operator node of the query tree, in this case Article. The starting point for this search is the meta-metaclass Class. Next, Stabilis determines the initial graph that has to be visited during query resolution, that is, the initial set of metaclasses that have to be visited. It does that by marking superclasses and subclasses of the class kept in the class operator. In our example the graph marked contain nodes Reference, Unit, Exclude, and TechReport. Multiple inheritance is taken into account during the marking.

3. **Resolution of the attribute clause**: In this example the attribute clause is very simple; it contains only a single term formed by a relational operator, an **equal operator** and its two operands, a key attribute, **number**, and a string constant, "TR400".

   The resolution of this term involves visiting metaclasses, that is, indices, of all nodes of the graph determined in the previous step of the query algorithm. Each metaclass in the graph indexes their own key attributes. If the query specifies attribute clauses which contain terms where relationship paths and key attributes of related classes, then the graph to be visited is expanded accordingly during the resolution of the query.

4. **Activation of retrieved object**: Suppose that this query has returned a non-empty set of pips (objects). In this case, Stabilis randomly chooses one of the pips of the set and sends a message **grow()** to it. Execution of method **grow()** triggers execution of the method **read()** of the object manager. The object manager uses Arjuna's name server to locate the object. Finally, the state of the object is retrieved, that is, the object is activated and its handle is assigned to **tr**. If we had constructed an **ObjectSet tr** instead of only one object, then all objects returned would have been activated and made available to the programmer of the distributed program.

Creating and relating objects

In this example we create a book and its author, an instance of class **Individual**, and relate according to what is specified in the structural model of DBib (Figure 3.14). The execution of method **relate** involves the creation of a nested atomic action. The first subaction takes care of the update of the relationship tables of the objects, the second subaction carries out the update of the indices of metaclasses **Unit** and **Individual**. The indices for the relationship between a book and its author, an individual, are not implemented by metaclass **Book** but by class **Unit** (Figure 3.14).

```
(1) Book tex("Article(tag = 'Knuth84' & title = 'The TeXbook'
                & year = 1984)", ..., BIRTH, ...);
(2) Individual knuth("Individual(firstname = 'Donald'
                & surname = 'Knuth')", ..., BIRTH, ...);
(3) knuth.relate("Author", tex);
(4) tex.put("Book(note = 'ISBN 0-201-13448-9' & tag = 'DKnuth84')");
```

In line 1 an instance of **Book** is created. Note that all operators used are assignments instead of relational operators. During the parsing of the query text this expression is flagged as an assignment expression. If the query mode is "Birth" and the query expression has operators other then the assignment operator then the expression is rejected. The code of line 2 creates an instance of class **Individual**. In line 3 we relate them, note the use of the role "Author" (Figure 3.14) to specify the relationship. In the model of DBib (Figure 3.14) we can see that the relationship between **Individual** and **Book** has been inherited by **Book** from the class **Unit**. Queries can specify any attribute of any related superclass and attributes of the classes related to the related superclasses, and so forth. This type of query is called a navigational query because the programmer can specify paths of the structural model in his queries. Finally, in line 4 we can see the programmer calling method **put** of its object to update/initialize some of the attributes of object **tex**. The method **put** is defined in class **Object** and is, consequently, inherited by all classes defined in database models.

A simple navigational query

The query below is an example of a simple navigational query where the query key attributes belong to a class that is related to the class of the object being retrieved. All objects of DBib model can be sent a message that tells them to write their state in a pre-determined format into a file. In the code below, statement of line 2 makes object **tex** write its contents in BibTEXformat [84].

```
Book tex("Book(Author::surname == 'Knuth')", ..., REINCARNATION, ...);
tex.bibtex("tex.bib");
```

Another navigational query

Suppose we want to retrieve all books whose author is the editor of a journal with title "Algorithms" and that have been published after 1992 exclusive. To solve this query Stabilis has to visit metaclasses **Book**, **Exclude**, **Unit**, **Individual**, and **Group** (Figure 3.14).

```
ObjectSet books("Book(Author::Editor::title == 'Algorithms'
                & year > 1992)", ..., REINCARNATION, ...);
int cardinality = books.cardinality();
books.reset();
for (int i = 0; i < cardinality; i++)
    {
     Book* book = books.next();
     book.bibtex("alg.bib");
    }
```

Retrieving objects of different classes

We can specify a query that retrieves a set of objects belonging to different classes of a class hierarchy.

```
ObjectSet e("Exclude(Author::surname == 'Servantes')", ...,
REINCARNATION, ...);
int card = e.cardinality();
Thesis* t = 0;
for (int i = 0; i < card; i++)
    {
      Object* o = e.next();
      if (o→is_a("Thesis"))
        t = new Thesis(o→get_pip(), ..., REINCARNATION, ...);
      t.bibtex("thesis.bib");
```

```
    delete t; t = 0;
}
```

The query above retrieves objects of all six subclasses of **Exclude** (Figure 3.14). The program excerpt that follows the query iterates through the set of objects, selects those that are instances of **Thesis**, instantiates them and writes them into file **thesis.bib** in BibTEXformat.

Impedance mismatch

Since all objects retrieved by a query are C++ objects there is no impedance mismatch between the type systems of the query language and C++. Query expressions are specified using a subset of C++ expressions, they do not change any of the semantics of C++ and, therefore, do not cause any impedance mismatch between Stabilis and C++.

3.6 Programming Interface

We have discussed most of Stabilis' backstage workings, now it is time to discuss what the audience sees. Additionally, we discuss the role played by the maestro in helping the audience to enjoy the concert.

When the management environment is complete, then the audience will be able to specify structural models using a graphical tool. While that does not happen we need a database administrator, a maestro, to translate a structural model into a schema program. Once that has been done, then all the audience has to do is to query the database and find the structural model of his program. For example, the audience that is implementing DBib, a distributed bibliography database, has to do the following:

- Retrieve DBib's structural model: "Model dbib = Model("Model(name == 'Distributed Database')", ..., REINCARNATION, ...);"
- The statement "dbib.gen_code();" triggers the generation of header and code files for each class of DBib's structural model. For each class of the structural model four files are generated: two class definition file, with extension **.h and .X.h**, and two code files, with extensions **.cc and .X.cc**. The files with extensions **.h** and **.cc** contain all source code automatically generated by Stabilis. These two files are not supposed to be changed by users of Stabilis. The other two files generated are the files where extra definitions and the implementation of user-defined methods is kept.
- For each structural model, Stabilis also generates a simple interactive query interpreter. During tests this simple interactive query interpreter can be

used to validate a structural model. The user can submit queries that test the various relationships present in the model to make sure that the representation of the model, database schema, is correct.

Using Stabilis is relatively simple. First, the audience designs the structural model of the distributed program it wants to implement. Next, members of the audience write a schema program to represent this model as a database schema. Having represented the model as a schema the audience is ready to run Stabilis and make it generate the header and code files for all classes of the program's model. Finally, the audience writes its distributed program. The distributed program can create, delete, and update objects belonging to any structural model managed by Stabilis.

3.7 Adapting Stabilis

At the very beginning of this Chapter we affirmed that the use of reflection and object orientation made Stabilis adaptable and extensible. In this Section we present a brief overview of the processes that have to be followed to adapt and extend Stabilis.

A maestro is a person who has knowledge of both backstage and on-stage. We are going to see how maestros can add new types to Stabilis and adapt the query interpreter in order to attend requirements of different areas of computing.

3.7.1 Extending Stabilis with new Types

Suppose we want to add a type **Complex** number to Stabilis. The steps listed below tell what a maestro should do to carry out this task.

- Define class **Complex** as a subclass of **Attribute**.
- Implement in **Complex** all methods of the public interface of **Attribute**. This task entails implementation of the constructors that accept query expressions, implementation of the methods used by the object manager and Arjuna to manipulate object states, a print method, and the methods used by the protocols of **Model** to generate C++ code automatically.
- Derive a class **ComplexConstant** from **Constant** and implement the public functions of the base class. The task is very similar to the one carried out in the previous step, except that now the maestro has to write the code of all relational operators used in the query language of Stabilis. For example, he has to implement a method to test if two complex numbers are equal values.

- In the last step, the maestro of Stabilis has to change the query interpreter to make possible the parsing and interpretation of complex values. He has to extend Stabilis' yacc and lex parsers to include the treatment of values of type Complex.

- All this new code is then compiled and the previous version of Stabilis is replaced by this new one. These operations do not involve any disruption to the Stabilis applications that are executing. They are only going to use the new code in their next execution.

- Now, we have to create the metaclasses for Complex and ComplexConstant. An incremental schema program is created and executed to initialize all indexing structures.

As Arjuna, Stabilis explores inheritance to allow programmers, i.e., maestros, to adapt it to the needs of new application areas.

3.7.2 Adapting the Query Interpreter

The query interpreter of Stabilis is composed of three main modules: a lexical analyzer, implemented using GNU's flex, a parser, implemented using GNU's bison, and a C++ class hierarchy that defines all operators used in the query language.

Suppose that a team of electronic engineers has added new types to Stabilis. For example, they can create types that represent the components used in the routing of printed-circuit boards. What the team of engineers want now is to change the semantics of certain of the query operators of Stabilis' query language to facilitate the writing of their programs. For example, they want the class operator :: to perform extra checks during the traversing of metaclasses of certain electronic components. In the current implementation of the query resolution algorithms the set intersection and union operators use only the UIDs of objects in the creation of intersections and unions. The team of electronic engineers might want to change the semantics of the set intersection and union operators to take into account the value of some attribute of objects.

In summary, it is possible to alter the implementation of operators of the query language using inheritance and operator polymorphism. Using these techniques a maestro can implement versions of query interpreters that behave differently in function of the types of application objects they are retrieving from the database. During query resolution the operator is trapped, the types of operands are checked and a certain operator behaviour is selected in function of the operand's types. Thus, Stabilis can be characterized as having a reflective architecture with limited reflective capabilities [97].

3.8 Conclusions

This Chapter has described the design and implementation of Stabilis, an object-oriented database programming tool programmed on top of Arjuna.

Our discussion of the architecture of Stabilis was carried out in three stages. In the first, we have defined the structural model supported by Stabilis. A comparison of the object-oriented models defined by C++ and Stabilis led us to conclude that we had to extend the model supported by C++ if we wanted to represent information about distributed programs in our system. The structural model supported by Stabilis can represent the concepts of classes and relationships (generalization/specialization, association, aggregation). The structural model is used in our management system to explicitly represent the structure of a distributed program. The architecture of Stabilis was defined using the structural model of Stabilis; as a consequence, a recursive architecture is created. The architecture of the system has three levels of information representation that are recursively generated: objects, metaobjects and meta-metaobjects.

In the second stage, we have shown how the layers of the architecture are created through an example where we instantiated a structural model. This example gave us a clear view of how the structural model of Stabilis moulds the architecture of the management system. At this point we introduced the orchestra metaphor and used it to convey an integrated perspective of the various architectural layers of the management system.

The subject of the third stage of the description of the architecture of Stabilis addresses the implementation of the indexing structures of the system. We have explored every detail of the process for instantiation of the metamodel of Stabilis and shown how the various problems of circularity have been solved. At the end of this stage, we have acquired the knowledge necessary to understand how queries are resolved.

Before discussing query resolution, we described the functionality of the object manager. This model implements the interface between Stabilis and Arjuna; it is responsible for the manipulation of objects, including caching and multiplexing.

Queries in Stabilis are used for two purposes: creation and retrieval of objects belonging to an object model. Stabilis allows the use of multiple classes and their attributes in query expressions. We have described the query language supported by Stabilis and have shown how the algorithms for query resolution operate through various query examples.

In this Chapter, we have discussed the programming interface of the management system, showing that it has not introduced any change to the programming language adopted (C++). Finally, we have shown how a programmer can add new types to Stabilis and how he can adapt Stabilis' query interpreter to his own needs.

Chapter 4

Vigil

We have argued that there are two basic modes of perceiving a distributed program: a static, or structural mode, and a dynamic, or control mode. In the previous Chapter we have concentrated on the issues related to management of structural information. In this Chapter we address management of control, or dynamic, information. We discuss the design and implementation of Vigil, a tool that processes *dynamic* information concerning an application program and allows the enforcement of changes upon it. Thus, the use of Vigil allows a distributed program to adapt to changes occurring in the environment where it is being executed.

Implementors of adaptable distributed programs are concerned with representing structural and control aspects of programs explicitly in order to simplify the management task. Stabilis and Vigil were designed to help implementors cope with the management of these explicit representations. Further simplification of the management task can be achieved by keeping the implementation of management policies apart from the implementation of the functional aspects of the distributed program. Viewing distributed programs as reactive systems can help implementors to achieve such separation of implementation.

4.1 Reactive Systems

A *transformational program* is the more conventional type of program, whose role is to produce a final result at the end of a terminating computation. A transformational program can be considered as a function from an initial state to a final state or a final result. A *reactive program* is a program whose role is to maintain an ongoing interaction with its environment rather than to compute some final value on termination. The family of reactive programs includes most classes of programs whose correct and reliable construction is considered to be difficult, including process control programs, operating systems, and management programs [67]. Management programs[1] are a good example of programs required to maintain a continuous interaction with the application programs they control.

The notions of concurrency and reactivity are closely related. In any program containing concurrent objects it is possible to study and analyze each object as a reactive program. This is because, from the point of view of each object, the rest of the distributed program can be viewed as an environment that continuously interacts with the object. Thus, we may have a program that in its entirety has a transformational role, i.e., it is expected to terminate with a final result. Nevertheless, because it is constructed from concurrent objects, it should be analyzed as a reactive program. This is exactly what occurs with programs that

[1] From now on the terms *management program* and *control program* are used as synonyms.

are designed to be reconfigurable. The application program may be a program that terminates and produces a result. Yet, because it is executed concurrently with the management program both have to be analyzed as reactive programs. To the management program the application program represents the environment, and vice versa. The same reasoning is recursively applicable to the objects that compose the programs and to the system were the program is executing.

4.1.1 Distributed Programs seen as Reactive Programs

From the point of view of the management system, a distributed program is seen as the superposition of two reactive programs: an application program and a management program.

In object-oriented systems, the designer of a distributed program usually conceives a structural model and a *control model* of the program before he implements the program. Usually, these models are directly translated into programs using ad-hoc implementation strategies, but in Stabilis and Vigil these models become instead database schemas. Later, these database schemas are used to generate part of the application and management programs. They are also used during the management of the program. A control program is a state machine that is executed by Vigil. At runtime, the objects of a control program constantly monitor the objects of an application program and allow them to adapt to changes occurring in their environment according to management policies encoded in the control program. Next, we see how an application program and a control program interact when viewed as reactive programs, such view is based on locality of control, that is, we want each application object of a distributed program to have all its control mechanisms encapsulated by its corresponding control object.

Application Program

An application program has the following form:

$$A :: [A_1 \| \ldots \| A_n]$$

where $A_1, \ldots, A_n$, $n \geq 1$, are application objects and A is the program obtained from their interactions, as dictated by the algorithm that governs the application program. The symbol "::" means "composed of", and the symbol "$\|$" means "concurrent execution"; square brackets are used to delimit groups of objects. The application program has a set of external states $E_A = (e_{a_1}, \ldots, e_{a_l})$, $l \geq 1$ which are maintained by the application objects and are made available to the management program for reference and modification only through the interface of the program. Each object of the application program is structured using the object-oriented action-based programming paradigm provided by Arjuna.

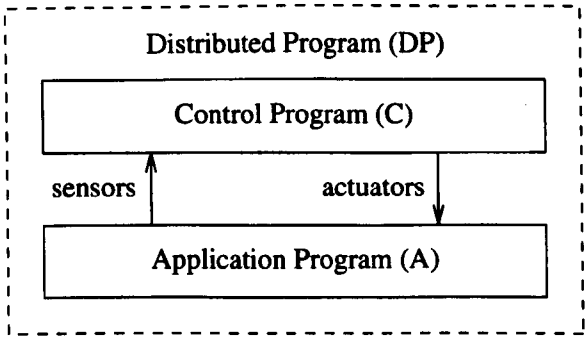

Figure 4.1: Distributed program viewed as a reactive system.

Control Program

The control program has the following form:

$$C :: [C_1 \| \ldots \| C_n]$$

where $C_1, \ldots, C_n$, $n \geq 1$, are control objects and C is the control program obtained from their interaction as dictated by the algorithm that governs the control program, which is implemented as a finite state machine. In the management system a finite state machine is implemented as a list of guarded commands; each guarded command has two parts—a condition and an action. The action part of a guarded command is executed whenever the condition part is satisfied. Guarded commands have access to the set of external states E mentioned above through sensors and actuators. Sensors and actuators are methods implemented at the interface of the application program A. Sensors are used to implement the condition part of a guarded command, actuators are used to implement its action part. Each object of the control program is also implemented in compliance with the object-oriented action-based programming paradigm, that is, sensors and actuators are atomic methods.

Distributed Program

A distributed program DP is a superposition of A and C, where A is the underlying program and C is the superposed program. It has the form:

$$DP :: [DP_1 \| \ldots \| DP_n]$$

where each $DP_i \in \{DP_1, \ldots, DP_n\}$, $n \geq 1$ is formed by the superposition of A_i and C_i (Figure 4.1).

The composition of the environment determines its communication pattern. Let C_i denote the ith control object; let A_i denote the corresponding application object. Each application object A_i is permitted to execute methods of any other application object but not of any control object. Each control object C_i is allowed to execute methods of any application object A_i but not of control objects C_j, $i \neq j, 1 \leq j \leq n$. Communication among control objects is indirect and accomplished via shared application objects. From the specification of programs C and A, we have that all methods of the distributed program DP are atomic. Additionally, each application object must have been superposed by a management object in order for the application object to be managed.

Objects of the distributed reactive program DP belong either to the application program A or to the control program C. Programmers are encouraged to implement algorithms dealing with functional aspects of DP in A and algorithms dealing with management aspects of DP in C. The structural and control models make *visible*, or *explicit*, the structure of the distributed program DP. Once the structure of the program is visible then it is possible to reason about the management of its parts, including its dynamic reconfiguration.

During the execution of DP, the application program A passes through a sequence of states. In response to a state transition the management program C may act upon A, which in turn may trigger another state transition. The concept of controlled state transitions leads to the definition of an abstract model for reactive programs called transition system, or finite state machine.

We have chosen a extension of state machines based on Statecharts [67] to model the control aspects of distributed programs because:

- Most of the object-oriented modelling literature [33, pages 167-174][55, pages 60-97][120, pages 84-115][123, 33-65] seems to agree that state machines are adequate to the description of control aspects of object-oriented programs.

- Extended state machines, unlike traditional state machines, avoid an exponential explosion of states and transitions through structure. Traditional state machines are "flat"; in contrast, extended state machines allow a hierarchical, modular and well-structured description of systems [47, 67, 68].

- Programs are implemented using data structures and methods. It has usually been easier to modify data structures of an existing program than to modify the structure of methods that implement the program. Modification of the structure of such a program involves modification of one or more methods, a task not easily accomplished. In contrast, when the algorithm of a program is represented as a finite state machine then its implementation consists simply of a set of guarded commands where each guarded command operate in complete independence of each other. Therefore, implementations of finite

state machines can be modified very easily by deleting productions or by inserting new ones. Thus, extended finite state machines are employed in the implementation of control programs.

In Vigil, control models are mainly used to capture and represent concerns related to the management of distributed programs. With this perspective of distributed programs in sight, we define the architecture of Vigil.

4.2 The Architecture of Vigil

We present the architecture of Vigil in steps, adopting a structure of dissertation very similar to that used to describe the architecture of Stabilis. In the first step we define the control model supported by Vigil, using concepts adopted by Harel [68, 70] and Rumbaugh [120] to define transition systems; we emphasize the use of transition systems for management purposes. The second step describes the implementation of Vigil using an example. Vigil's implementation is based on the representation of states, guarded actions, and transition systems as database objects. Consequently, the description of Vigil's architecture can be simplified because we have already seen how to represent structural models using Stabilis. Backstage, at meta-level, the active component of Vigil is a scheduler of guarded commands, i.e., a state machine engine. Vigil's scheduler is similar to a rule processing module of an active database management system. The last step discusses the use of Stabilis and Vigil; we briefly discuss the steps followed by implementors of distributed programs that use the management system.

4.2.1 Control Model

One of the difficulties with the design of distributed programs comes from the treatment of temporal relationships among objects. To avoid such difficulties—at least temporarily—we tend to abstract away temporal relationships by first examining the static structure of the objects that form the application. Once we are familiar with the structural model of the distributed program we can return and examine changes to the objects and their relationships over time.

Although the concept of transition system is well-known we have decided to include a definition of transition system at this stage mainly to make clear where these concepts stand in relation to the object-oriented concepts we have introduced so far.

States

The values of the attributes, including relationships, held by an object are called its *internal state*, denoted by a tuple $I = (i_1, \cdots, i_n)$ where each i_i is the current

value of one attribute of the object and n is the number of attributes of the object. The *internal state space of an attribute* is the set of values it can assume; the *internal state space of an object* is the Cartesian product of the internal state spaces of its attributes. For example, if an object has attributes A and B, where A has internal state space $\{T, F\}$ and B internal state space $\{-1, 0, 1\}$, then the internal state space of this object is $\{(T,-1),(T,0),(T,1),(F,-1),(F,0),(F,1)\}$.

During program execution, objects stimulate each other, resulting in a series of changes to their internal states. The response to an stimulus depends on the state of the object receiving it, and can include a change of state and/or the sending of another stimulus to the original object or to a third object. In this scenario, internal states are not of much use from a modelling perspective since any change in the value of an attribute results in a "different" state and this leads potentially to a infinite set of internal states. Therefore, we introduce the concept of external state. The external behaviour of an object can be described in terms of a finite set of *external states.* To our purposes, an *external state* is a set of values obtained through the application of an abstraction function that maps internal states into the values of an external state. For example, an object has an integer attribute, say x, whose state space is defined by the integer interval [-100, 100]. An external state for x might be defined as the set of values {NEGATIVE, ZERO, POSITIVE}. An *external state space* is a tuple of external states such that any two members of the tuple are disjoint. Objects of a distributed program perceive each other only through discrete transitions from one external state to another. From now on the word *state* stands for *external state* and the term *state space* stands for *external state space*, unless specified otherwise.

In summary, a *state* is an abstraction of the attribute values of an object. Sets of values of attributes are grouped together into a state according to properties that affect the gross behaviour of the object. In defining states, we ignore those attributes that do not affect the external behaviour of the object, and we lump into a single state all combinations of attribute values that have the same response to stimuli. A programmer defines stimuli and states depending on the level of abstraction he is using to model his distributed program. States are used by the implementor of a management program to guarantee that changes only take place when the objects involved in the change are in a consistent state. What is important is that the relevant abstraction of the dynamic behaviour of a distributed program is captured and made *explicit* by representing it as a control model.

Events

An stimulus from one object to another is an *event.* An event is something that happens at a point in time, such as *user depresses left button of mouse.* An event

has no duration. Naturally, nothing is really instantaneous; an event is simply an occurrence that is fast compared to the granularity of the time scale of a given abstraction. In the object and action model of computation an *event* is an execution of an object's atomic method. The data values conveyed by an event are its *attributes*, like the data values held by an object. As we have discussed in the Section on reactive programs (Section 4.1.1), these atomic methods are logically grouped into two families: sensors and actuators. Sensors are used to probe the application's state and actuators are used to make objects compute new values that may in turn change the state of the application.

Guards

A *condition* is a Boolean predicate on states. Conditions can be used as *guards* on transitions. A guarded transition is traversed when its event occurs, but only if the guard condition is true. For example, "when a radio is turned on (event), if the volume is too high (condition), then turn the volume down (action)." In the object-oriented paradigm only the execution of methods can change a condition. We call the condition existing prior to the execution of a method a *precondition*, and the condition that exists after the execution of a method a *postcondition*. In our notation, a guarded condition on a transition is shown as a Boolean expression in square brackets following the event signature. In Vigil, sensors are used in the specification of conditions.

Actions

An *action* is an instantaneous operation. An action is associated with an event. For example, *disconnect phone line* might be an action in response to an *on-hook* event for a phone line. A real-world operation is not really instantaneous, but modelling it as an action indicates that we do not care about its internal structure for control, or management, purposes. If we do care, then an operation should be modelled as a series of actions, with a starting event, ending event, and possibly some intermediate events. Actuators are used to implement actions in Vigil.

State Diagrams

A state diagram relates events and states. When an event is received, the next state depends on the current state as well as on the event. A *state diagram* is a directed graph whose nodes are states and whose directed edges are transitions labelled by event names. A state is drawn as a rounded box containing an optional name. A transition is drawn as an arrow from the receiving state to the target state; the label on the arrow has the name of the event causing the transition. All the transitions leaving a state must correspond to different events. Figure 4.2

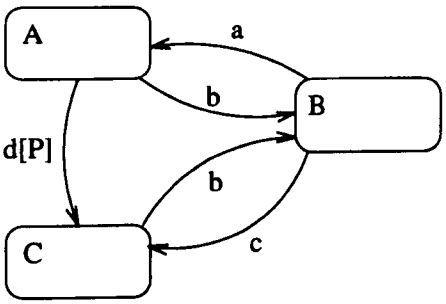

Figure 4.2: Simple state diagram.

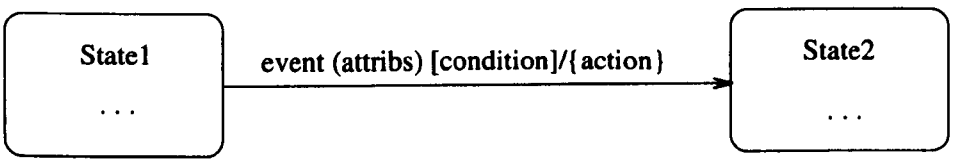

Figure 4.3: Flat state diagrams: summary of notation.

shows a simple state diagram containing three states A, B, and C and events a, b, c, and d. In Figure 4.2 event d occurring in state A transfers the system to state C, but only if condition P holds at the instant of the occurrence.

The state diagram specifies the state sequence caused by an event sequence. If an object is in a state and an event labelling one of its transitions occurs, the object enters the state on the target end of the transition. The transition is said to *fire*. If more than one transition leaves a state, then the first event to occur causes the corresponding transition to fire. If an event occurs that has no transition leaving the current state, then the event is ignored. A sequence of events corresponds to a path through the graph. A *control model*, or transition system, is a collection of state diagrams that interact with each other via shared events. Figure 4.3 summarizes the notation introduced until now for state diagrams.

Nested State Diagrams

State diagrams can be structured to permit concise descriptions of complex systems. The ways of structuring state machines are similar to the ways of structuring classes. In the structural model, the generalization/specialization relationship is used to model classes and subclasses, in the control model, generalization/specialization allows states and events to be arranged into hierarchies with inheritance of common structure and behaviour, similar to inheritance of

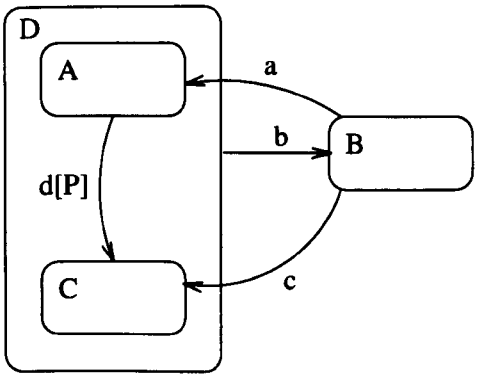

Figure 4.4: State generalization.

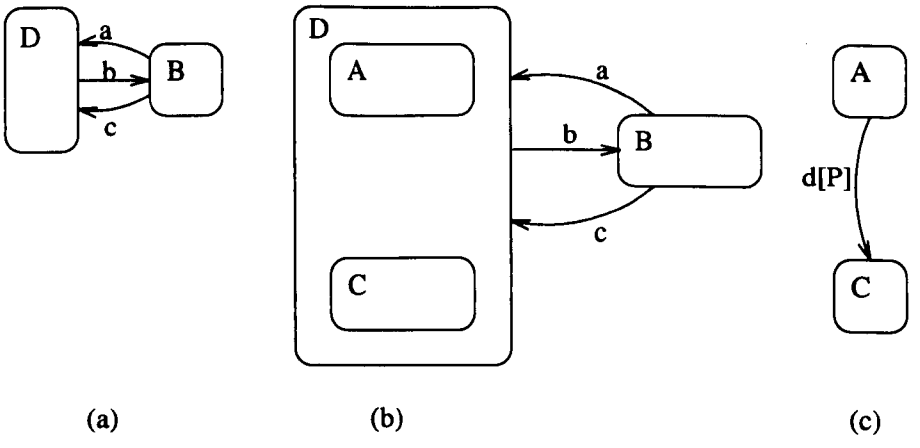

Figure 4.5: State generalization/specialization.

attributes in classes. Aggregation allows a state to be broken into independent components, with limited interaction among them, similar to the aggregation relationship in the structural model. Aggregation is used to describe state machines that are executed concurrently.

Generalization/Specialization Let us observe Figure 4.2 again. Since event b takes the system from either A or C to B we can *cluster* A and C into a new superstate D and replace two b edges by one, as in Figure 4.4. The semantics of D is then the *exclusive-or* of A and C; i.e., to be in state D one must be either in A or in C, and not in both. Thus, D is really an abstraction of A and C; a generalization. Figure 4.4 might also be approached from a different angle: first we might have decided upon the situation of Figure 4.5a, and then state D could

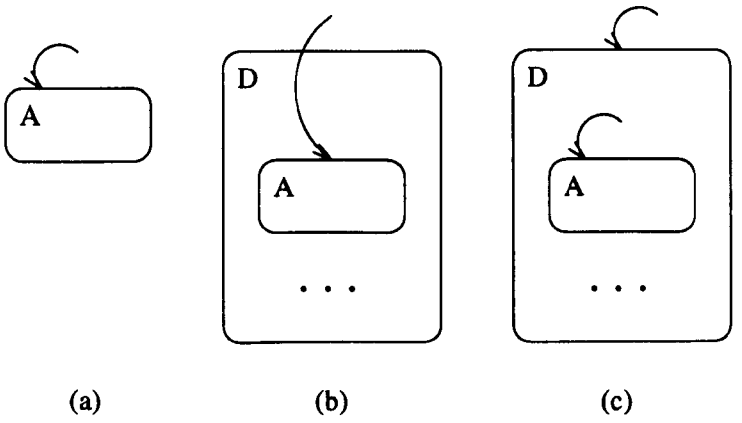

Figure 4.6: Default states.

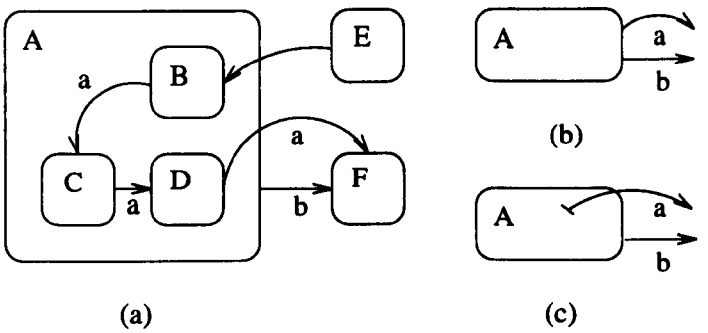

Figure 4.7: Exiting states.

have been *refined*, that is, *specialized*, to consist of A and C, yielding Figure 4.5b. Having made this refinement, however, the incoming a and c edges become underspecified, as they do not say which of A or C is to be entered. Extending them to point directly to A and C, respectively, solves the underspecification problem, and if the d transition within D is added, we obtain Figure 4.4.

Depending on the level of abstraction we are working at we can consider states as being atomic or not. For example, if we do not consider D atomic then we look "inside" it and perceive what is represented in Figure 4.5c. If we consider D atomic then the substates are abstracted away and we reason using only the state diagram of Figure 4.5a.

Suppose now that as far as the "outside" world is concerned A is the *default state* among A, B, and C, in the sense that if asked to enter the A, B, C group of states the system is to enter A unless otherwise specified. There are different ways to denote default states. For a state diagram as the one shown in Figure 4.2,

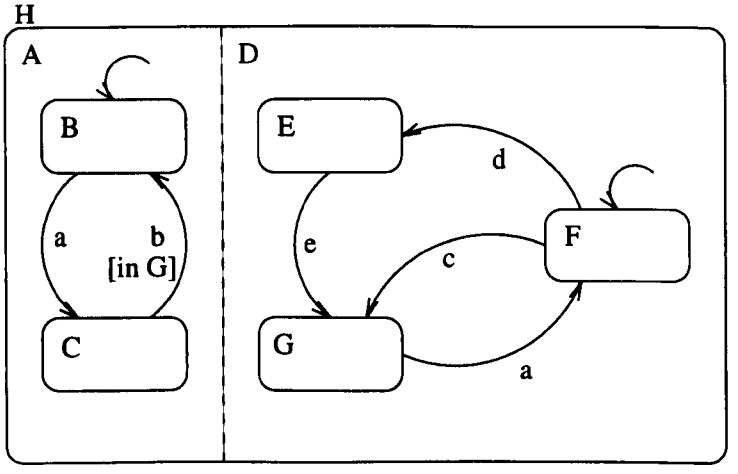

Figure 4.8: State aggregation.

a small arrow can indicate it directly (Figure 4.6a). For Figure 4.4 it is possible to use the direct notation of Figure 4.6b, or alternatively, the two-step notation of Figure 4.6c, which says that D is default between D and B, and A is the default between A and C. Default arrows are analogous to the start states of finite-state automata.

Figure 4.7a shows a non-atomic view of state A. There are to edges exiting A. The exiting transition from substate D of A to F specifies that A should be exited only when event a occurs in substate D. On the other hand, the transition from A to F specifies that A should be exited starting from any of substates B, C, D when event b occurs. Figures 4.7b and c show respectively the wrong and right ways of representing these two exit transitions when we consider A an atomic state.

Aggregation We have introduced the concept of generalization/specialization of states and some related notation. Now, we define the aggregation of states, capturing the property that, being in a aggregated state, the system must be in *all* substates of the aggregate. We represent aggregated states by splitting the aggregate into its components using dashed lines.

Figure 4.8 shows a state H consisting of aggregated components A and D, with the property that being in H entails being in some combination of B or C with E, F, or G. We say that H is the aggregation (orthogonal product) of A and D. Entering H from the outside, in the absence of any additional information, is actually entering the combination (B,F) be the default edges. If event a occurs, then it transfers B to C and F to G *simultaneously*, resulting in the new combined

state (C,G). This illustrates a certain kind of *synchronization*: a single event causing two simultaneous happenings. If, on the other hand, d occurs at (B,F) it affects only the D component, resulting in aggregated state (B,E). This, in turn, illustrates a certain kind of *independence*, since the transition is the same whether the system is in B or in C in its A component. Both behaviours are part of the *orthogonality* of A and D, that is, a property of the aggregate state H.

An application of aggregation is in splitting a state in accordance with the independent subsystems that compose it.

Structural Model and Control Model

A control model specifies the behaviour of a class of objects as state machines. In this Section we evaluate what happens to the state machines associated to a class when classes are organized in hierarchies. First, we consider what happens to the state machines when only single inheritance is used to form class hierarchies. Next, we assess the inheritance of state machines when multiple inheritance is used in a class hierarchy.

Single Inheritance The assumption of strict inheritance as the norm for the generalization/specialization relationship (Section 3.1.1) allows us to derive a set of rules about the inheritance of transition systems associated to each class [104].

Rule 1 *A subclass cannot delete a state of any of its parent classes.* For example, suppose that class, say A, has a state space $S_A = \{s_{A_1}, \cdots, s_{A_n}\}$ where n is the number of states of A, then the state space of class B, subclass of A, is given by $S_B = \{s_{A_1}, \cdots, s_{A_n}, s_{B_1}, \cdots, s_{B_m}\}$ where m is the number of states introduced in class B that were not in class A. It is possible to define a function that associates states of B to states of A. Thus, it is possible to modify states in the sense that their representation may be remapped into other states, provided the external behaviour of the methods involved does not change. This rule is a consequence of the requirement that the method invariant hold for every method of the classes involved in the strict inheritance.

Rule 2 *Any new state introduced in a subclass is wholly contained in an existing state of one of the parent classes.* There may be design factors that lead to a further refining of the states of B. In such case, one of two situations is possible, (i) the additional refinement does not involve changing any of the original states of the state space of A or, (ii) the refinement of the state space of B is based only on internal states that were not used in the mappings used to create the state space of A. In the former case, the result is the creation of an state in B that is substate of an state of A. The latter case results

in a set of states that are independent of and disjoint from the original states of A. In this case an instance of B can be thought of as being in one state from each set concurrently. A third possibility is rejected as invalid. This possibility involves the creation of states for A and B that overlap. Such situation is not allowed if the class hierarchy has to comply with the requirement of strict inheritance.

Rule 3 *A subclass may not delete a transition from the state machine of one of its parent classes.* If a transition were deleted then the method that implements that transition must have been modified to loosen its postcondition, a violation of the method invariant. However, it is possible for a subclass to override an inherited method and change its implementation provided that its postcondition remain as they were or become more restricted than those of the overridden method.

Multiple Inheritance The use of multiple inheritance as the basis for a new class adds complexity at both the implementation and design levels. At the implementation level, some mechanism must exist to resolve any conflicts that arise from the merging of the state machines. For example, names of external states might be duplicated. Techniques to recognize and solve some of these problems have already been developed during the implementation of object-oriented languages and state machine interpreters [69]

At design level, the requirement a class hierarchy has to comply with is that the classes being used as parent classes must represent independent concepts [104]. The independence of the concepts implies disjoint representations.

The representation of the parent classes define state machines for their original classes. The independence of concepts and the disjointness of the representations imply that the state machines of the separate classes do not have any overlap. The class that results from the inheritance of the multiple parent classes should have a state machine that contains the machines of the parent classes simultaneously. The subclass then is in a state of each of its parent classes concurrently.

Finally, we can summarize the control model in a set of principles.

C_1 A *state* describes an external state of an object. Every object has an external state space associated to it that can be used to control some of the activities of the object.

C_2 An *event* is something that happens at a point in time. In the object and action model adopted by the management system, events are methods implemented as atomic methods.

C_3 A *guard* is a boolean atomic method. Values returned by guards determine if an object is in a certain state and are used to decide whether or not a certain

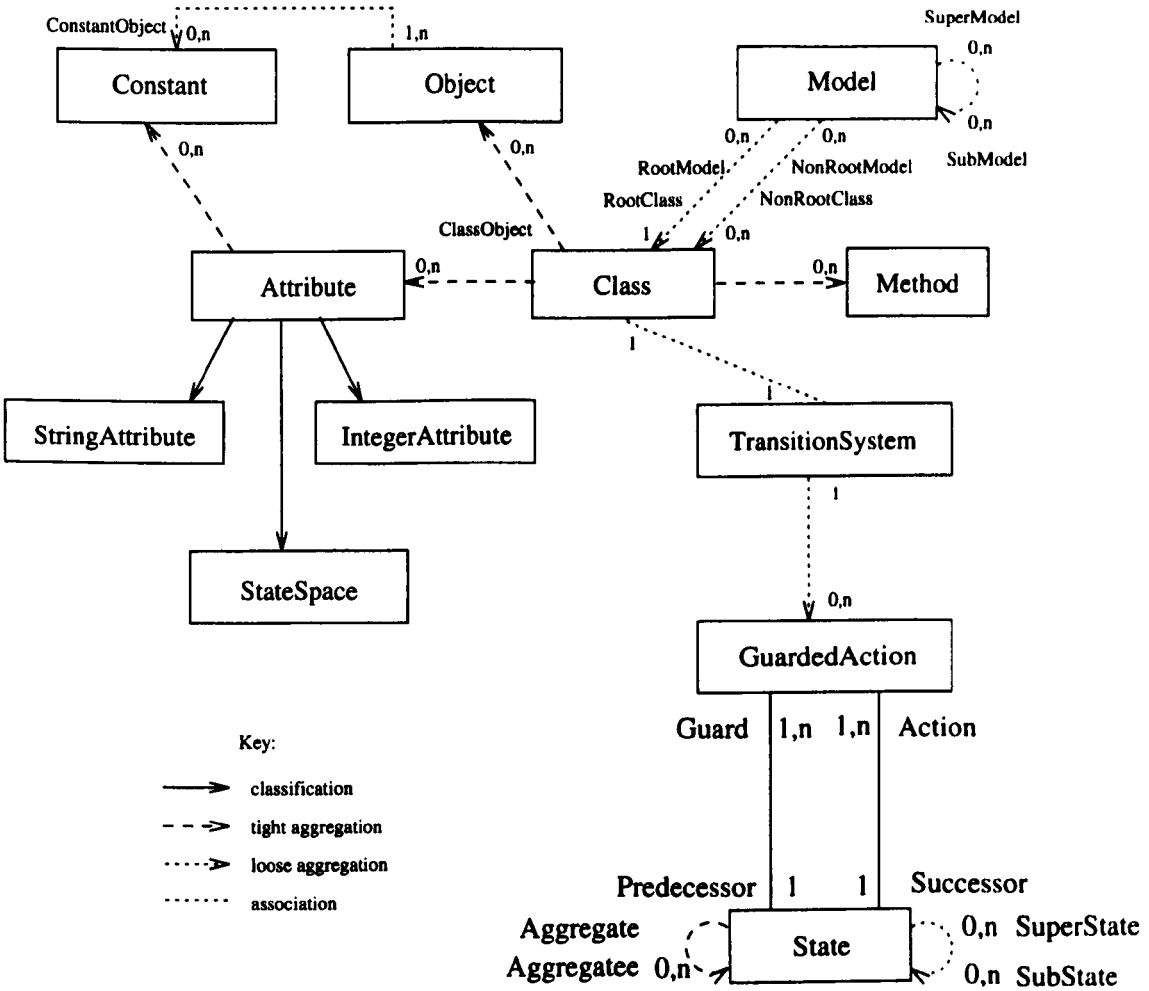

Figure 4.9: Vigil's structural model.

transition should fire when an event happens. Guards are implemented as atomic methods.

C_4 *Actions* are instantaneous atomic operations associated to events.

C_5 A change of state caused by an event is a *transition.* When a transition fires it traverses from a *predecessor state* to a *successor state.* In the management system, a transition is represented by a triple. The first and second terms of the triple are an event and a guard, respectively. They are specified as a Boolean predicate that involve results returned by sensors. The third term of the triple is an action that is specified as an atomic action involving the execution of actuators. In the management system this triple is termed *guarded*

action. Nested atomic actions are used to implement guarded actions. In Vigil, the class GuardedAction is used to represent guarded actions. Perhaps, a better name for this class could be Transition but we have decided to keep the name that we used in the implementation of the system.

C_6 A state can be defined as a substate of another state; substates and superstates are defined using the generalization/specialization relationship for states. In Vigil, the class State represents the most general state an object can have associated to it. All states in a control model are, directly or indirectly, represented as subclasses of the class State.

C_7 A state can be defined as an aggregation of states; aggregate and component states are defined using the aggregation relationship for states.

C_8 A transition system is a collection of guarded actions and states. Every manageable class has a transition system associated to it. In Vigil, the class TransitionSystem is used to represent a transition system.

4.3 Implementation

We can turn our attention to the realization of the management system that corresponds to the abstract model described in the previous Sections. Figure 4.9 shows the structural model that describes the relation maintained between the various components of Vigil and Stabilis. Vigil's structural model has been designed in function of principles C_1 to C_8. Principle C_8 states that each manageable class has a transition system associated to it. Consequently, we have modelled classes as having a transition system associated to them. From principle C_5 we have the associations between guarded actions and states. Principles C_6 and C_7 were used to model the relationships between states that we see represented in the model (Figure 4.9, class State).

The representation of the structural model of Vigil is carried out in the same way as the representation of any other model (database schema) in Stabilis. A schema program is created to instantiate metaclasses for each of the classes of the model. The execution of the schema program creates the backstage structures that are needed to index objects that represent control models, i.e., metaobjects. During the booting up of Stabilis the metaclasses of the structural model of Vigil are created together with the meta-metaclasses and metaclasses of Stabilis.

The architecture of Vigil is essentially identical to the architecture of Stabilis with a difference in their implementation: transition systems are not stored as database objects. Figure 4.10 shows where the implementation of both systems stand in relation to each other, when storage of information in database

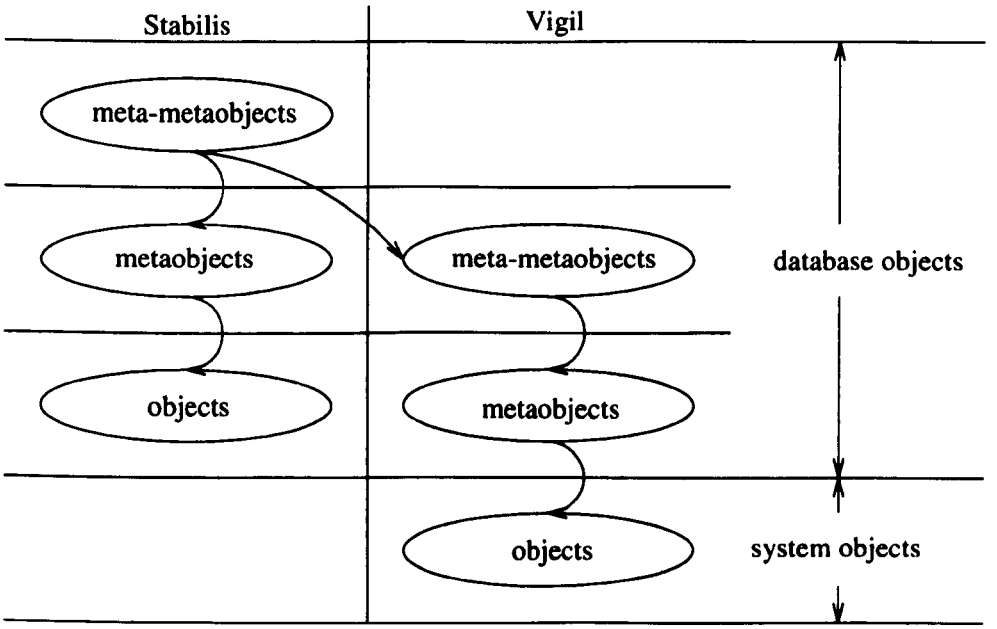

Figure 4.10: (Meta)object layers in Stabilis and Vigil.

objects is considered. For example, metaobjects of Stabilis are used to represent meta-metaobjects of Vigil (Figure 4.10, second row from top to bottom). We can observe that all information related to Stabilis components is stored in database objects. This is not the case with the information related to Vigil. Meta-metainformation and metainformation regarding transition systems is stored as database objects but information, that is, a transition system, is not. They are maintained as executable programs in the file system of the host operating system. This restriction in the implementation of the architecture does not represent a major problem for users as it is possible to implement a version-checking algorithm to make sure that application objects and control objects are properly matched, that is, that a user is not executing mismatched versions of control and application programs when it runs his distributed program. Ideally, the object store of the management system should be able to handle executable code; this would allow Arjuna, Stabilis and Vigil to have complete control over objects and, therefore, would make the system even more flexible and easy to use.

4.3.1 On Circularity

If we had followed strict object-orientation principles, then states, or state spaces, could have been considered *attributes* derived from the internal states of objects.

Therefore, in the structural model of Vigil (Figure 4.9) we could have modelled an external state as being an attribute of the class Class. In this case, we would not have classes StateSpace and State in the structural model of Vigil, instead, we would have the class State as a subclass of the class Attribute.

Suppose that we have a model were the class State is a subclass of the class Attribute. This design creates a circularity in the system because an attribute of a class represented as a database object is itself (represented as) a database object. An analogy might help us here. There are programming languages, such as Smalltalk and Lisp, that are based on such strict object-oriented design. In these languages, every concept of the programming system is implement as an object, e.g., instances of primitive types are objects, operators are objects, etc. This is not the case with C++ where instances of operators and primitive types are not objects. For example, we do not have in C++ a class Int from which integers are instantiated. In the design of Stabilis and Vigil we have decided to comply with the programming abstraction provided by C++. This is the reason behind the use of different classes to represent states; the class State stores information about states of a transition system associated to a class and the class StateSpace is used to store the state spaces of an object during runtime.

The class StateSpace implements an external state space as a graph using an adjacency list whose interface has methods `set_state(String state)` and `test_state(String state)`. Additionally, the interface of class Object has been augmented with methods `set_state(String state)` and `test_state(String state)`. These methods are used in the implementation of transitions to manipulate the external state spaces of objects. Suppose that we have a class A that is derived from class Object with an attribute `StateSpace state`. Then, if we have an instance of class A, say a, we can write "`a.set_state("S");`". The method `set_state("S")` of class Object is executed, which calls the method `set_state("S")` of class StateSpace.

Suppose that the external state space of class A, implemented by the variable `state`, has two external states $E_1 = \{E, F, G\}$ and $E_2 = \{H, I\}$. If we want to retrieve an instance of class A with current state $\{F, I\}$, then we can specify the following query: `A a("A(state == 'F' && state == 'I')", ...)`. This query is resolved by Stabilis against the value of the attribute `state` using graph traversal methods implemented by the class StateSpace. The name of the StateSpace attribute can be repeated up to as many times as the number of external states of the class. In this example, `state` can appear up to two times in the query expression.

Class Name	Class Definition	lines
State	4.1	3-12 (constructors), 15-17
GuardedAction	4.2	4-13 (constructors), 16-18
TransitionSystem	4.3	2-11 (constructors), 14-16

Table 4.1: Families of methods: State, GuardedAction, and TransitionSystem.

4.3.2 Classes Interfaces

The structural model of Vigil (Figure 4.9) is part of the structural model of the management system. The same indexing protocols used in Stabilis are also valid for the metaclasses of the structural model of Vigil. We discuss the interfaces of the classes of Vigil very concisely because we know already how structural models are represented in Stabilis. We group attributes and methods into families to facilitate our discussion.

Methods

The constructors of the classes State, GuardedAction, and TransitionSystem (Table 4.1) have signatures and functionality identical to the constructors of the classes of the structural model of Stabilis. There are constructors that are used only during the booting of the management system and constructors that are used during the normal operation of the system, with queries being passed as parameters. In fact, in almost every other method the interfaces of these classes are similar to the interfaces of the classes of the structural model of Stabilis. The most important difference is in the addition of methods for the automatic generation of control programs (transition systems).

Code Generation

Automatic code generation (Table 4.2) is implemented by a branch of the family of methods of code generation that have their execution triggered when a user instantiates an object model, an instance of class **Model**, and sends to it a message **gen_code**. The method **gen_code** has parameters specifying a path where the generated code is to be written and a version number.

For example, suppose that a programmer wants to generate a control program for application object `a`. The generation of a control program for an application object entails three steps: (i) the retrieval (activation) of the object `a`, (ii) the retrieval of the object model of the application, say `m`, and (iii) execution of `m.gen_code(..., a)`. Additionally, if the parameter of type **Object** is null, then

Class Name	Class Definition	lines
State	4.1	13, 14
GuardedAction	4.2	14, 15
TransitionSystem	4.3	12, 13

Table 4.2: Family of code generation methods.

Figure 4.11: Relationship multiplicity and code generation.

gen_code will traverse the whole object model (database schema) and generate control programs for each of the objects indexed by it.

Relationship multiplicity and code generation Suppose we have designed an structural model where two classes, say A and B, are related to each other as shown in Figure 4.11. Further, let us suppose that we have k instances of B identified by $b_i, 0 \leq i \leq k$ related to one instance of A, say a, and that A's control object has to have access to b_i. If we call `gen_code(..., a)`, then the control program generated for application object a has the form: $C_a :: [C_{ab_1} \| \ldots \| C_{ab_k}]$, where each object C_{ab_i} is an instantiation of the transition system of the class A for object b_i of the class B, i.e., we have an instance of the transition system of A for each object of class B related to a.

4.3.3 An Example Program

Let us introduce a very simple example to show how the control model of Vigil allows us to instantiate and interpret management programs that are based on transition systems. We already know that a Stabilis program consists of two autonomous subprograms: a schema program and an application program. Similarly, a Vigil program consists of a schema program and a control program.

When a schema program instantiates metaclasses of the control model it generates objects that *define* a control model: the states, guarded actions, and transition systems. The internals of Stabilis use information stored in these metaobjects to generate C++ programs that implement control programs. Later, the code of the control programs is linked to Vigil's libraries to create an executable control program. The schema program for a control model generates the backstage objects that represent transition systems. This schema program is run only once to

Class 4.1 Class State.

```
class State : public Object
{
 protected:
 (1) String name;
 (2) Boolean initial;

 private:
 (3) Boolean save_common_state(ObjectState&);
 (4) Boolean restore_common_state(ObjectState&);

 protected:
 (5) virtual OpHistory* volatile_rc();
 (6) virtual OpHistory* permanent_rc();

 public:
 (7) State(Context*, Birth, OpHistory*); // class for class
 (8) State(String a_name, Context*, Class*, Birth, OpHistory*);
 (9) State(String sexpr, Context*, Birth, OpHistory*);
 (10) State(String sexpr, Context*, Reincarnation, OpHistory*);
 (11) State(String sexpr, Context*, Provide, OpHistory*);
 (12) State(Pip*, Context*, Reincarnation, OpHistory*);

 (13) OpHistory* gen_code(String path, unsigned version, Object*);
 (14) OpHistory* gen_code(String path, unsigned version, Object*,
                 StringList& models, StringList& classes);

 (15) virtual Boolean save_state (ObjectState&, ObjectType);
 (16) virtual Boolean restore_state(ObjectState&, ObjectType);
 (17) virtual ostream& print(ostream&);
};
```

Class 4.2 Class GuardedAction.

```
class GuardedAction : public Object
{
 protected:
 (1) String guard;
 (2) String on;
 (3) String action;

 private:
 (4) Boolean save_common_state(ObjectState&);
 (5) Boolean restore_common_state(ObjectState&);

 protected:
 (6) virtual OpHistory* volatile_rc();
 (7) virtual OpHistory* permanent_rc();

 public:
 (8) GuardedAction(Context*, Birth, OpHistory*); // class for class
 (9) GuardedAction(String a_name, Context*, Class*, Birth, OpHistory*);
 (10) GuardedAction(String sexpr, Context*, Birth, OpHistory*);
 (11) GuardedAction(String sexpr, Context*, Reincarnation, OpHistory*);
 (12) GuardedAction(String sexpr, Context*, Provide, OpHistory*);
 (13) GuardedAction(Pip*, Context*, Reincarnation, OpHistory*);

 (14) OpHistory* gen_code(String path, unsigned version, Object*);
 (15) OpHistory* gen_code(String path, unsigned version, Object*,
                          StringList& models, StringList& classes);

 (16) virtual Boolean save_state (ObjectState&, ObjectType);
 (17) virtual Boolean restore_state(ObjectState&, ObjectType);
 (18) virtual ostream& print(ostream&);
};
```

Class 4.3 Class TransitionSystem.

```
class TransitionSystem : public Object
{
 protected:
 (1) String name;

 private:
 (2) Boolean save_common_state(ObjectState&);
 (3) Boolean restore_common_state(ObjectState&);

 protected:
 (4) virtual OpHistory* volatile_rc();
 (5) virtual OpHistory* permanent_rc();

 public:
 (6) TransitionSystem(Context*, Birth, OpHistory*); // class for class
 (7) TransitionSystem(String a_name, Context*, Class*, Birth, OpHistory*);
 (8) TransitionSystem(String sexpr, Context*, Birth, OpHistory*);
 (9) TransitionSystem(String sexpr, Context*, Reincarnation, OpHistory*);
 (10) TransitionSystem(String sexpr, Context*, Provide, OpHistory*);
 (11) TransitionSystem(Pip*, Context*, Reincarnation, OpHistory*);

 (12) OpHistory* gen_code(String path, unsigned version, Object*);
 (13) OpHistory* gen_code(String path, unsigned version, Object*,
                          StringList& models, StringList& classes);

 (14) virtual Boolean save_state (ObjectState&, ObjectType);
 (15) virtual Boolean restore_state(ObjectState&, ObjectType);
 (16) virtual ostream& print(ostream&);
};
```

define a transition system schema which is going to be used as the basis for the creation and execution of many replicas of control programs.

The Plant-Pollinator system (Figure 4.12) has been chosen as an example because it allows a reasonably complete, but simple description of Vigil's functionality. Next, we specify the system informally. We are required to specify a control program to govern the pollination process of an insect-pollinated plant.

The structural model of the Plant-Pollinator system is very simple: a plant is associated to a pollinator in a one-to-one association (Figure 4.12). Both classes have to be derived from the base class Object. Inheritance from class Object guarantees access to pre-defined sensors and actuators that allow the manipulation of the external state of objects.

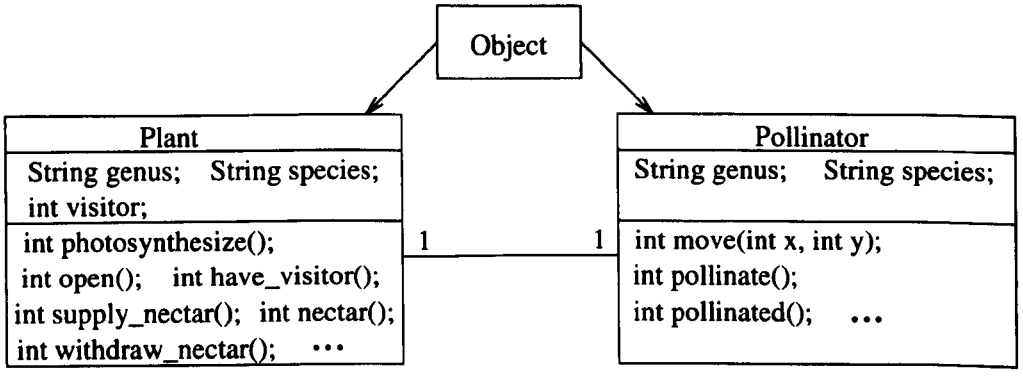

Figure 4.12: Plant-Pollinator structural model.

In our idealized system, a plant goes through a very simple life cycle: as soon as it is born it begins to photosynthesize (Figure 4.13, subtransition system A). The transition "/`photosynthesize()`" means "when **true** do `photosynthesize()`". The guard is always **true**, meaning that actuator `photosynthesize()` is executed continuously; this transition is always enabled; thus, it always fires.

The specification "/`photosynthesize()`" (Figure 4.13, subtransition system A) is equivalent to "/⟨`Plant`⟩.`photosynthesize()`". The latter uses a class qualifier enclosed in angular brackets to specify that the actuator `photosynthesize()` belongs to an instance of the class Plant. When, in our notation, the actuator or sensor is not prefixed by a class qualifier it belongs to the instance of the class to which the control model is associated.

During the flowering season a plant opens its flowers when the sun rises and closes them when the sun sets (Figure 4.13, subtransition system B). Plants are very economical, while open they do not offer nectar continuously (Figure 4.13, subtransition system OPEN). They only start producing nectar when a pollinator is visiting the plant, that is, when the pollinator has become associated to the

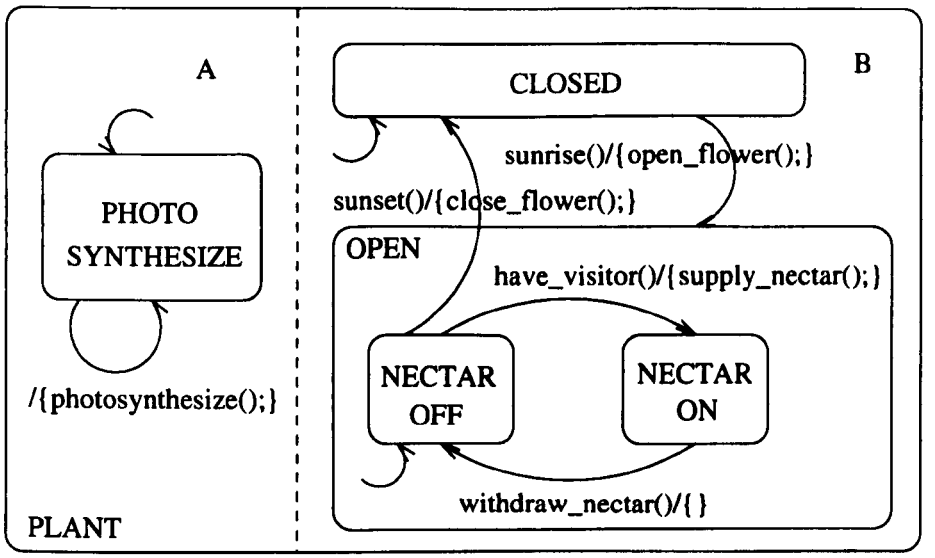

Figure 4.13: Plant control model.

plant. Once they have offered nectar to the pollinator for a period of time, they stop offering nectar to it. The control program of a plant determines it has a visitor using sensor `Boolean have_visitor()`. The actuator `supply_nectar()` is executed by the control program of a plant to make it offer some nectar to its associated pollinator. After a period of time sensor `withdraw_nectar()` becomes `true` and the guarded action that makes the transition from NECTAR_ON to NECTAR_OFF is fired. The specification of this transition introduces additional notation; if a transition has its action part empty or specified by an empty statement "{}", then a null action is executed during the transition firing.

A pollinator is a simple creature, immediately after its birth it starts searching for nectar (Figure 4.14 subtransition system A). As soon as it finds a plant with flowers open it becomes associated to it in the hope of getting a steady supply of nectar. An association between plant and pollinator is only possible when their coordinates match. If the nectar supply is not to the liking of the pollinator it departs in search for a better nectar supplier. If a plant offers nectar at least once, then the pollinator tries to pollinate it (Figure 4.14, subtransition system B).

In the control model of class **Plant** (Figure 4.13) we have deliberately represented all aspects of the behaviour of our idealized plant. We have done this to show that it is the responsibility of the designer of the distributed program the decision of how much of the behaviour of the program he wants to model using transition systems. As designers of the Plant-Pollintator program, we have

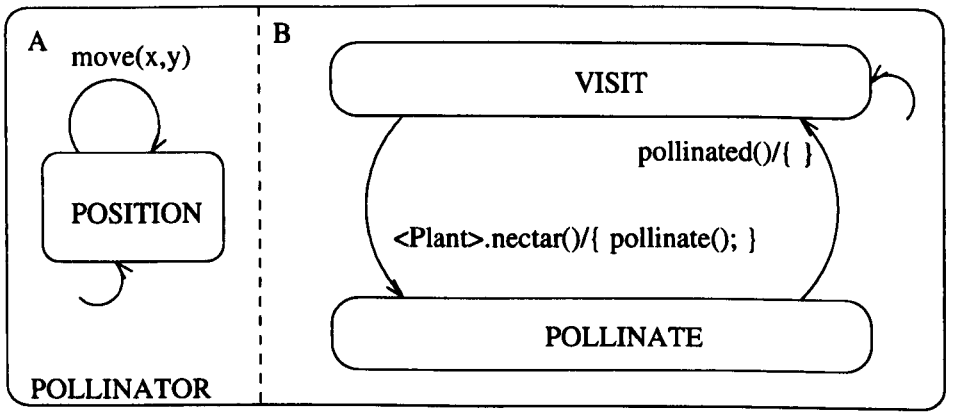

Figure 4.14: Pollinator control model.

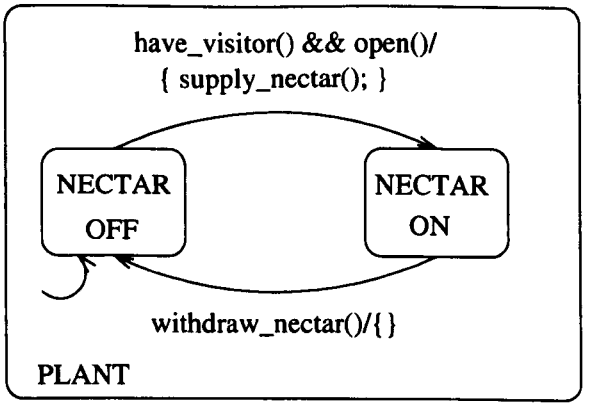

Figure 4.15: Simplified Plant control model.

decided that certain aspects of it can be resolved by the application program because we are only interested in the management of the pollination process. Thus, we reduce the control models to only those transitions that we see as being relevant to the management of the pollination process.

Photosynthesis and the opening/closing of flowers are functions that have an indirect impact on pollination, then we can reduce the control model of the plant to the transitions that are inside the state OPEN (Figure 4.15). Photosynthesis and opening/closing flowers has become concern of the implementors of the application program for the class Plant. At management level, we are interested in representing explicitly only the transitions related to the offer of nectar because they have a direct relationship with the pollination process. If we compare the reduced control model (Figure 4.15) to the original (Figure 4.13) we can see

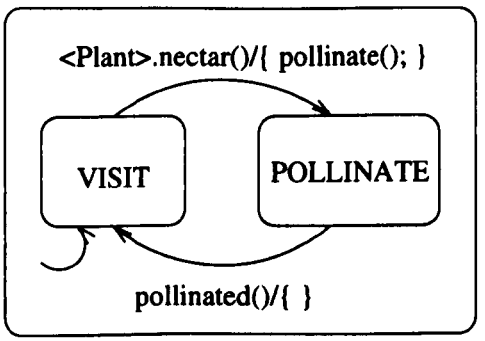

Figure 4.16: Simplified Pollinator control model.

that the guard of the transition from state NECTAR_OFF to state NECTAR_ON in the reduced control model (Figure 4.15) is specified using two sensors, one of them is the sensor `open()`. This sensor hides from the implementor of the control program the details of the system that were explicitly represented in the original control model (Figure 4.13). To us the components of the system that deal with photosynthesis and opening/closing of flowers have become atomic. Similarly, we can reduce the control model of the pollinator (Figure 4.14) to a submodel were subtransition system A is not considered (Figure 4.16). This simplification is made because we consider the movement of the pollinator and its association to a plant not to be a management concern.

Once we are satisfied with the design of the structural and control models for the Plant-Pollinator system, we can write the schema program for it. An excerpt of the schema program for the control model of class **Plant** is shown in Program 4.1. Schema programs create backstage objects. To represent a control model we have to create metaclasses **TransitionSystem**, **GuardedAction** and **State**.

The structure of the schema program mirrors the structure of the object model (structural and control models) of the Plant-Pollinator system (Program 4.1). Line 1 has the function of creating the model for the Plant-Pollinator. In Line 2, we create the class responsible for indexing instances of class **Plant**. In line 3 we indicate that this class is part of the Plant-Pollinator object model by relating it to the instance created in line 1. Lines 1, 2, and 3 create backstage objects that belong to the representation of the structural model. Line 4 creates an instance of **TransitionSystem** which represents the root of the Plant control model. Next, line 5 relates the transition system to its class, in accordance with principle C_8. Lines 6, 7, and 8 create instances of class **State**. We have an instance for each state of the Plant control model (Figure 4.15). Lines 9 and 10 create the SuperState-SubState relationships between state PLANT and states NECTAR_ON and NECTAR_OFF. Finally, lines 11, 12, and 13 create metaob-

jects that represent a guarded action. This excerpt illustrates that diagrammatic representations of a transition system can be translated into a database schema.

Program 4.1 Schema for Plant-Pollinator control model.

```
main() {
 (1) Model* model_PlantPollinator = new Model(
       "Model(name == 'PlantPollinator')", universe, BIRTH, oph);
 (2) Class* m_Plant = new Class("Class(name == 'Plant')", universe,
       BIRTH, oph);
 (3) *oph += model_PlantPollinator→relate("RootClass", m_Plant);
 (4) TransitionSystem* plant_vigil = new TransitionSystem(
       "TransitionSystem(name = 'Plant')", universe, PROVIDE, oph);
 (5) *oph += m_Plant→relate("TransitionSystem", plant_vigil);
 (6) State* plant = new State("State(name = 'PLANT')", universe, BIRTH, oph);
 (7) State* nectar_off = new State("State(name = 'NECTAR_OFF'
       && initial = 1)", universe, PROVIDE, oph);
 (8) State* nectar_on = new State("State(name = 'NECTAR_ON')",
       universe, PROVIDE, oph);
 (9) *oph += plant→relate("SubState", nectar_on);
 (10) *oph += plant→relate("SubState", nectar_off);
 (11) GuardedAction* g1 = new GuardedAction("GuardedAction(
                  guard = 'have_visitor() && open()' &&
                  action = '{ supply_nectar(); }')", universe, PROVIDE,
oph);
 (12) *oph += g1→relate("Predecessor", nectar_off);
 (13) *oph += g1→relate("Successor", nectar_on); }
```

4.4 Developing Distributed Programs

After the execution of the schema program we have all the necessary backstage structures in place and can consider the generation and execution of control programs. Before proceeding to the explanation of how control programs are interpreted we make explicit the process we have followed so far to develop a distributed program.

Problem Analysis In this phase the designer/implementor of the distributed program has to write or obtain an initial description of the problem and analyze it to delimit requirements and goals.

Modelling Identify classes and attributes. Attributes include data terms, methods and relationships. Relationships are used to simplify the structural

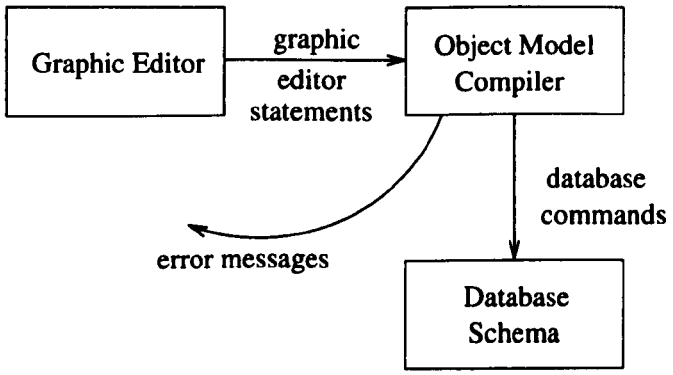

Figure 4.17: Automating the process of model representation.

model of the distributed program. Developing a control model is the next step of the process. In this step we have to prepare a control model for each class that plays an important role in the reconfiguration of the components of the program. At the end of this phase the designer of the distributed program has finished the preparation of the program's object model.

Model Representation The object model is represented as a database schema. Currently, the programmer of the application has to translate object models into schema programs. This step can be totally automated using a graphical model editor and a model compiler. The model compiler is responsible for the translation of the graphical representation of the object model into an schema program (Figure 4.17).

Program Generation The generation of parts of the distributed program is carried out in two steps:

1. The object model of the distributed program is retrieved from the database and the message `gen_code(...)` is passed to it. This message triggers the generation of the C++ source code that defines the classes of the model. The code of a application-specific interactive query interpreter is also generated by `gen_code(...)`. Now the user has to complete the implementation of the methods of the classes. Clients are also implemented during this step. Using the interactive query interpreter the programmer instantiates application objects. For example, the code of the automatically generated header file for class **Plant** is shown in Class 4.4. In lines 1 to 4 of the header file (Class 4.4) we can see the attribute declarations. In lines 5 to 8 we have the methods used by the object manager to manipulate the state of objects. The constructors are

declared in lines 9 to 12. In lines 14 and 15 we see some of the sensors and actuators of the class. Finally, in lines 16 to 17 we have the methods required by the state manager of Arjuna.

Class 4.4 Class Plant.

```
class Plant : public Object
{
 protected:
 (1) String genus; int number;
 (2) String species; int visitor;
 (3) int open;
 (4) StateSpace state;
 private:
 (5) Boolean save_common_state(ObjectState&);
 (6) Boolean restore_common_state(ObjectState&);
 protected:
 (7) OpHistory* volatile_rc();
 (8) OpHistory* permanent_rc();
 public:
 (9) Reference(String sexpr, Context*, Birth, OpHistory*);
 (10) Reference(String sexpr, Context*, Reincarnation, OpHistory*);
 (11) Reference(String sexpr, Context*, Provide, OpHistory*);
 (12) Reference(Pip*, Context*, Reincarnation, OpHistory*);
 (13) int photosynthesize();
 (14) int have_visitor(); int open();
 (15) int supply_nectar(); int withdraw_nectar();
 (16) virtual Boolean save_state (ObjectState&, ObjectType);
 (16) virtual Boolean restore_state(ObjectState&, ObjectType);
 (17) virtual ostream& print(ostream&);
};
```

2. Using the query interpreter the programmer retrieves the object model, say m, and application objects. For each application object of the Plant-Pollinator system, `m.gen_code(...)` is executed. Let us suppose we have an instance of class Plant, say p of species "sp", the execution of `m.gen_code(..., p)` causes the generation of the Program 4.2. The control program obtains a reference to Stabilis (Program 4.2, line 1) and to the application object which it is controlling. The code of transitions 1 (Program 4.2, lines 3-7) and 2 (lines 8-10) is created using the control metainformation stored in Stabilis. In the main program we have the instantiation of the scheduler (Program 4.2, line 11), the registration of transitions (lines 12-13), and the activation of the scheduler (line 14).

Program 4.2 Control program for class Plant.

```
/* automatically generated code */
/* Plant species = sp */
(1) Context* universe = new Context("Universe", REINCARNATION, oph);
(2) Plant plant("Plant(species == 'sp')", universe, REINCARNATION, p_oph);
(3) int guard_1()
(4) { return (plant.test_state("NECTAR_OFF") && plant.have_visitor() &&
(5) plant.open()); }
(6) int action_1()
(7) { plant.set_state("NECTAR_ON"); plant.supply_nectar(); }
(8) int guard_2()
(9) { return (plant.test_state("NECTAR_ON") && plant.withdraw_nectar()); }
(10) int action_2() { return 1; }
main() {
(11) const unsigned DELAY = 30; Vigil vigil(DELAY);
(12) vigil.register(guard_1,action_1);
(13) vigil.register(guard_2,action_2);
(14) vigil.run(); }
```

Execution To execute the distributed program the programmer runs the control program and application programs. Reconfiguration can be accomplished by changing the relationships between the objects of the program.

4.4.1 Scheduling of Guarded Actions

In this Section we show the code of the interpreter of transition systems. We have implemented to schedulers of guarded actions.

An important element of the object and action model adopted in this work is that concurrency is represented by *interleaving*. This means that two objects executing in parallel never execute their atomic methods at precisely the same instant, but take turns in executing atomic methods. When concurrent computation is represented by interleaved computation two problems arise:

Interference Atomicity guarantees absence of interference between concurrently executing guarded actions. The degree of concurrency allowed in the computation of the guarded actions can be increased through the implementation of different of atomic action models.

Independent Progress The problem of independent progress is that, in an parallel execution, the computation of each object of the distributed program,

keeps advancing, since each guarded action is independently responsible for its own progress. In an interleaved computation, the requirement for control objects is that enabled transitions be continuously chosen and executed. If there is nothing to disallow a computation in which only one transition is ever chosen, then such a computation ensures progress of the preferred action, or actions, but keeps all other guarded actions stagnant.

The solution to the problem of independent progress is the introduction of the requirement of *fairness* into the computation model of interleaved computations. According to Ben-Ari, four possible specifications of fairness are:

Weak fairness If an object continuously makes a request, eventually it will be granted.

Strong fairness If an object makes a request infinitely often, eventually it will be granted.

Linear waiting If an object makes a request, it will be granted before any other object is granted the request more than once.

First-in first-out (FIFO) If an object makes a request, it will be granted before that of any object making a later request.

The first scheduler is based on a simple algorithm that is not fair. Class 4.5 and Program 4.3 show the header file and implementation of method **run** for the base class **Vigil**. The constructor of the scheduler accepts a parameter (Class 4.5, line 2) that specifies the delay time in milliseconds between transition evaluations. The method `registre(...)` (Class 4.5, line 2) is used to register transitions with the scheduler, it accepts pointers to methods as parameters: the first parameter is a pointer to a guard, the second is a pointer to an action. The attributes (Class 4.5, lines 6-9) provide the data structures used to manage transitions.

The code of the method `run()` of class **Vigil** (Program 4.3) is simple. The initialization (lines 1-4) creates an operation history, initializes temporary variables used to test the result of guards and actions, and declares a pointer to an atomic action. The scheduler sits on an infinite loop (line 5) doing the following:

1. select a guarded action (line 6),
2. start an atomic action (line 10),
3. evaluate the guard of the selected transition (line 11).
4. if the guard of the selected transition is **true**, then evaluate the action of the selected transition (line 12),

5. if both guard and action have executed without problems, then commit the atomic action, otherwise, abort it (lines 13-15),

6. wait `delay_time` before starting the evaluation of the next transition (line 17).

Class 4.6 and Program 4.4 show the header file and code for a subclass of the class **Vigil** where method **run** schedules guarded actions using a priority-queue based algorithm; this algorithm implements weak fairness. Apart from the use of a priority queue to select the next transition to be evaluated, this algorithm has an structure similar to the one we have discussed above.

Weak fairness is termed justice in [102, pages 103-175] where the algorithm for class **jVigil** is detailed. Manna and Pnueli claim that this algorithm can generate every fair computation of a program based on interleaved execution of guarded actions [102, pages 158-159].

The design of Vigil is such that at any time a programmer can specialize the base class **Vigil** and implement a new scheduler for guarded actions. We have experimented with these two algorithms primarily because they are simple.

Class 4.5 Base class **Vigil**.

```
(1) const int MAX_GA = 500; // Maximum number of guarded actions
class Vigil
{ public:
 (2) Vigil(unsigned delay_time = 5);
 (3) ~Vigil();
 (4) void registre(int (*guard)(), int (*action)());
 (5) void run();
 private:
 (6) unsigned delay_time;
 (7) int gan; // actual number of guarded actions
 (8) int (*guards [MAX_GA])();
 (9) int (*actions[MAX_GA])();
};
```

Program 4.3 Implementation of method **run** for class Vigil.

```
void Vigil::run()
(1) { OpHistory* oph = new OpHistory;
(2) int guard_satisfied = 0; int action_satisfied = 0;
(3) unsigned unslept = 0;
(4) AtomicAction* A; int i = 0;
(5) for (;;) {
(6) i = (i + 1) % gan;
(7) A = new AtomicAction();
(8) guard_satisfied = 0; action_satisfied = 0;
(9) if ((guards[i] ≠ 0) && (actions[i] ≠ 0))
(10) { *oph += A→Begin();
(11) guard_satisfied = (*guards[i])();
(12) if (guard_satisfied ≠ 0) action_satisfied = (*actions[i])();
(13) if ((guard_satisfied ≠ 0) && (action_satisfied ≠ 0))
(14) *oph += A→End();
(15) else *oph += A→Abort();
(15) delete A; A = 0;
(17) unslept = sleep(delay_time); } } }
```

Class 4.6 Class jVigil, justice-based scheduler of guarded actions.

```
(1) const int MAX_GA = 500; // Maximum number of guarded actions
class pVigil : public Vigil
{ public:
 (2) Vigil(unsigned delay_time = 5);
 (3) ~Vigil();
 (4) void registre(int (*guard)(), int (*action)());
 (5) void run();
 private:
 (6) unsigned delay_time;
 (7) int gan; // number of guarded actions registered
 (8) int (*guards [MAX_GA])();
 (9) int (*actions[MAX_GA])();
 (10) int priority[MAX_GA];
 (11) int minimum();
 (12) void set_priority(int k);
};
```

Program 4.4 Implementation of method **run** for class jVigil.

```
void Vigil::run() {
(1) OpHistory* oph = new OpHistory;
(2) int guard_satisfied = 0; int action_satisfied = 0;
(3) unsigned unslept = 0;
(4) AtomicAction* A;
(5) int i = 0;
(6) for (;;) {
(7) i = minimum();
(8) A = new AtomicAction();
(9) guard_satisfied = 0; action_satisfied = 0;
(10) if ((guards[i] ≠ 0) && (actions[i] ≠ 0))
(11) {
(12) *oph += A→Begin();
(13) guard_satisfied = (*guards[i])();
(14) if (guard_satisfied ≠ 0) action_satisfied = (*actions[i])();
(15) if ((guard_satisfied ≠ 0) && (action_satisfied ≠ 0))
(16) *oph += A→End();
(17) else *oph += A→Abort();
(18) delete A; A = 0;
(19) unslept = sleep(delay_time); }
(20) set_priority(i); } }
```

4.5 Conclusions

We have started this Chapter arguing that implementors of adaptable distributed programs have to be concerned with the explicit representation and management of structural and control information about a distributed program. Further, we have argued that it is beneficial during program development to maintain the implementation of management policies separated from the implementation of functional aspects.

The notion of reactive programs has played an important role in the definition of the functionality of Stabilis and Vigil. From the point of view of the management system a distributed program is seen as the superposition of two reactive programs: an application program and a management program. Application programs are implemented using the facilities provided by Arjuna, control programs are implemented using the same facilities, but are state machines whose interpretation is the responsibility of Vigil's guarded-action scheduler.

Having specified that control programs were implemented as state machines, or transition systems, we proceeded to define the conventions adopted in the representation of state machines (control models). The control model supported by Vigil allows the representation of states and guarded actions (transitions). Additionally, Vigil's control model allows the creation of hierarchies and aggregation of states. The generalization/specialization relationship for states allows the representation of superstates and substates. The aggregation of states allows us to compose state machines that are executed independently.

The implementation of Vigil is based on the instantiation of a structural model that represents the concepts of the control model. The representation of the structural model of Vigil is carried out in the same way as the representation of the structural model of Stabilis. A schema program is created to instantiated metaclasses for each of the classes of the model. The execution of the schema program creates the objects that maintain information about control models.

The discussion of the functionality of Vigil is carried out using the example of the Plant-Pollinator system. With the Plant-Pollinator example we were able to follow each of the steps of the design and implementation of a distributed program that uses the management system. These steps include the design of the object model, its representation in Stabilis using schema programs, automatic code generation, implementation, and execution.

Finally, we discussed the implementation of two schedulers of guarded actions for Vigil. The first scheduler is not fair; the second algorithm implements weak fairness.

Chapter 5

Example Programs

In this Chapter we discuss two examples of use of the management system. The first example is the result of extending the well-known dining philosophers problem into a distributed program that shows how dynamic reconfiguration of objects is achieved using Stabilis and Vigil. The second example is an index system of a distributed database; it shows how Stabilis and Vigil can be used to implement a dynamically reconfigurable index. The index system relies on a cache coherency protocol for its operation, Stabilis and Vigil are used to implement the control mechanisms of this protocol that have to be integrated with the mechanisms used in the reconfiguration of the index.

5.1 Evolving Philosophers

The distributed dining philosophers problem [50], diners, is solved for situations where the population of a community of philosophers can change. The solution presented here is essentially the same solution presented by Kramer and Magee [80] for the Conic environment; it has only been reinterpreted using Arjuna, Stabilis and Vigil.

In the specification of the diners proposed by Kramer and Magee [80], philosophers are born, live, and die without disturbing the lives of the other philosophers, despite having to share resources with their neighbours while alive. This version of the diners problem is known as the evolving dining philosophers.

Philosophers are arranged in rings with neighbouring philosophers sharing a fork between them. A philosopher is either thinking, hungry, or eating. To move from hungry to eating a philosopher must acquire both his left-hand and right-hand forks. The fact that philosophers share forks and must acquire two forks before they can eat leads to conflicts. To solve the original diners problem is essentially to design a program to adjudicate conflicts: neighbouring philosophers do not eat simultaneously and hungry philosophers eat eventually, given that no philosopher is allowed to eat forever. To solve the evolving dining philosophers [80] is to add mechanisms for dynamic configuration of rings of philosophers while maintaining the overall dependency of the application.

5.1.1 Reconfiguration Management

In the evolving philosophers, the death and birth of philosophers is used as an analogy to dynamic addition and removal of objects of an application program. The strategy devised by the Conic team [80] to solve the problem of dynamically reconfiguring objects of a distributed program includes the following steps:

- Determine the set of objects, say Q, that are affected by the reconfiguration so that disruption to the activity of the program is minimized.

- All elements of Q have to be taken from their operational state, that is, an active state, to a quiescent state, or passive state. Such change has to be carried out without compromising the correctness of the distributed program. Quiescence is achieved with the contribution of the application program, meaning that the program's design has to be extended to provide the operations used during passivation/activation of its objects.

- The application objects affected by the dynamic change are unlinked. Link and unlink operations are used to logically connect and disconnect objects. If an object is logically connected to another then it is able to establish a communication link with that object when necessary.

- Objects are removed and/or added as dictated by the reconfiguration procedure.

- The objects affected by the reconfiguration procedure are linked, respecting the structural model of the distributed program.

- All objects passivated earlier are reactivated, taking the application program back to a fully operational state.

5.1.2 Structural Model

We can model each philosopher as an object that communicates with its left-hand and right-hand neighbours by sending messages to them via method execution. The diners structural model is simple: the class **Philosopher** is related to itself via two distinct association relationships: **Left** and **Right**, meaning that a philosopher knows its left and right neighbour (Figure 5.1). These associations represent the fact that philosophers must be able to communicate with each other in order to exchange forks.

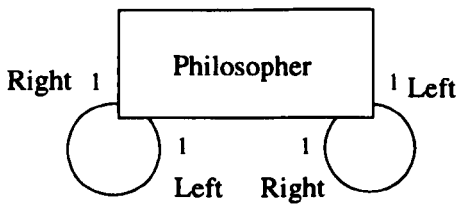

Figure 5.1: Structural model for the diners.

Class Philosopher

Instances of class **Philosopher** (Class 5.1) belong to the application program. Consequently, they have to be instrumented with sensors and actuators before their control objects can gain access to their state.

In our implementation of the evolving philosophers, each philosopher has a name that uniquely identifies him (Class 5.1, line 1); each philosopher has also an order number that is unique (Class 5.1, line 2). Additionally, from an analysis of the solution to the original problem (Section 5.1.3), we can say that each philosopher knows whether or not he has got the forks he shares with his neighbours (Class 5.1, line 3) and whether the forks are dirty or not (Class 5.1, line 4). If a philosopher has a token for a fork then he has not got that fork, but has the right to request it from his neighbour (Class 5.1, line 5). In line 6 we have the declaration of the philosopher's state space.

We have sensors and actuators for each attribute that is of management interest. For example, we have actuator `set_fork(...)` and sensor `fork(...)`; they are used by management objects to control the forks a philosopher has with him (Class 5.1, line 7). Similarly, we have sensors and actuators for the other attributes whose values we want to control (Class 5.1, lines 8-10). The next Section brings a thorough description of the sensors and actuators of class **Philosopher**; their function can be better understood in the light of the control model that governs a philosopher's life.

Class 5.1 Class Philosopher.

```
#define HANDS 2
enum Hand {LEFT, RIGHT}
class Philosopher: public Object {
protected:
 (1) String name;
 (2) int order;
 (3) Boolean fork[HANDS];
 (4) Boolean dirty[HANDS];
 (5) Boolean token[HANDS];
 (6) StateSpace state;
public:
 // constructors deleted
 (7) int set_fork(Hand, Boolean); int fork(Hand);
 (8) int set_dirty(Hand, Boolean); int dirty(Hand);
 (9) int set_token(Hand, Boolean); int token(Hand);
 (10) int set_order(int); int order();
};
```

5.1.3 Control Model

Chandy and Misra [42] describe the behaviour of the diners as follows: "A fork is either clean or dirty. A fork being used to eat with is dirty and remains dirty until it is cleaned. A clean fork remains clean until it is used for eating. A philosopher cleans a fork when mailing it; he is hygienic. An eating philosopher does not satisfy requests for forks until he has finished eating." When not eating, philosophers defer requests for forks that are clean and satisfy requests for forks that are dirty. This solution can be considered to implement a precedence graph such that an edge directed from a philosopher node U to T indicates that U has precedence over T (Figure 5.2).

In Chandy and Misra's solution a philosopher U has precedence over his neighbour T if and only if:

1. U holds the fork and it is clean, or
2. T holds the fork and it is dirty.

Chandy and Misra show that if initially all forks are dirty and located at philosophers such that the precedence graph is acyclic it will remain acyclic since:

1. the direction of an edge, from U to T, can only change when U starts eating, and
2. both edges of a philosopher are simultaneously directed towards him when he starts eating. Chandy and Misra prove that since immediately on finishing eating a philosopher yields precedence to his neighbours, all hungry philosophers will commence eating in a finite time, i.e., no philosopher remains hungry forever.

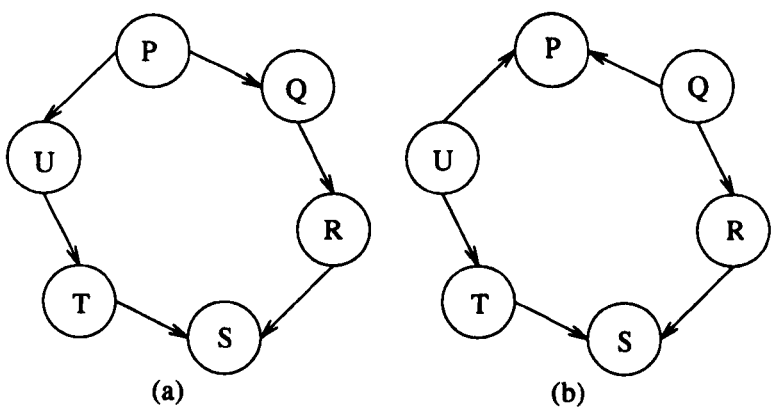

Figure 5.2: Precedence graph. (a) P is hungry. (b) P is eating.

Initialization

Initially all forks are dirty. Forks are distributed among philosophers such that the precedence graph is acyclic. If U and V are neighbours then either U holds the fork and V the request token or vice versa.

Basic Control Model

We describe the control algorithm of a philosopher in function of the control model we have devised for him (Figure 5.3). The state space of a philosopher includes states THINKING, HUNGRY, and EATING. The sensors and actuators necessary to specify the philosopher's control algorithm are:

`fork(`h`)` True if philosopher holds fork of hand h (h can assume values LEFT or RIGHT).

`set_fork(`h`, Boolean)` Passes fork of hand h to neighbour which shares this fork.

`token(`h`)` True if philosopher holds request token for fork of hand h.

`set_token(`h`, Boolean)` Passes request token for fork of hand h to neighbour.

`dirty(`h`)` True if fork of hand h is at philosopher and is dirty.

`set_dirty(`h`, Boolean)` Sets fork of hand h to dirty (TRUE) or clean (FALSE).

`order()` Returns the unique identification number of a philosopher.

`eating_timeout()` True when eating time is over.

`thinking_timeout()` True when thinking time is over.

`other_hand(`h`)` Returns the other fork of hand h that a philosopher uses. For example, `other_hand`(LEFT) = RIGHT.

Not all of these sensors and actuators are used in the actual implementation of the control algorithm, some of them have been included here only as a notational aid. In the diners control model (Figure 5.3), transitions drawn using dashed arcs are implemented and fired by application objects or by queries. Although dashed transitions are not of direct management interest, it would be more difficult to visualize the behaviour of a philosopher if we had not represented them in the control model.

For instance, we can see how transitions t4 and t5 (Figure 5.3) are implemented by analyzing the code of an application object for the diners (Program 5.1). First, the object retrieves a philosopher (Program 5.1, line 1). Next, the object enters a loop (Program 5.1, line 2) that implements what a philosopher normally does if he is active: he thinks for a while, becomes hungry (Figure 5.3, transition t5), eats, and goes back to do his thinking (Figure 5.3, transition t4).

Transition t3 (Figure 5.3) could have been represented as a dashed transition as it is not directly involved in the runtime reconfiguration of a community of philosophers. In this example, we have decided to implement it as part of the control program of a philosopher to illustrate that the division between management and functional aspects of a distributed program is a design decision.

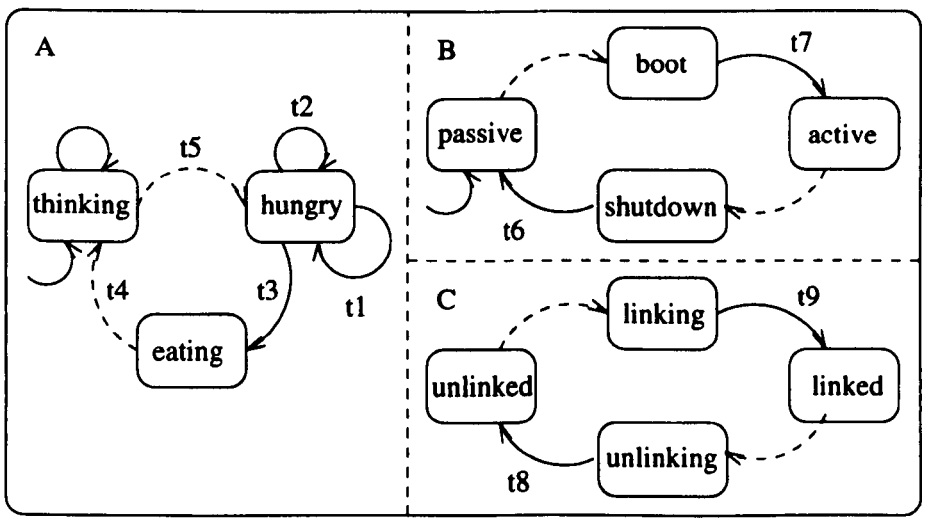

Figure 5.3: Control model for the diners.

A philosopher is at a certain state for a period of time whose duration is determined randomly. This behaviour of a philosopher is implemented by the code of line 3 (Program 5.1). Moreover, he always publishes what he has been doing (Program 5.1, line 4).

Program 5.1 Philosopher's application program.

```
main() {
(1) Philosopher p("Philosopher(surname == 'P')",..., REINCARNATION,...);
    int n = 0;
(2) for (;;)
(3) { srand(time(0)); n = rand(); n = n % MAX_SLEEP; sleep(n);
(4) cout << p;
(5) if ( p.test_state("ACTIVE") ) {
(6) if ( p.test_state("THINKING") ) { p.set_state("HUNGRY"); }
(7) if ( p.test_state("EATING") ) { p.set_state("THINKING"); } }
    }
}
```

We can see the code that implements transition t5 in line 6: the philosopher flips his state from thinking to hungry. Transition t4 is implemented by the code of line 7: the philosopher decides that he has finished eating and resumes thinking (Program 5.1).

In line 5 (Program 5.1), we can see that a passive philosopher does not change his state. When a philosopher is passive he stays at the state his control program has set him until he is reactivated. The code of line 5 is an example of a program's contribution to the dynamic reconfiguration of its objects. The algorithm of the original diners problem does not include the code of line 5. It has been added to allow dynamic reconfiguration of philosophers.

The transitions that implement the basic control model of a philosopher are (Figure 5.3, transition system A):

(t1) Requesting a fork of hand h:

$$(\texttt{token}(h) \wedge \neg\texttt{fork}(h))/$$
$$\{\langle\texttt{neighbour}_h\rangle.\texttt{set\_token}(\texttt{other\_hand}(h), \text{TRUE});$$
$$\texttt{set\_token}(h, \text{FALSE})\}$$

(t2) Releasing a fork of hand h:

$$(\texttt{token}(h) \wedge \texttt{dirty}(h))/\{\texttt{set\_dirty}(h, \text{FALSE}); \texttt{set\_fork}(h, \text{FALSE});$$
$$\langle\texttt{neighbour}_h\rangle.\texttt{set\_fork}(\texttt{other\_hand}(h), \text{TRUE});\}$$

(t3) Philosopher hungry to eating transition:

(`fork`(LEFT) ∧ `fork`(RIGHT))/
{`set_dirty`(LEFT, TRUE); `set_dirty`(RIGHT, TRUE); }

(t4) Philosopher eating to thinking transition: (`eating_timeout`())/{}

(t5) Philosopher thinking to hungry transition: (`thinking_timeout`()/{}

Let us analyze part of the control program where transitions t1 and t3 are implemented. The code shown in Program 5.2 implements transition t3. In the first line of the the same figure we can note the query used by the control object to get a reference to its application object.

Program 5.2 Implementation of transition t3: HUNGRY to EATING.

```
/* automatically generated code */
/* Philosopher surname = 'P' */
Philosopher p("Philosopher(surname == 'P')", ..., REINCARNATION, ...);
// Transition: HUNGRY → EATING
int guard_1() {
 return (p.test_state("HUNGRY") && p.fork(LEFT) && p.fork(RIGHT)); }
int action_1() {
 p.set_state("EATING"); p.set_dirty(LEFT, TRUE);
 p.set_dirty(RIGHT, TRUE); }
```

The code that implements transition t1 is shown in Programs 5.3 and 5.4; a philosopher is hungry and has both forks with him then he can start eating. For example, the code of `action_2()` (Program 5.3) shows how queries are used by control objects to name application objects in a context independent way.

In Program 5.4 we can see the code of the main program of this control object. The code instantiates Vigil, registers guarded actions with it, and runs Vigil's scheduler by calling `vigil.run()`.

Program 5.3 Implementation of transition t1: Requesting LEFT fork.

```
// Transition: Requesting LEFT fork
int guard_2() {
 return ( p.test_state("HUNGRY") && p.token(LEFT) && !p.fork(LEFT) ); }
int action_2() {
 Philosopher left("Philosopher(Left::Right::surname == 'P')", ...,
REINCARNATION, ...);
 left.set_token(RIGHT, TRUE); p.set_token(LEFT, FALSE); }
```

Program 5.4 Implementation of transition t1: Requesting RIGHT fork.

```
// Transition: Requesting RIGHT fork
int guard_3() {
 return ( p.test_state("HUNGRY") && p.token(RIGHT) && !p.fork(RIGHT) ); }
int action_3() {
 Philosopher right("Philosopher(Right::Left::name == 'P'", ...,
REINCARNATION,...);
 right.set_token(LEFT, TRUE); p.set_token(RIGHT, FALSE); }
main() {
 const unsigned DELAY = 1; jVigil vigil(DELAY);
 vigil.register(guard_1,action_1); vigil.register(guard_2,action_2);
 vigil.register(guard_3,action_3); vigil.run();
}
```

Extended Control Model: Dynamic Reconfiguration

In the extended control model we have to consider the creation of a ring (community) of philosophers, addition of a new philosopher (birth) and deletion of an existing philosopher (death).

An extended diners program must allow philosophers to enter a quiescent state and to be logically linked/unlinked to/from another philosopher. The consistency requirements for the extended diners problem are:

1. that a fork is always shared between two adjacent connected philosophers, and
2. that the precedence graph remains acyclic.

The case where a community has only one philosopher is dealt with by connecting it to itself, thereby allowing it to have two forks.

Philosopher passivate/activate actions Let us explain the contribution a philosopher has to make to allow his control program to activate/passivate him. A philosopher can be passivated if he has not requested a fork and is not hungry or will not become hungry. We model this extended behaviour of a philosopher through state space {SHUTDOWN, PASSIVE, BOOT, ACTIVE}. We have added states SHUTDOWN and BOOT to ease the implementation of reconfiguration procedures; especially when a query interpreter is used as reconfiguration managers. Figure 5.3B shows the transition system devised for the passivation/activation of a philosopher. Again, the dashed transitions are implemented by the application program or can be triggered using a query interpreter. Transitions t6 and t7 are specified:

(t6) Philosopher becomes passive: [**on** (¬HUNGRY)]/{}

(t7) Philosopher becomes active: [**on** (¬HUNGRY)]/{}

Part of the control program generated for transition system B (Figure 5.3) is shown in Program 5.5. Transition t6 is implemented by guarded action 1 (Figure 5.5, `guard_1()` and `action_1()`). Guarded action 2 (Program 5.5, `guard_2()`, `action_2()`) implements transition t7.

Program 5.5 Part of control program generated for transition system B.

```
// Transition: SHUTDOWN → PASSIVE
int guard_1() {
 return (!p.test_state("HUNGRY") && p.test_state("SHUTDOWN")) }
int action_1() { p.set_state("PASSIVE"); }

// Transition: BOOT → ACTIVE
int guard_2() {
 return (!p.test_state("HUNGRY") && p.test_state("BOOT")) }
int action_2() { p.set_state("ACTIVE"); }
```

Philosopher link/unlink actions The control model that deals with unlinking and linking of philosophers is represented by transition system C (Figure 5.3). As in the case of passivation/activation of philosophers, we have not only the states LINKED and UNLINKED, but two extra states: UNLINKING and LINKING.

The linking/unlinking of philosophers has to take into consideration consistency requirements regarding the number of forks held by a philosopher to guarantee that his community remains with the right number and right distribution of forks when he is linked/unlinked to/from it. To satisfy these requirements Kramer and Magee require that each philosopher has a unique identifier and that these identifiers permit a total ordering. Unique identifiers are used as a mean of adjudicating who keeps the forks when the number of philosophers in the community changes. The adjudicator's algorithm can be specified: the philosopher that precedes his neighbour in the total ordering decides where a fork is to be allocated. Thus, only one fork is allocated. The specification of transition t9 has to take into consideration the order of the philosophers being linked/unlinked:

1. a philosopher is being linked (unlinked) to (from) himself, i.e., a philosopher whose order is equal to this order. See transition t9a below.
2. a philosopher is being linked (unlinked) to (from) a philosopher whose order is greater than his order. See transition t9b below.
3. a philosopher is being linked (unlinked) to (from) a philosopher whose order is smaller than his order. See transition t9c below.

(t8) Philosopher becomes unlinked:

$[on(\text{PASSIVE})]/\{\texttt{set\_fork}(h, \text{FALSE}); \texttt{set\_token}(h, \text{FALSE}); \texttt{set\_dirty}(h, \text{FALSE}); \}$

(t9a) Philosopher sets up a link to himself:

$(\texttt{order}() = \langle \texttt{neighbour}_h \rangle.order())[\texttt{on}(\text{PASSIVE})]/$
$\{\texttt{set\_fork}(h, \text{TRUE}); \texttt{set\_dirty}(h, \text{TRUE}); \texttt{set\_token}(h, \text{FALSE}); \}$

(t9b) Philosopher sets up a link to a philosopher whose order is greater than his order:

$(\langle \texttt{neighbour}_h \rangle.order() > \texttt{order}())[\texttt{on}(\text{PASSIVE})]/$
$\{ \texttt{set\_fork}(h, \text{TRUE}); \texttt{set\_dirty}(h, \text{TRUE});$
$\langle \texttt{neighbour}_h \rangle.\texttt{set\_token}(\texttt{other\_hand}(h), \text{TRUE});$
$\langle \texttt{neighbour}_h \rangle.\texttt{set\_state}(\text{"LINKED"}) \}$

(t9c) Philosopher sets up a link to a philosopher whose order is smaller than his order:

$(\langle \texttt{neighbour}_h \rangle.order() < \texttt{order}())[\texttt{on}(\text{PASSIVE})]/$
$\{\ \texttt{set\_token}(h, \text{TRUE});$
$\langle \texttt{neighbour}_h \rangle.\texttt{set\_fork}(\texttt{other\_hand}(h), \text{TRUE});$
$\langle \texttt{neighbour}_h \rangle.\texttt{set\_dirty}(\texttt{other\_hand}(h), \text{TRUE});$
$\langle \texttt{neighbour}_h \rangle.\texttt{set\_state}(\text{"LINKED"})\ \}$

Implementation of link/unlink actions The implementation of the guarded actions used to control the linking/unlinking of philosophers is based on a more detailed version of control model C (Figure 5.3). In its detailed form this transition system C is specified as two concurrent transition systems (Figure 5.4). The state spaces defined for the detailed version make explicit the notion of "hand". For example, the state space of transition system C-left is {UNLINKED_LEFT, LINKING_LEFT, LINKED_LEFT, UNLINKING_LEFT}. If we swap the suffix LEFT for RIGHT in the names of C-left's state space we obtain C-right's state space. As an example of implementation we briefly discuss the implementation of part of the control program generated for C-left's control model.

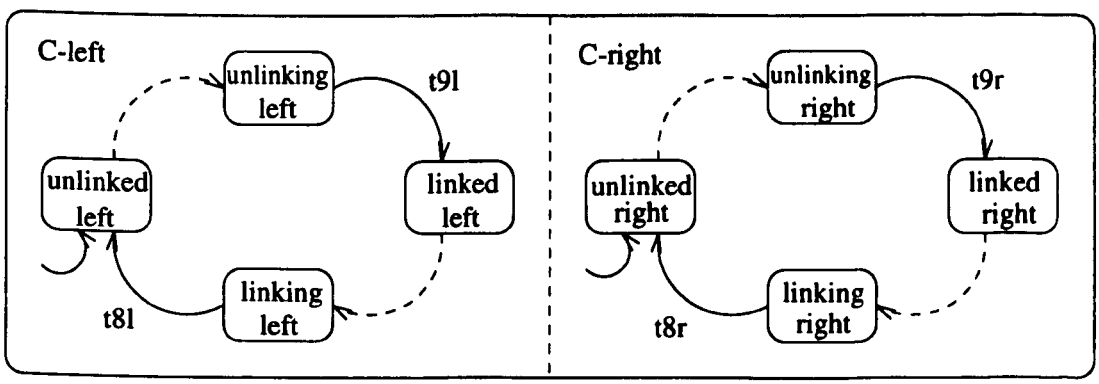

Figure 5.4: Refined transition system C.

Guard and action 1 (Program 5.6) implement transition t9-left (Figure 5.4). When a philosopher is passive and must be unlinked from his left neighbour then it looses its fork.

Program 5.6 Excerpt of control program for C-left (Part 1).

```
// Transition: UNLINKING_LEFT → UNLINKED_LEFT
int guard_1() {
 return ( p.test_state("PASSIVE") && p.test_state("UNLINKING_LEFT") ); }

int action_1() {
 p.set_fork(LEFT, FALSE); p.set_token(LEFT, FALSE); p.set_dirty(LEFT, FALSE);
}
```

Guard and action 2 (Program 5.7) implement transition t8-left for the case where a philosopher is being linked to himself.

Program 5.7 Excerpt of control program for C-left (Part 2).

```
// Transition: LINKING_LEFT → LINKED_LEFT (left.order() = this.order())
int guard_2() {
 Philosopher left("Philosopher(Left::Right::surname == 'P')",...,
 REINCARNATION,...);
 return (p.test_state("PASSIVE") && p.test_state("LINKING_LEFT") &&
         (left.order() == p.order));
}
int action_2() {
 p.set_fork(LEFT, TRUE); p.set_dirty(LEFT, FALSE); p.set_token(LEFT, TRUE);
 p.set_state("LINKED_LEFT");
}
```

Guard and action 3 (Program 5.8) implement transition t8-left for the case where a philosopher is being linked to a left neighbour whose order is greater than his order. In this case he retains the fork and his left neighbour retains the token.

Program 5.8 Excerpt of control program for C-left (Part 3).

```
// Transition: LINKING_LEFT → LINKED_LEFT (left.order() > this.order())
int guard_3() {
 Philosopher left("Philosopher(Left::Right::surname == 'P')",...,
 REINCARNATION,...);
 return (p.test_state("PASSIVE") && p.test_state("UNLINKING_LEFT") &&
        (left.order() > p.order));
}
int action_3() {
   Philosopher left("Philosopher(Left::Right::surname == 'P')", universe,
                    REINCARNATION, p_oph);
   p.set_fork(LEFT, TRUE); p.set_dirty(LEFT, FALSE);
   p.set_state("LINKED_LEFT");
   left.set_token(RIGHT, TRUE); left.set_state("LINKED_RIGHT");
}
```

Having seen the specification and implementation of the control model for a philosopher we can continue and evaluate how communities of philosophers are created and evolve.

Creating a community of philosophers

We can create a community of philosophers using the interactive query interpreter generated by Stabilis for the diners object model. The initialization of the distributed program is made with the instantiation of philosophers and the starting up of the application and control programs. When a philosopher is first instantiated he is thinking, passive, and unlinked.

For example, the creation of a community of philosophers consisting of five philosophers involves the resolution of the queries shown in Programs 5.9 and 5.10. In line 1 (Program 5.9) philosopher P is retrieved; using similar queries we retrieve philosophers Q, R, T, and U. Then, they are associated to form a ring (Program 5.9, lines 2-6); these associations have to be in accordance with the diners structural model.

In the next step, we have to set the state of the philosophers from UNLINKED, their initial state, to LINKING_LEFT and LINKING_RIGHT. Line 7 (Program 5.9) shows the query used to promote P's state from UNLINKED to LINKING. The states of Q, R, T, and U are promoted to LINKING in a similar way. By changing the state of philosophers to LINKING we declare that they are ready to become logically linked to each other.

Program 5.9 Creating a community of philosophers (Part 1).

```
(1) > Philosopher p("Philosopher(surname == 'P')", REINCARNATION)
> ...
(2) > p.relate("Left", q)
(3) > p.relate("Right", u)
(4) > q.relate("Left", r)
(5) > r.relate("Left", t)
(6) > t.relate("Left", u)
(7) > p.put("Philosopher(state = 'LINKING_LEFT' && state =
'LINKING_RIGHT')")
> ...
```

To know if their control programs have acted and effectively made them logically linked we have to retrieve newer versions of them. To do so, we delete the instances we have retrieved initially (Program 5.10, line 8) and query the database for instances whose state is LINKED_LEFT and LINKED_RIGHT. For example, line 9 (Program 5.10) brings the query we have submitted to retrieve P; we retrieve the other philosophers in a similar way. If a certain philosopher has not yet reached the state we want then the query fails and we have to retry it. In order to simplify our examples, we assume that queries are always successful.

Program 5.10 Creating a community of philosophers (Part 2).

```
(8) > delete p q r t u
(9) > Philosopher p("Philosopher(surname == 'P'
      && state == 'LINKED_LEFT' && state == 'LINKED_RIGHT')",
REINCARNATION)
> ...
(10) > p.put("Philosopher(state = 'BOOT')")
> ...
```

Once we have established that the philosophers are linked we can activate them. We carry out their activation by setting their state to BOOT; in line 10 (Program 5.10) we do that for philosopher P. Moments later we should have a fully operational community of philosophers because their control programs will take them from BOOT to ACTIVE.

Birth of a new philosopher

To add a new object to an executing program we have, according to Kramer and Magee's reconfiguration strategy, to determine the set of objects affected

by the reconfiguration procedure and make the elements of this set quiescent. Suppose that we want to add a philosopher, say S, between philosophers R and T (Figure 5.5a). In this case, according to the reconfiguration strategy, the set of philosophers that have to be made passive is $\{Q, R, T, U\}$.

When the reconfiguration procedure is about to be started all philosophers of the community are active and going about their lives as dictated by their algorithms. Once again we use the diners query interpreter to carry out the reconfiguration procedure. We have to start by getting hold of the philosophers that have to be passivated. Line 1 (Program 5.11) does it for philosopher Q; the queries for retrieving R, T, and U are similar to this. The query shown in line 2 retrieves philosopher S, the philosopher we are going to add to the ring.

In the next step we have to passivate philosophers Q, R, T, and U. We carry out the passivation by setting their states to SHUTDOWN. In line 2 we show how we prepare philosopher Q to be passivated. Similar queries are used to passivate the other philosophers. Once we have made sure that they are passive, by retrieving versions of them where they assert their passiveness, we can proceed and unlink philosophers R and T from each other (Program 5.11, lines 4-5). In the next step we make sure philosopher R is already unlinked (Program 5.11, lines 6-7). Similarly, we have to make sure that philosopher T is unlinked. Statements of lines 8 to 10 add philosopher S to the ring by unrelating R and T and relating R to S and S to T.

Setting the state of philosophers R, S, and T to LINKING_LEFT and LINKING_RIGHT, as required, allows their control objects to proceed and complete their logical linking (Program 5.11, lines 11-13). In lines 14 and 15 we see some of the statements we have to resolve to make sure the philosophers are linked to each other as desired.

The last steps of the reconfiguration procedure simply reset the states of philosophers Q, R, S, T, and U to BOOT. The philosopher's control objects will then promote these philosophers to a fully operational state.

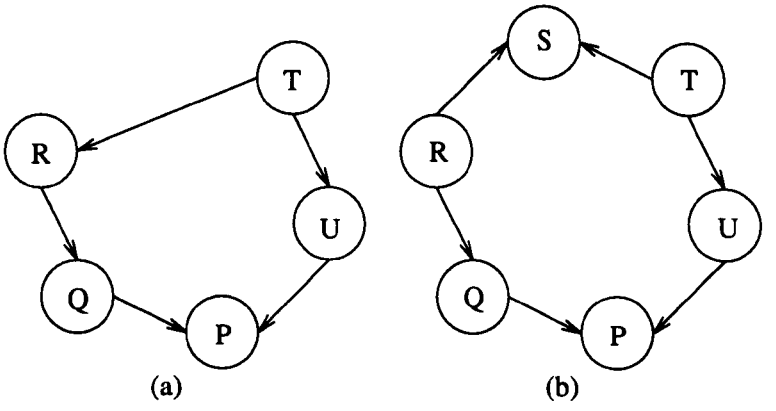

Figure 5.5: Birth of a philosopher.

Program 5.11 Birth of a philosopher.

```
(1) > Philosopher q("Philosopher(surname == 'Q' && state == 'ACTIVE')",
REINCARNATION)
> ...
(2) > Philosopher s("Philosopher(surname == 'S')", REINCARNATION)
(3) > q.put("Philosopher(state = 'SHUTDOWN')")
> ...
(4) > r.put("Philosopher(state = 'UNLINKING_LEFT')")
(5) > t.put("Philosopher(state = 'UNLINKING_RIGHT')")
(6) > delete r t
(7) > Philosopher r("Philosopher(surname == 'R' && state ==
'UNLINKED_LEFT')", REINCARNATION)
> ...
(8) > r.unrelate("Left", t)
(9) > r.relate("Left", s)
(10) > s.relate("Left", t)
(11) > r.put("Philosopher(state = 'LINKING_LEFT')")
(12) > s.put("Philosopher(state = 'LINKING_LEFT' && state =
'LINKING_RIGHT')")
(13) > t.put("Philosopher(state = 'LINKING_RIGHT')")
(14) > delete r s t
(15) > Philosopher r("Philosopher(surname == 'R' && state ==
'LINKED_LEFT')", REINCARNATION)
> ...
(16) > q.put("Philsopher(state = 'BOOT')")
> ...
```

Death of a philosopher

Suppose that we want to remove philosopher S from the community. The reconfiguration procedure necessary to remove S can be specified as this (Figure 5.6):

- passivate Q, R, S, T, U (Program 5.12, lines 1-2).
- unlink R from S; unlink T from S (Program 5.12, lines 3-8).
- remove S (Program 5.12, line 9).
- link R to T (Program 5.12, lines 10-14).
- activate Q, R, T, U (Program 5.12, line 15).

Program 5.12 Removing of a philosopher from his ring.

```
(1) > Philosopher q("Philosopher(surname == 'Q' && state == 'ACTIVE')",
REINCARNATION)
    > ...
(2) > q.put("Philosopher(state = 'SHUTDOWN')")
    > ...
(3) > r.put("Philosopher(state = 'UNLINKING_LEFT')")
(4) > s.put("Philosopher(state = 'UNLINKING_LEFT' && state =
'UNLIKING_RIGHT')")
    > ...
(5) > delete r s t
(6) > Philosopher r("Philosopher(surname == 'R' && state ==
'UNLIKED_LEFT')", REINCARNATION)
    > ...
(7) > r.unrelate("Left", s)
(8) > s.unrelate("Left", t)
(9) > remove s
(10) > r.relate("Left", t)
(11) > r.put("Philosopher(state = 'LINKING_RIGHT')")
(12) > t.put("Philosopher(state = 'LINKING_LEFT')")
(13) > delete r t
(14) > Philosopher r("Philosopher(surname == 'R' && state ==
'LINKED_RIGHT')", REINCARNATION)
    > ...
(15) > q.put("Philsopher(state = 'BOOT')")
    > ...
```

The reconfiguration procedure above ensures that R, S, and T are quiescent before S is removed. Later, during the re-linking of R and T, they can decide,

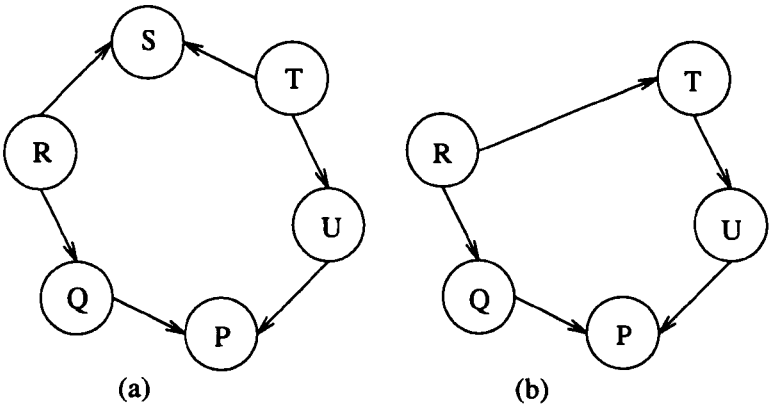

Figure 5.6: A philosopher is removed from his ring.

based on their order, how to allocate the forks they share. Suppose that R has order 6, S has order 1, and T has order 2. Then, with the removal of philosopher S (Figure 5.6), philosopher R would not retain the dirty fork shared with T because it has order greater than T's order. The specification of transitions t10 and t11 (page 5.1.3), used for linking philosophers, ensure that the precedence graph remains acyclic when philosophers are relinked, after the death of one of the members of the ring.

Merging communities of philosophers

If two communities of philosophers, as shown in Figure 5.7, are to be merged then the following reconfiguration procedure has to be executed:

- Passivate A, B, C, F, P, Q, R, and U (Figure 5.13, lines 1-5).

- Unlink B from A, A from B, Q from P, and P from Q (Figure 5.13, lines 6-11 and Figure 5.14, lines 12-14). This step breaks the connections represented as dashed arrows in Figure 5.7.

Program 5.13 Merging of two communities of philosophers (Part 1).

```
(1) > Philosopher a("Philosopher(surname == 'A' && state == 'ACTIVE')",
REINCARNATION)
      > ...
(2) > a.put("Philosopher(state = 'SHUTDOWN')")
(3) > b.put("Philosopher(state = 'SHUTDOWN')")
      > ...
(4) > delete a b p q
(5) > Philosopher a("Philosopher(surname == 'A' && state == 'PASSIVE')",
REINCARNATION)
      > ...
(6) > a.put("Philosopher(state = 'UNLINKING_RIGHT')")
(7) > b.put("Philosopher(state = 'UNLINKING_LEFT')")
(8) > p.put("Philosopher(state = 'UNLINKING_LEFT')")
(10) > q.put("Philosopher(state = 'UNLINKING_RIGHT')")
(11) > delete a b p q
```

Program 5.14 Merging of two communities of philosophers (Part 2).

```
(12) > Philosopher a("Philosopher(surname == 'A' && state =
'UNLINKED_RIGHT')", REINCARNATION)
      > ...
(13) > a.unrelate("Right", b)
(14) > p.unrelate("Left", q)
(15) > b.relate("Left", q)
(16) > a.relate("Right", p)
(17) > a.put("Philosopher(state = 'LINKING_RIGHT')")
(18) > b.put("Philosopher(state = 'LINKING_LEFT')")
      > ...
(19) > delete a b p q
(20) > Philosopher a("Philosopher(surname == 'A' && state =
'LINKED_RIGHT')", REINCARNATION
      > ...
(21) > a.put("Philosopher(state = 'BOOT')")
      > ...
(22) > u.put("Philosopher(state = 'BOOT')")
```

- Link B to Q, Q to B, A to P, and P to A (Figure 5.14, lines 15-20). This step creates the connections shown as bold arrows in Figure 5.7.
- Activate A, B, C, F, P, Q, R, and U (Figure 5.14, lines 21-22).

Once again, rules t10 and t11 (Section 5.1.3), defined for linking and unlinking philosophers, ensure that the right number of forks is properly distributed between the philosophers being relinked. It does not matter where the fork is placed since some other philosopher must have two forks. Kramer and Magee [80] argue this proposition as follows: "There are n philosophers and n forks; the two philosophers being connected have 1 fork, consequently the remaining $n-2$ philosophers have $n-1$ forks. Therefore, one of these $n-2$ philosophers must have 2 forks. The original algorithm ensures that a philosopher cannot hold a clean and a dirty fork simultaneously; consequently, the precedence graph must be acyclic."

With this example we have shown how Stabilis and Vigil can be used to implement object-oriented action-based distributed programs that are dynamically reconfigurable. As the example shows, only philosophers affected by the reconfiguration procedures had do be passivated allowing the remaining philosophers to continue with their normal operation.

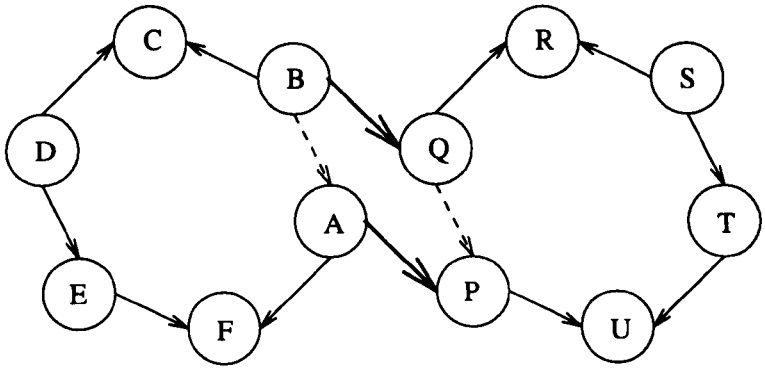

Figure 5.7: Merging communities of philosophers.

5.2 Database Index

Our second example addresses the implementation of a cache coherency protocol for a distributed index subsystem of an object-oriented distributed database management system; index subsystems are responsible for resolving queries. Cache coherency protocols guarantee that coherency is maintained between copies of data stored at different objects of the index system.

5.2.1 Specification

The index system has three subsystems arranged in a tree-based hierarchy (Figure 5.8). At the root of the tree, depth 0, we have the index subsystem; it

implements the interface between the index system and the database manager and acts as a query planner, deciding the best way to partition a query before submitting it to index managers (Figure 5.8). The index subsystem is implemented as an aggregation, or cluster, of index managers. Index managers are objects that resolve queries and function as clients to index servers, with the function of coordinating their operation during query resolution. Index managers are located at depth 1 in the tree (Figure 5.8). Index servers are autonomous servers with query processing capabilities; they are located at the leaves of the tree (Figure 5.8). To the database manager the distributed index system provides an illusion of a virtually centralized query processor (Figure 5.8, dotted-line rectangle). Additionally, to database managers the reconfiguration of index systems is transparent.

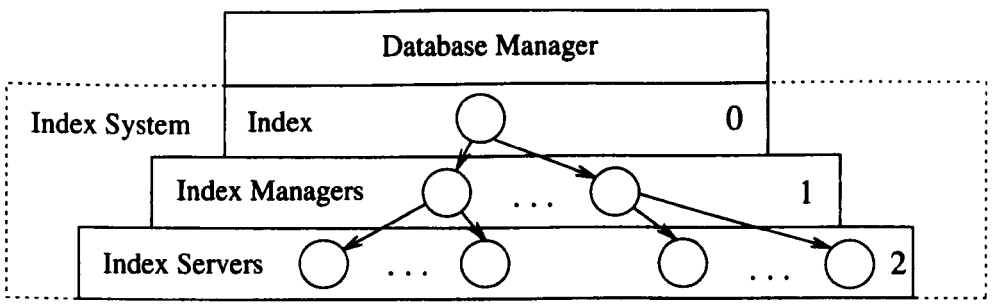

Figure 5.8: Architecture of the index system.

Some of the important requirements that the design of the index system should meet are:

- concurrency: several queries can be forwarded to an index system concurrently and a reasonable organization should maximize concurrency.
- availability: the service provided by an index system should degrade gracefully in the presence of failures.
- reconfiguration: it should be possible to substitute objects of the index system without having to stop it entirely; only the objects affected by the reconfiguration procedure should have to be stopped.

In the remaining paragraphs of this Section we study further the organization of the index system and argue that its design attends the requirements listed above.

As we have already discussed, the index is organized as an aggregation of index managers that have references to sets of index servers. An index manager keeps caches of data related to a single object model (database schema); the names

of object models are unique within a database. Thus, an index manager can be uniquely identified by the name of the object model it manages. Furthermore, we consider that names of classes are unique within an object model and that an index server only manages data related to one class, that is, all key attributes of a given class are indexed by a single index server. Consequently, an index server can be uniquely identified by the name of the class it manages.

Queries are resolved in the following way: the database manager accepts a query made by its client, e.g., an object model editor, and sends it to the index system. The index plans the execution of the distributed query using cached data stored at its index managers, although the data might be stale. Once the resolution plan is complete the index forwards subqueries to its index managers. The index managers, in their turn, resolve as much as possible of these subqueries using their caches; unresolved subqueries are forwarded to index servers. Query results propagate from index servers to index managers, where they are unified and sent back to the index subsystem. Finally, the result is sent to the database manager for delivery to the database manager's client.

The organization of the index aims at minimizing problems of concurrency, availability and reconfiguration. The association of a tree-based structure with class-specific index servers allows a reasonable level of concurrency. The tree-based architecture allows the use of independent transactions on objects belonging to independent paths of the tree, increasing concurrency. Alternative organizations are possible; instead of assigning one class to one server as we did, we could have assigned classes to index servers differently: from an index server being responsible for the management of a single or a group of attributes of a class to an index server being responsible for the indexing of several classes of an object model. After experimenting with some of these possibilities we have decided that the one-class-one-server assignment offered a reasonable balance in terms of concurrency, data encapsulation, and complexity of implementation [36].

The structure of the index and the use of caching allow index servers to be stopped for a certain period of time with the consequence that only a fraction of the information related to a given object model becomes unavailable. Thus, to a certain extent the service offered by the index system offers graceful degradation of its service in the presence of either dynamic reconfiguration or failure of index servers. Dynamic reconfiguration benefits from our cache design because index servers, the main target of dynamic reconfiguration procedures, can be replaced without causing much disruption to other subsystems of the index.

Finally, the overall performance of the index can be improved by the use of caches because index managers can optimize communication with index servers by relying on cached data to resolve queries. Ultimately, the performance of the index system depends on the combined efficiency of all algorithms used in its implementation, including the efficiency of the cache coherency algorithm.

A combination of two different policies is used by the cache coherency protocol:

1. Each index server keeps change ratios for the attribute entries it manages and determines when it is necessary to send a message to its associated index manager invalidating the manager's cached data. This will cause index managers to refresh their caches eventually.

2. The index manager monitors the load of the nodes where the index servers reside and the load of its own node. When manager and server nodes are both lightly loaded, the index manager sends a message to the server requesting a refresh. Index servers honor this request by sending up-to-date data to the requesting manager. The index system behaves opportunely and only refreshes its caches when the distributed system can afford it.

5.2.2 Structural Model

Three classes: **Index**, **IndexManager** and **IndexServer**, implement the index system. Instances of class **Index** have the role of query planners; they coordinate the workings of index managers. In order to simplify our example we do not discuss the implementation of the class **Index**, as it does not have a role in the implementation of the cache coherency protocol which is our primary focus of attention.

Instances of **IndexServer** implement index servers and instances of **IndexManager** implement index managers. In the structural model, an index system is modelled as a tight aggregation of index managers (Figure 5.9). Perhaps it is opportune to remember the meaning of a tight aggregation of objects: the existence of the component depends on the existence of the aggregate. This is the case with index managers, they have to be aggregated to an index to be able to provide their services. In other words, instances of class **Index** integrate index managers into a modular unity. Index managers are associated to a set of index servers; index servers are associated to one index manager. The structural model also shows that managers and servers are associated to a computing system (node), meaning that they are located at the node associated to each of them (Figure 5.9).

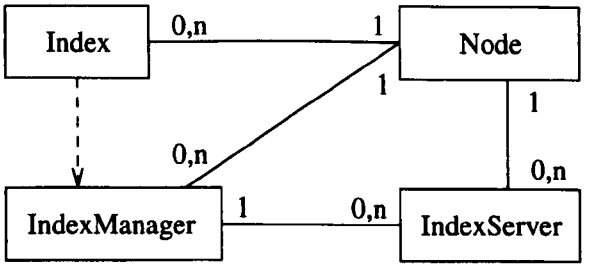

Figure 5.9: Structural model for the index system.

Our discussion of the classes IndexManager and IndexServer centres on the control aspects of the cache coherency protocol.

Class IndexServer

This class has to maintain information regarding the invalidation of the cache of its associated index manager; the attribute `stale` is used for this purpose (Class 5.2, line 5). If `stale[i]` is **true** then the value of attribute entry `i` of the index manager is out-of-date (Class 5.2, line 5). The attribute `stale` has its elements set to **true** by queries that update attribute values; conversely, its elements are set to **false** when the index server carries out a refresh of the index manager's cache.

The cache of an index manager is invalidated when a certain proportion of it has become stale in relation to the server's data. The sensor `change_ratio()` gives a measure of the divergence between the manager's cache and the server's data (Program 5.15; Class 5.2, line 9).

Program 5.15 Sensor `change_ratio`.

```
float IndexServer::change_ratio()
{
 int number_of_updated_entries = 0;
 for (int i = 0; i < number_of_entries; i++)
     if (stale[i]) number_of_updated_entries++;
 return ((float) number_of_updated_entries / number_of_entries);
}
```

In the code of the change ratio sensor (Program 5.15), the variable `number_of_updated_entries` holds the number of index entries updated since the last refresh of the index manager's cache. The attribute `number_of_entries` holds the number of index entries currently in use (Class 5.2, line 2). The value returned by

`change_ratio` is in the real interval [0.0, 1.0]. For example, if this sensor returns 0.5, it means that half of the data items managed by a server have changed since the last refresh of its manager's cache.

The sensor `ratio_limit()` (Class 5.2, line 9) returns the staleness limit over which a server should notify its associated manager that a refresh is needed. A database administrator can change the behaviour of the cache coherency protocol by altering the value returned by this sensor. If the value returned is increased then it is probably going to take longer for a server to send an invalidation message to its manager; the opposite effect is obtained if the value returned by `ratio_limit()` is decreased.

Class 5.2 Class IndexServer.

```
#define MAX_INE 50 // maximum number of data items (attributes)
class IndexServer : public Object {
 protected:
 (1) int class_name;
 (2) int number_of_entries;
 (3) float ratio_limit;
 (4) float server_load;
 (5) Boolean stale[MAX_INE];
 (6) Boolean refresh;
 (7) StateSpace state;
 public:
  // constructors deleted
 (8) int class_name(); void set_class_name(int class_name);
 (9) float change_ratio();
 (10) void refresh();
 (11) void idle();
 (12) float ratio_limit(); void set_ratio_limit(float ratio);
 (13) float server_load(); void set_server_load(float load);
};
```

The Boolean attribute `refresh` (line 6) indicates that the manager has requested a refresh of its cache; this attribute is set by actuator `refresh()` (line 8).

The sensor `class_name()` (line 8) simply returns the name of the class managed by a server as an integer; the same line shows the signature of the actuator used to update the name of the class.

Finally, sensor `server_load()` (line 10) returns a load limit in the real interval [0.0, 1.0]. The implementation of the second cache coherency policy uses the value returned by this sensor to determine when to request a refresh of the manager's

cache. If the load of the node where the server is located is less than the value returned by this sensor, then a refresh can be requested. The value returned by this sensor can be set by a database administrator to fine tune the operation of the index system.

Class IndexManager

The sensor `model()` (Class 5.3, line 4) returns the value of the attribute `model_name` (line 1);

The sensor `manager_load()` (line 5) has a function analogous to the function of sensor `server_load()` of an index server; it returns the the load limit used to decide when to trigger refreshes.

This class has to keep track of invalidation messages sent by its associated servers. It uses an array of state spaces indexed by the name of the server's class to record such information (Class 5.3, line 3). We have an independent state space for each server associated to an index manager; consequently, we have as many independent server-specific control objects controlling a manager as the number of index servers associated to it.

Class 5.3 Class IndexManager.

```
#define MAX_SVR 50 // maximum number of servers
class IndexManager : public Object {
 protected:
 (1) int model_name;
 (2) float manager_load;
 (3) StateSpace state[MAX_SVR];
 public:
  // constructors deleted
 (4) int model(); void set_model(int model);
 (5) float manager_load();
 (6) Boolean idle(int server);
};
```

Class Node

This class has a sensor `load()` that returns the average load of a node of a distributed system as a real in the interval $[0.0, 1.0]$; a return value 0.0 means that the node is idle, a value 1.0 means that the node is running at its full processing capacity. This sensor is used by the control program of index managers to determine when to request a refresh of a cache.

This class is an example of a class whose function is to be an interface between services that are not implemented within the object and action computation model and systems that are implemented within this computation model. Classes like this have the important function of allowing distributed programs developed in Arjuna to interact with programs that were developed using different computation models.

5.2.3 Control Model

In this Section we present the cache coherency algorithm employed by the index system. Before proceeding any further, we should emphasize that the activity of refreshing the manager's cache is carried out at application level; it can involve the transmission of considerable volumes of data in the form of object states. At control level, we are interested in controlling when refresh requests happen but not how the refresh is realized. We consider that refresh requests triggered by the control program of the index system are honored dependably by its application program.

Index Server

The application objects of an index server are constantly monitored by their corresponding control objects to ensure that the associated manager is notified of the staleness of its cache.

The control model for index servers, with states: GATHER and NOTIFY and transitions t1 and t2 is shown in Figure 5.10. A server is in state GATHER while the change ratio has not reached the limit over which it is necessary to invalidate the managers' cache. When the change ratio reaches a specified limit, then transition t1 (Figure 5.10) is fired, sending an invalidation message to the manager. A server is in state NOTIFY if it has just notified its associated manager that a refresh is necessary.

The transition from NOTIFY to GATHER, transition t2, happens when the application program of the server carries out the refresh of the manager's cache (Figure 5.10). At the application program, transition t2 is implemented in two steps: first, the index server waits for a refresh request to arrive, attribute **refresh** becomes **true**; second, it sends the update requested by its manager and moves from NOTIFY to GATHER.

The specification of transition t1 is shown below; the code generated by Stabilis for this transition is shown in Program 5.16.

(t1) Invalidating manager caches:

```
(change_ratio() > ratio_limit())/
{(associated_manager).set_state("STALE", class_name());}
```

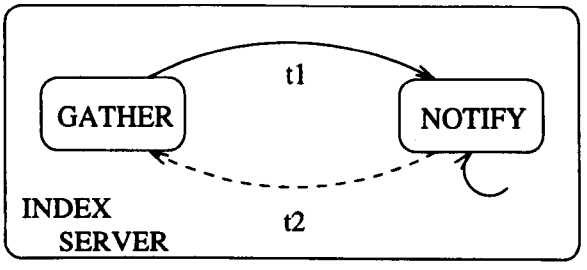

Figure 5.10: Index server control model.

Program 5.16 Implementation of index server transition t1.

```
/* automatically generated code */
/* IndexServer's class name = C */
IndexServer is("IndexServer(class_name == C)",, REINCARNATION,);
int guard_1() {
return (is.test_state("GATHER") && is.change_ratio() > is.ratio_limit()); }
int action_1() {
IndexManager im("IndexManager(IndexServer::class_name == C)",,
REINCARNATION,);
im.set_state("STALE", is.class_name()); is.set_state("NOTIFY"); }
```

Index Manager

The index manager's control model has three states: UPTODATE, STALE and REFRESH (Figure 5.11). A manager is in state UPTODATE if its cache is coherent with the data stored at an index server, that is, it has just received an update for its cache. A manager is in state STALE if it has received a cache invalidation message from an index server. Finally, the state REFRESH means that a manager has already requested a refresh of its cache but has not received it yet.

Transition t1 (Figure 5.11) is represented as a dashed-line arc to show that it is not implemented by the index manager's control program; this transition is executed by the action part of transition t1 of the index server's control model (Figure 5.10, Program 5.16).

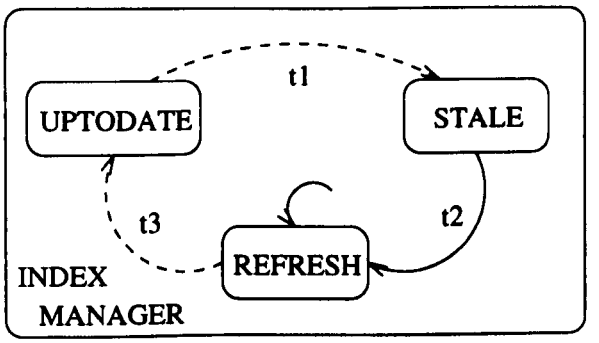

Figure 5.11: Index manager control model.

Transition t2 (Figure 5.11, Program 5.17) implements the second policy of the cache coherency protocol: when the load of the manager's node is less than the value returned by the manager's load sensor and the load of the server's node is less than the value returned by the server's load sensor, then the manager requests a refresh to the index server.

(t2) Requesting a cache update:

```
((⟨manager_node⟩.load() < manager_load())∧
(⟨server_node⟩.load() < ⟨server⟩.sever_load()))/
{⟨server⟩.refresh();}
```

Program 5.17 Implementation of index manager transition t2.

```
/* IndexManager's model name = M */
/* IndexServer's class name = C */
IndexManager im("IndexManager(model_name == M)",, REINCARNATION,);
int guard_1() {
Node mn("Node(IndexManager::model_name == M)",, REINCARNATION,);
Node sn("Node(IndexServer::class_name == C)",, REINCARNATION,);
IndexServer is("IndexServer(IndexManager::model_name == M && class_name
== C)",, REINCARNATION,);
return (im.test_state("STALE", is.class_name()) && mn.load() < im.manager_load()
&& sn.load() < is.server_load()); }
int action_1() {
IndexServer is("IndexServer(IndexManager::model_name == M && class_name
== C)",, REINCARNATION,);
is.refresh(); im.set_state("REFRESH", C); }
```

Transition t3 is carried out at application level; a manager goes from state REFRESH to UPTODATE as soon as it receives the cache update it has requested from an index server.

5.2.4 A Reconfigurable Index

So far, the design of the index system has not taken into account dynamic reconfiguration; in this Section we extend it to include reconfiguration mechanisms.

Index initialization

When the index system is first instantiated all its objects are passive; objects might already have been related to each other at database level. A passive index does not accept query requests and has all its indexing data structures reset. Subsequently, during activation, the index loads all data it needs to serve queries and becomes operational. Once activated, an index accepts queries submitted by database managers and can have one or more of its active subsystems dynamically reconfigured. In this case, only the subsystems affected by the reconfiguration are passivated while the remaining subsystems continue in operation.

Reconfigurable Index Server

Activation. At their instantiation, index servers load data of the class they manage into their indexing data structures; their initial state is NOTIFY and PASSIVE (Figure 5.12). Immediately after the load, index servers send a refresh to their associated index manager; having done it, they move from NOTIFY to GATHER (Figure 5.12). All that remains to be done in order to activate the servers is to promote them from PASSIVE to ACTIVE. First, using a query interpreter, we take them from PASSIVE to BOOT; next, their control programs take them from BOOT to ACTIVE.

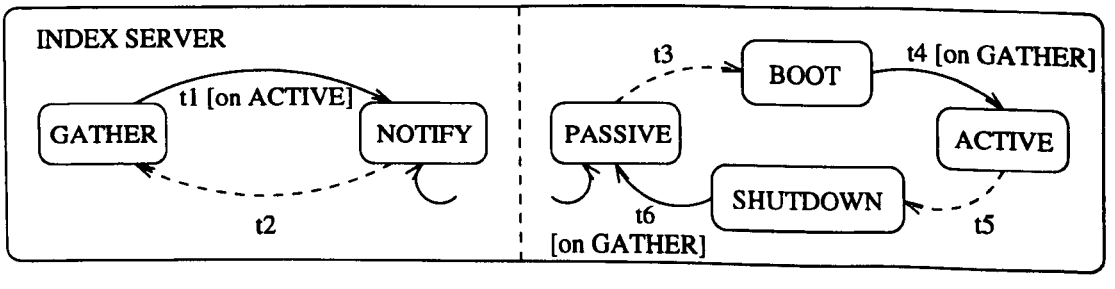

Figure 5.12: Extended index server control model.

The specification of transition t4 (Figure 5.12) is: [on GATHER]/{}; its code is shown in Program 5.18.

Program 5.18 Implementation of index server transition t4.

```
/* automatically generated code */
/* IndexServer's class name = C */
IndexServer is("IndexServer(class_name == C)",, REINCARNATION,);
int guard_1() {
return (is.test_state("BOOT") && is.test_state("GATHER"));
}
int action_1() { is.set_state("ACTIVE"); }
```

Passivation. An active server can be passivated when it is in the state GATHER because in this state it is not going to become engaged in a refresh operation.

A passive index server stops processing queries; additionally, it stops notifying index managers that their cache is stale. Passive servers no longer process refresh requests.

Transition t6 of the extended control model (Figure 5.12) is specified as: (idle())[on GATHER]/{}; its code is shown in Program 5.19.

Program 5.19 Implementation of index server transition t6.

```
/* automatically generated code */
/* IndexServer's class name = C */
IndexServer is("IndexServer(class_name == C)",, REINCARNATION,);
int guard_1() {
return (is.test_state("SHUTDOWN") && is.test_state("GATHER") && is.idle());
}
int action_1() { is.set_state("PASSIVE"); }
```

Reconfigurable Index Manager

Activation. When instantiated an index manager is in state REFRESH and PASSIVE, meaning that it is waiting for an update of its cache with data supplied by an index server. The first update allows the manager to assert to the index subsystem that it is ready to accept queries. The manager becomes fully operational only when all of its servers have been able to send cache updates to it.

Passivation. An index manager can be passivated gradually by setting each of the manager's control objects to PASSIVE. The index manager involved in the reconfiguration can continue resolving queries, but it has to stop querying the index server that is going to be replaced. Queries involving attributes of the class indexed by the server that is being replaced can still be solved using cached data, but they might return results with smaller accuracy due to cache staleness. Further, the index manager has to stop requesting refreshes to the server whose replacement is being carried out.

The extended control model of the class **IndexManager** is shown in Figure 5.13. In a reconfigurable index, transition t2 (Figure 5.13) is allowed to fire only when the index is active.

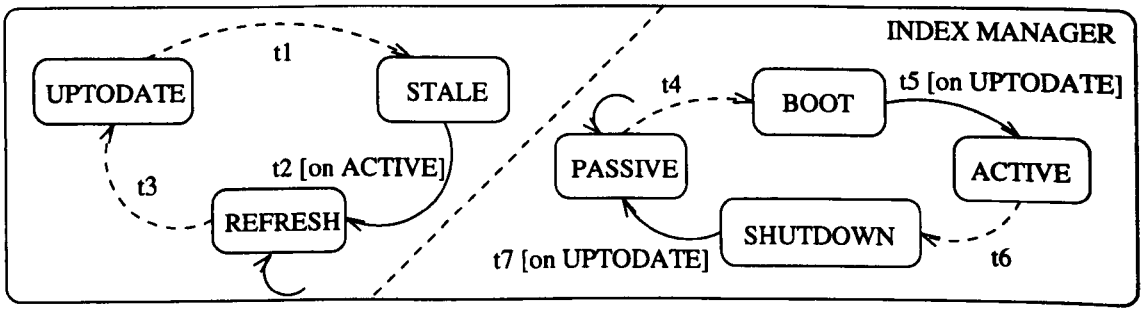

Figure 5.13: Extended index manager control model.

Let us define transitions t5 and t7 of the extended index manager control model. Transition t5 is specified as:

(t5) Transition from BOOT to ACTIVE: [on UPTODATE] / {}

(t7) Transition from SHUTDOWN to PASSIVE: [on UPTODATE] `(idle())` / {}

As with class **IndexServer**, a sensor `idle()` has been added to the index manager's interface to allow the specification of transition t7 (Figure 5.13); this sensor returns **true** when the index manager is not processing a query.

Excerpts of code of transitions t5 and t7 are shown in Programs 5.20 and 5.21, respectively.

Program 5.20 Implementation of index manager transition t5.

```
/* automatically generated code */
/* IndexManager's model name = M */
/* IndexServer's class name = C */
IndexManager im("IndexManager(model_name == M)",, REINCARNATION,);
int guard_1() {
return (im.test_state("BOOT", C) && im.test_state("UPTODATE", C)); }
int action_1() { im.set_state("ACTIVE", C); }
```

Program 5.21 Implementation of index manager transition t7.

```
/* automatically generated code */
/* IndexManager's model name = M */
/* IndexServer's class name = C */
IndexManager im("IndexManager(model_name == M)",, REINCARNATION,);
int guard_1() {
return (im.test_state("SHUTDOWN", C) && im.test_state("UPTODATE", C) &&
im.idle(C)); }
int action_1() { im.set_state("PASSIVE", C); }
```

Assembling an Index System

The initial assembly entails the creation of the objects that compose an index system using their birth constructors; these objects are created at the nodes designated to them by the database administrator. Objects of class **Node** already exist; they have been created during the installation of the distributed system. The database administrator must then associate each of the newly created objects to their respective nodes, that is, to the instances of class **Node** that represent the computing systems of the distributed system. All relationships created must conform to the structural model of the index system.

The next assembly phase begins with the reincarnation of the index manager and of one of the index servers created before (Program 5.22, lines 1-2). In line 3, the index manager and index server are related to each other. At this point two control objects have their execution started: the control object of the manager that has been generated for server `isa`, class A, and the control object of index server `isa`. The queries executed in lines 4 and 5 of the procedure activate `im` and `isa`. In line 4 we can see how the state space of class **IndexManager** is used; there is an independent state space for each server associated to an index manager.

The index server **im** is already capable of resolving queries that only involve attributes of class A. In lines 6 to 9, we reincarnate the index server that manages attributes of class B, relate it to the manager **im**. We can now start the control object of the manager generated for server **isb**, and the control object of index server **isb**.

To verify that the configuration process carried out so far has been successful the configuration procedure deletes the current instantiation of the objects and reincarnates them once again but querying for active objects (Program 5.22, lines 11-13). If any of these queries fail, then there may be something wrong with the configuration process or some other cause prevented a control object from activating its application object, for example, a heavily loaded node.

Program 5.22 Assembling an index system.

```
(1) > IndexManager im("IndexManager(model_name == M)",,REINCARNATION,)
(2) > IndexServer isa("IndexServer(class_name == A)",,REINCARNATION,)
(3) > im.relate("IndexServer", isa)
(4) > im.put("IndexManager(state[A] = 'BOOT')")
(5) > isa.put("IndexServer(state = 'BOOT')")
(6) > IndexServer isb("IndexServer(class_name == B)",,REINCARNATION,)
(7) > im.relate("IndexServer", isb)
(8) > im.put("IndexManager(state[B] = 'BOOT')")
(9) > isb.put("IndexServer(state = 'BOOT')")
(10) > delete im isa isb
(11) > IndexManager im("IndexManager(model_name == M && state[A] ==
'ACTIVE' && state[B] == 'ACTIVE')",, REINCARNATION,)
(12) > IndexServer isa("IndexServer(class_name == A && state ==
'ACTIVE'),, REINCARNATION,)
(13) > IndexServer isb("IndexServer(class_name == B && state ==
'ACTIVE'),, REINCARNATION,)
```

An evolving Index System

One of the major advantages of an object-oriented approach to designing and implementing software systems is that the concepts of generalization/specialization and inheritance allow existing distributed programs to be extended. In an integrated database environment where many users share a single database, different users often need to view object models differently. This means that the users need to modify object models; further, it is highly desirable to be able to modify dynamically an object model without forcing a system shutdown or incurring significant performance penalty. Once the mechanisms for object model evolution

(modification) are in place then it is fundamental to have indexing subsystems that can be configured dynamically to reflect the changes made to object models. The index subsystem discussed in this example has been designed to support object model evolution.

Suppose we have evolved class A by adding a new attribute to it and want to evolve the index assembled in the previous Section to reflect this change; the configuration procedure presented below does exactly this.

The procedure passivates the part of the index manager that indexes class A and its associated index server (Program 5.23, lines 1-7), unrelates them (line 8), and removes the index server from the system (line 9). A updated index server is created for class a (line 10), related to the index manager (line 11). Finally, both subsystems are made active.

Program 5.23 An evolving an index system.

```
(1) > IndexServer isa("IndexServer(class_name == A)",, REINCARNATION,)
(2) > isa.put("IndexServer(state = 'SHUTDOWN')")
(3) > IndexManager im("IndexManager(model_name == M)",, REINCARNATION,)
(4) > im.put("IndexManager(state[A] = 'SHUTDOWN')")
(5) > delete isa im
(6) > IndexServer isa("IndexServer(class_name == A && state ==
'PASSIVE')",, REINCARNATION,)
(7) > IndexManager im("IndexManager(model_name == M && state[A] ==
'PASSIVE')",, REINCARNATION,)
(8) > im.unrelate("IndexServer", isa)
(9) > remove isa
(10) > IndexServer isa("IndexServer(class_name = A)",, BIRTH,)
(11) > im.relate("IndexServer", isa)
(12) > im.put("IndexManager(state[A] = 'BOOT')")
(13) > isa.put("IndexServer(state = 'BOOT')")
```

5.3 Conclusions

We began this Chapter by making a brief account of the experiences we had during the implementation and execution of the examples we have just described.

The design of any distributed program implemented using Stabilis and Vigil is carried out in two main phases:

1. the creation of the structural model, where the arrangements, or structure, of the objects of the application is captured;

2. the creation of the control model, where the control aspects of the algorithm of the distributed program are captured. In this phase the user has to have a clear view that he is developing a reactive program composed of two subprograms: an application program and a control program. The examples shown in this Chapter illustrate this point, the design and implementation of the Evolving Dining Philosophers and of the Database Index involved the construction of a distributed program with the form $DP :: [CP \| AP]$, where DP is, for example, the Evolving Philosophers; CP is the control program obtained from the instantiation of the dines control model, and AP is the program written by the user during the implementation of class **Philosopher**.

The implementation of an application includes the following steps:

1. representation of the object model (structural model and control model) as a database schema. This step involves the writing of programs that instantiate metaobjects for each of the classes of the object model devised for the application. We have discussed these steps in detail in Chapters 3 and 4.

2. to make use of the code generation facilities of Stabilis the user has to write a program that reincarnates the object that is the root of the application object model and sends it a message that triggers the code generation. The generation of the code of the classes is carried out first and results in the writing of the header files and templates of code files for the application. During this phase Stabilis automatically generates an application-specific interactive query interpreter that later is used not only as query interpreter but also as a simple configuration manager. At this point, we should emphasize that the use of a query interpreter as a configuration manager is a positive evidence that our thesis has been demonstrated, that is, for object-oriented action-based distributed systems the run-time management of distributed programs can be considered an information processing activity.

3. the implementation of the application program is carried out using the files generated by Stabilis from the structural model. The code of the application program is compiled and linked to the libraries of Arjuna and Stabilis.

4. objects are created, using the constructors with argument BIRTH, and related according to the structural model. Once this has been done, the user can execute the second phase of code generation, that is, he writes a program that reincarnates the object that is the root of the object model, the application objects, and triggers the generation of the code of the control programs. The control programs have to be then compiled and linked to object libraries of Arjuna, Stabilis, and Vigil.

At this point the user can unrelate objects if he wants to, later during the execution of the application he can relate them again, this time the relationships created are a function of the run-time needs of the application. For example, during the preparation of the Database Index we created one index manager and four index servers. We had to relate them in order to use the code generation facilities of Stabilis. Prior to the execution of the program we had the objects of the index system unrelated so that we could experiment with their dynamic reconfiguration. We started the execution of the database index with one manager and two servers running, later we added the other two.

The execution of the application entails: running the application and control objects. For each node where the application is being executed we have to start Arjuna's run-time system and then the application and control objects. From this point onwards we can use the query interpreter to reconfigure our application; as the examples have shown.

Having given an account of the procedure followed to program and run applications in the environment we have created, we can discuss briefly what happened during the execution of the two examples shown in this Chapter. Both applications were executed using networked Unix workstations. Our tests did not make use of the replication mechanisms provided by Arjuna and used objects with small object state sizes, that is, less than a megabyte. The object states of the philosophers are inherently small. The object states used for index servers were kept small by using classes with a small number of simple attributes like integers and strings. If Stabilis and Vigil have had to manipulate objects with large state sizes, say several megabytes, then the performance would probably have degraded, as the present implementation of the communication subsystem of Arjuna is not well-suited for transferring large object states.

Application and control objects of both programs have to resolve queries in order to instantiate the object they manipulate. For example, the client that implements the application object of a philosopher has to execute a query in order to obtain a reference to the database object it manipulates. The execution of the query is processed by Stabilis and involves the activation/deactivation of persistent objects; the average elapsed time taken by Stabilis to resolve a query was approximately 6s; this is an average of the query times of both examples. In the same way, the control object used queries during the firing of transitions. The average time taken to fire a transition was approximately 8s; therefore, control objects were firing an average of 7 transitions per minute. The factors affecting the average times are examined below:

1. the resolution of a query handles large number of remote objects. Arjuna implements a simple scheme for accessing remote objects [114]: top-level actions access remote objects through independent single-threaded servers (processes). In the implementation of Stabilis we have minimized the perfor-

mance problem created by the use of independent single-threaded servers by implementing a server multiplexing scheme, but multiplexing alone did not give us the performance improvement we expected. Ideally, we would like a future version of Arjuna to have multi-threaded and multiplexed servers.

2. the current object store of Arjuna is not well-suited for database applications as it does not allow a program to gain access to parts of the state of an object independently. An alternative solution can be tried and we have done so in the implementation of the object manager and indexing structures of Stabilis, but this is not the best solution as our indexing structures had inevitably to be stored in Arjuna's object store. In an ideal situation, we would like Arjuna to have an object store with capabilities close to those found in the object stores of database management systems like GemStone and O_2; in these systems the object stores can directly and independently manipulate parts of objects. For database applications like ours the time taken to retrieve the state of an object during the activation/deactivation is crucial due to the number of accesses performed to resolve queries; this fact alone explains the times we have obtained.

The examples have provided us with evidence that supports the thesis that management is fundamentally an information processing activity and that the *object model*, as applied to *action-based* distributed systems and *database systems*, is an appropriate representation of the management information. Resolution of queries on the distributed program's object model enable the management system to control activities of distributed programs. Certainly, the runtime management of object-oriented action-based distributed programs can be carried out using the information processing facilities provided by Stabilis and Vigil.

The implementation of the examples has been greatly simplified by the fact that Stabilis can generate part of the code of the distributed program automatically; although the current version of Stabilis does not generate the code of the transition systems automatically.

During the implementation of Stabilis and Vigil we experienced problems only when we tried to tailor some of the tools provided by Arjuna to our specific needs. For example, the implementation of the object manager of Stabilis took much longer than we expected because we had to implement a server multiplexor for it.

Although the overall performance we have observed during the use of the management system was not as good as we expected, it should be stressed that there are several areas of the system that can be improved to increase performance and that these improvements do not affect the basic system architecture.

Conclusions

This thesis has addressed the problem of runtime management of object-oriented action-based distributed programs using an object-oriented active database system. In this thesis, the management of distributed programs is viewed as an information processing activity. Object-orientation and transactions have provided us with the model we needed to represent and process structural and control information about distributed programs. Processing of structural and control information through the use of queries allows programmers to control activities of their distributed programs.

In this Chapter we summarize the contributions of this work, pointing out directions for future research.

Thesis Contributions

The main contributions of this thesis can be summarized as follows:

- Identification and integration of a set of principles and techniques that allow the management of the runtime behaviour of object-oriented action-based distributed programs to be viewed as an information processing activity. The survey presented in Chapter 2 provided much of the insights we have used in the synthesis of concepts that lead to the formulation of the thesis. That survey demonstrates that software systems are becoming more and more alike, especially those whose design and implementation is based on object-orientation.

- Implementation of a management system based on the ideas of the thesis. This management system is composed of two subsystems, Stabilis and Vigil, that were the result of a design based on our theory. In Chapters 3 and 4 we have discussed their design and implementation. Object-oriented programming environments usually lack mechanisms that allow programmers to represent and manage the configuration (composition) of objects of a distributed program. Stabilis and Vigil make a contribution towards complementing object-oriented programming environments with support for management of metainformation regarding object-oriented programs.

- Active database systems are usually implemented from scratch and in a monolithic way, that is, all mechanisms used to implement indexes, persistence, atomic actions, and distribution are tightly coupled and, therefore, very difficult to modify or extend. Our approach represents a contribution to the area of extensible active databases; instead of providing programmers with a monolithic tool we have decided to provide them with a set of tools that can be used independently. All features of Arjuna, Stabilis, and Vigil

are equally available to programmers. Therefore, programmers can choose the combination of tools that best fit their needs.

Stabilis/Vigil: A Concise History

The implementation of Stabilis and Vigil was started in August, 1991. In April, 1992 we had finished the implementation of the first version of Stabilis. This version provided very simple queries, only attributes of a single class were allowed in the specification of query predicates using a window-based query interpreter, implemented using InterViews. The user could create, retrieve and relate objects using the query interpreter. In this version, object models were not represented using database objects, that is, metainformation was stored outside the domain of Stabilis. The first version of the algorithms for automatic code generation were implemented in this version.

For the first version we had to develop a tailor-made name server and a cached object manager. Using this implementation we created a distributed bibliography database, Dbib, and were able to manage a distributed database of around 500 references [36]. The tests with Dbib revealed some of the problems of the current implementation of Arjuna; they have already been discussed in the previous Chapter.

From August 1992 to December 1993 we implemented the second version of Stabilis, designed and implemented Vigil, and carried out tests of both systems. In the second version of Stabilis queries can be formulated against any portion of an object model and there is a good support for navigational queries. The object manager has been much improved, with the addition of caching and multiplexing of servers.

The current version of Vigil gave us the minimum capabilities necessary to verify our ideas. In this version, metainformation concerning control models is stored not as database objects but as text; information, i.e., transition systems, is stored as executable programs. Two simple schedulers of guarded action have been implemented but new schedulers can easily be added to Vigil using inheritance.

A problem we faced during the implementation of these systems was the lack of support in C++ for program access to metainformation. As a result, we had to design and implement our own structures and algorithms for manipulation of metainformation.

Future Work

During the development of Stabilis and Vigil a number of areas of future work have become apparent; they are described in this Section.

Although the current implementation of Arjuna uses a simple object store that provides primitive support for database applications like the ones we have developed, we believe that our work has demonstrated that it is possible to implement database applications on top of a system based on explicit transactions.

One of the interesting features of Arjuna is that unlike most of the systems studied in Chapter 2, for example, Camelot/Avalon (Section 2.3.5), Guide (Section 2.1.6), and Argus (Section 2.3.3), it did not modify the language or the operating system. Therefore, Arjuna is a system that is more easily accepted by programmers and widely available. In keeping with this design policy we have implemented a system that uses object-oriented models to represent explicitly information about distributed programs and standard C++. Object-oriented modeling is an inherent part of object-oriented programming activity, therefore, we have chosen them as the representation medium for the explicit representation of the structural and control aspects of object-oriented programs. Future work has certainly to address the use of object models to represent other aspects of distributed applications. For example, we can try to extend object models so that the specification of relationships maintains information about the type of communication protocol used by objects. An aggregation relationship could, for instance, specify if the communication between the aggregate object and its components should be implemented using RPCs or messages. Darwin/Regis already offers facilities for explicit specification of the communication protocol adopted by objects (Section 2.3.2).

Object model evolution (schema evolution) is an interesting area of future research. In our system we use object models to represent the structure of distributed programs, so object model evolution can be used to keep track of the various versions of distributed programs.

Our management system would greatly benefit from a more flexible object store. For example, we could investigate ways of adding to Arjuna an object store with support for B-trees. A very desirable attribute of such an object store would be the ability for manipulating executable code. B-trees would probably improve the performance of the system as a whole because smaller units of data could then be manipulated during activation/deactivation of objects; storing executable code as objects would allow us to implement a version of Vigil where transitions systems are maintained as database objects.

Further study of algorithms for transforming state machine specifications into programs is needed. We have developed and are implementing simple algorithms for this purpose but we are aware that tools like STATEMATE [69] have a much

better support for automatic generation of programs from STATECHART specifications.

During this work we have seen that the splitting of the algorithm of a distributed program into management and application parts is not always easy; it is a decision that is left to the designer of the distributed program. Future research has to explore better ways of approaching this problem in order to simplify the task of designing and implementing reconfigurable distributed programs.

Research in integration of graphical interfaces and automatic generation of code can be interesting. The integrated use of these facilities should help reduce the time taken to develop distributed programs, from design to coding and testing. Users of the system will benefit from a tool to assist the validation and debugging of object models. A complete management environment would have an object model editor, a graphical query interpreter, and a configuration manager.

Interoperability is another interesting issue for future research. We can study ways of creating bridges between the management system we have implemented and other management systems. It would be interesting to explore ways of making environments like Darwin/Regis (Section 2.3.2), ISIS/Meta (Section 2.3.1), and Stabilis/Vigil to co-operate.

Stabilis and Vigil have reflective architectures, that is, both systems have structural and functional attributes that make them good candidates for experimenting with reflection. Reflection allows the implementation of a system to be exposed to programmers in a controlled way. Programmers have access and can redefine the behaviour of certain parts of a reflective system. It would probably be very interesting to have Arjuna, Stabilis and Vigil running in a system like Apertos (Section 2.1.8). Stabilis and Vigil could be transformed into a metaspace of Apertos. In this way, if a programmer wanted to have access to management facilities he could associate the objects of his program with the metaspace provided by Stabilis and Vigil.

In Conclusion

Perhaps it is opportune to recall the metaphor of the orchestra we have introduced in Chapter 3 for the description of Stabilis. There, we wrote that the management system could be seen as being a concert. Users, which we considered as the audience, only got to see what was played on-stage; we considered the programming interface as being on-stage. The realization of the concert, with all its instrument tuning and rehearsals, was backstage and supported what users saw on-stage. Finally, we described the implementors of the management system as being the maestros and musicians because they got to see both what happened on-stage and backstage; they were responsible for the concert. Our

metaphor describes accurately our intention with this project. From the outset we were concerned with creating a flexible and simple environment where users were not only the audience, but also maestros. In Arjuna, Stabilis, and Vigil the programmer is invited to orchestrate, i.e., to adapt the system to his own liking, provided he is prepared to respect a set of rules that guarantee the consistency not only of the management system, but also of the distributed program he is implementing.

In conclusion, the design, implementation and management of distributed programs is a complex and error prone task that must be executed correctly if reliable programs are expected as the outcome. This thesis makes a contribution to the area of distributed programming using objects and actions by explaining how to use object orientation, transactions, database management system techniques, and reconfiguration management techniques to the advantage of the distributed programming task.

References

[1] M. Accetta, R. Baron, D. Golub, R. Rashid, A. Tevanian, and M. Young. Mach: A New Kernel Foundation for UNIX Development. In *Proceedings of Summer USENIX Conference*, pages 93–112, Atlanta, GA, USA, 1986.

[2] R. Agrawal, N. Gehani, and J. Srinivasan. OdeView: The Graphical Interface to ODE. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 37–43, Atlantic City, NJ, USA, May 1990.

[3] R. Agrawal and N. H. Gehani. ODE (Object Database and Environment): The Language and the Data Model. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 36–45, Portland, OR, USA, May-Jun. 1989.

[4] R. Agrawal and N. H. Gehani. Rationale for the Design of Persistence and Query Processing Facilities in the Database Programming Language O++. In *Second International Workshop on Database Programming Languages*, pages 25–40, Portland, OR, USA, June 1989.

[5] A. Albano, G. Ghelli, and R. Orsini. The Implementation of Galileo's Persistent Values. In M. P. Atkinson, P. Buneman, and R. Morrison, editors, *Data Types and Persistence*, pages 253–263. Springer-Verlag, 1988.

[6] G. T. Almes, A. P. Black, E. D. Lazowska, and J. D. Noe. The Eden System: A Technical Review. *IEEE Transactions on Software Engineering*, SE-11(1):43–59, Jan. 1985.

[7] P. Alsberg and J. Day. A Principle for Resilient Sharing of Distributed Resources. In *Proceedings of the Second International Conference on Software Engineering*, pages 562–570, San Francisco, CA, USA, Oct. 1976.

[8] T. Anderson and P. A. Lee. Software Fault Tolerance Terminology Proposals. In Shrivastava [125], pages 6–13.

[9] M. P. Atkinson, P. J. Bailey, K. J. Chisholm, W. P. Cockshott, and R. Morrison. An Approach to Persistent Programming. *Computer Journal*, 26(4):360–365, Dec. 1983.

[10] M. P. Atkinson, K. J. Chisholm, and W. P. Cockshott. PS-Algol: an Algol with a Persistent Heap. Technical Report CSR-94-81, University of Edinburgh, Department of Computer Science, Dec. 1981.

[11] M. P. Atkinson et al. Object-Oriented Database System Manifesto. In *Proceedings of the First International Workshop on Deductive and Object-Oriented Databases*, Kyoto, Japan, Dec. 1990. North-Holland.

[12] H. Bal, M. Kaashoek, and A. Tanenbaum. Experience with Distributed Programming in Orca. In *Proceedings of the International Conference on Computer Languages*, Atlantic City, NJ, USA, 1990. In The Amoeba Distributed Operating System, Technical Report, Vrije Universiteit, The Netherlands, pages 218-239, 1990.

[13] H. E. Bal, J. G. Steiner, and A. S. Tanenbaum. Programming Languages for Distributed Computing Systems. *ACM Computing Surveys*, 21(3):261–322, Sept. 1989.

[14] H. E. Bal and A. S. Tanenbaum. Orca: A Language for Distributed Object-Based Programming. *ACM SIGPLAN Notices*, 25(5):17–24, May 1990.

[15] R. Balter, J. Bernadat, D. Decouchant, A. Duda, A. Freyssinet, S. Krakowiak, M. Meysembourg, P. L. Dot, H. N. Van, E. Paire, M. Rivelli, C. Roisin, X. R. de Pina, R. Scioville, and G. Vanôme. Architecture and Implementation of Guide, an Object-Oriented Distributed System. *ACM Computing Systems*, 4(1):31–68, Winter 1991.

[16] P. Bates. Debugging Heterogeneous Distributed Systems Using Event-Based Models of Behavior. In *Proceedings of Workshop on Parallel and Distributed Debugging*, pages 11–22. ACM, Wisconsin Center, University of Wisconsin, USA, May 1988.

[17] P. Bates and J. Wileden. An Approach to High-Level Debugging of Distributed Systems. *ACM SIGPLAN Notices*, 18(8):107–111, Aug. 1983.

[18] P. Bates and J. Wileden. High-Level Debugging of Distributed Systems: The Behavioral Approach. *Journal of Systems and Software*, 4(3):255–264, Mar. 1984.

[19] D. S. Batory et al. GENESIS: An Extensible Database Management System. In Zdonik and Maier [147], pages 500–518.

[20] M. Benveniste and V. Issarny. Concurrent Programming Notations in the Object-Oriented Language Archie. Technical Report 1822, Institut National de Recherche en Informatique et en Automatique (INRIA), France, Dec. 1992.

[21] B. Bhargava, Y. Zhang, and E. Mafla. Evolution of a Communication System for Distributed Transaction Processing in Raid. *Computing Systems*, 4(3):277–313, Summer 1991.

[22] K. Birman. ISIS: A System for Fault-Tolerant Distributed Computing. Technical Report TR 86-744, Department of Computer Science, Cornell University, Ithaca, NY, USA, Apr. 1986.

[23] K. Birman, T. Joseph, and F. Schmuck. *ISIS-A Distributed Programming User's Guide and Reference Manual.* The ISIS Project, Department of Computer Science, Cornell University, Ithaca, NY, USA, Mar. 1988.

[24] K. P. Birman. The Process Group Approach to Reliable Distributed Computing. *Communications of the ACM*, 36(12):37–53,103, Dec. 1993.

[25] K. P. Birman and T. A. Joseph. Exploiting Virtual Synchrony in Distributed Systems. In *Proceedings of the 11th ACM Symposium on Operating Systems Principles*, pages 123–138, Austin, TX, USA, Nov. 1987.

[26] K. P. Birman, T. A. Joseph, and F. Schmuck. Isis Documentation: Release 1. Technical Report 87-849, Department of Computer Science, Cornell University, Ithaca, NY, USA, July 1987.

[27] A. D. Birrel and B. Nelson. Implementing Remote Procedure Calls. *ACM Transactions on Computing Systems*, 2(1):39–59, Feb. 1984.

[28] A. Black, N. Jutchinson, E. Jul, H. Levy, and L. Carter. Distribution and Abstract Types in Emerald. *IEEE Transactions on Software Engineering*, SE-13(1):65–76, Jan. 1987.

[29] A. P. Black. Supporting Distributed Applications: Experience with Eden. In *Proceedings of the 10th ACM Symposium on Operating Systems Principles*, pages 181–193, Washington DC, USA, Dec. 1985. ACM Press.

[30] A. P. Black, E. D. Lazowska, J. D. Noe, and J. Sanislo. The Eden Project: A Final Report. Technical Report 86-11-01, University of Washington, Department of Computer Science, Washington DC, USA, 1986.

[31] G. Blair, J. Gallagher, D. Hutchison, and D. Shepherd, editors. *Object-Oriented Languages, Systems and Applications.* Pitman Publishing, first edition, 1991.

[32] J. Blakeley, P.-A. Larson, and F. Tompa. Efficiently Updating Materialized Views. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 61–71, Washington DC, USA, May 1986.

[33] G. Booch. *Object-Oriented Design with Applications.* Benjamin/Cummings, Redwood City, CA, USA, 1991.

[34] O. P. Buneman and E. K. Clements. Efficiently Monitoring Relational Databases. *ACM Transactions on Database Systems*, 4:368–382, Sept. 1979.

[35] P. Butterworth, A. Otis, and J. Stein. The GemStone Object Database Management System. *Communications of the ACM*, 34(10):64–77, Oct. 1991.

[36] L. E. Buzato and A. Calsavara. Stabilis: A Case-study in Writing Fault-Tolerant Distributed Applications Using Persistent Objects. In A. Albano and R. Morrison, editors, *Proceedings of the Fifth International Workshop on Persistent Object Systems*, Workshops in Computing, pages 354–375, San Miniato, Italy, Sept. 1992. Springer-Verlag.

[37] R. H. Campbell, N. Islam, D. Raila, and P. Madany. Designing and Implementing Choices: An Object-Oriented System in C++. *Communications of the ACM*, 36(9):117–126, Sept. 1993.

[38] R. H. Campbell, G. M. Johnston, P. W. Madany, and V. F. Russo. Principles of Object-Oriented Operating System Design. Research Report UIUCDCS-R-89-1510, University of Illinois at Urbana-Champaign, Department of Computer Science, Apr. 1989.

[39] R. H. Campbell and P. W. Madany. Considerations of Persistence and Security in Choices, an Object-Oriented Operating System. Technical Report UIUCDCS-R-91-1670, University of Illinois at Urbana-Champaign, Department of Computer Science, Mar. 1991.

[40] M. Carey, D. DeWitt, D. Graefe, D. Haight, J. Richardson, D. Schuh, E. Shekita, and S. L. Vandenberg. The EXODUS Extensible Database Manament System Project: An Overview. In Zdonik and Maier [147], pages 474–499.

[41] M. J. Carey, D. J. Dewitt, and S. L. Vendenber. A Data Model and Query Language for EXODUS. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 413–423, Chicago, IL, USA, June 1988.

[42] K. M. Chandy and J. Misra. The Drinking Philosophers Problem. *ACM Transactions of Programming Languages and Systems (TOPLAS)*, 6(4):632–646, Oct. 1984.

[43] M. Chandy and L. Lamport. Distributed Snapshots: Determining Global States of Distributed Systems. *ACM Transactions on Computing Systems*, 3(1):63–75, Feb. 1985.

[44] E. F. Codd. A Relational Model of Data for Large Shared Data Banks. *Communications of the ACM*, 13(6):377–387, June 1970.

[45] O.-J. Dahl, B. Myhrhaug, and K. Nygaard. Simula 67 Common Base Language. Research Report S-22, Norwegian Computing Centre, Oslo, Oct. 1970.

[46] P. Dasgupta, R. Ananthanarayanan, S. Menon, A. Mohindra, and R. Chen. Distributed Programming with Objects and Threads in the Clouds System. *Computing Systems*, 4(3):243–275, Summer 1991.

[47] A. M. Davis. A Comparison of Techniques for the Specification of External System Behaviour. *Communications of the ACM*, 31(5):514–530, May 1988.

[48] U. Dayal et al. The HiPAC Project: Combining Active Databases and Timing Constraints. *ACM SIGMOD Record*, 17(1):51–70, Mar. 1988.

[49] O. Deux and et al. The O_2 System. *Communications of the ACM*, 34(10):34–48, Oct. 1991.

[50] E. W. Dijkstra. *Hierarchical Ordering of Sequential Processes*, pages 72–93. Academic Press, New York, NY, USA, 1972.

[51] U. Dittrich, K.R.; Dayal and A. Buchmann, editors. *On Object-Oriented Database Systems*. Topics in Information Systems. Springer-Verlag, Berlin, Germany, 1991.

[52] G. N. Dixon. *Object Management for Persistence and Recoverability*. Ph.D. Thesis, University of Newcastle upon Tyne, Department of Computing Science, Dec. 1988.

[53] G. N. Dixon, G. D. Parrington, S. K. Shrivastava, and S. M. Wheater. The Treatment of Persistent Objects in Arjuna. In *Proceedings of the Third European Conference on Object-Oriented Programming, ECOOP'89*, pages 169–189, Nottingham, GB, July 1989.

[54] A. Duda and J. Cayuela. System Management in the Guide Distributed System. Bull-IMAG Systèmes, Gières, France, Sept. 1992.

[55] D. Embley, B. Kurtz, and S. Woodfield. *Object-Oriented Systems Analysis: A Model-Driven Approach*. Yourdon Press, Englewood Cliffs, NJ, USA, 1992.

[56] J. L. Eppinger, L. B. Mummert, and A. Z. Spector, editors. *Camelot and Avalon: A Distributed Transaction Facility*. Data Management Systems. Morgan Kaufmann Publishers, Inc., San Mateo, CA, USA, 1991.

[57] K. Eswaran and D. Chamberlain. Functional Specifications of a Subsystem for Data Base Integrity. In *Proceedings of the First Very Large Databases Conference*, pages 48–68, Framingham, MA, USA, Sept. 1975.

[58] K. P. Eswaran, J. N. Gray, R. A. Lorie, and I. L. Traiger. The Notions of Consistency and Predicate Locks in a Database System. *Communications of the ACM*, 19(11):624–633, Nov. 1976.

[59] N. H. Gehani, R. Agrawal, and O. Shmueli. Event Specification in an Active Object-Oriented Database. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 81–90, San Diego, CA, USA, June 1992.

[60] A. Goldberg and D. Robson. *Smalltalk-80: The Language and its Implementation*. Addison-Wesley, Reading, Massachusetts, 1983.

[61] K. E. Gorlen, S. M. Orlow, and P. Plexico. *Data Abstraction and Object-Oriented Programming in C++*. John Wiley and Sons, 1989.

[62] J. N. Gray, R. A. Lorie, G. R. Putzolu, and I. L. Traiger. *Granularity of Locks and Degrees of Consistency in a Shared Data Base.* Elsevier North-Holland, Amsterdam, 1976.

[63] D. Gries, editor. *The Science of Programming.* Springer-Verlag, first edition, 1981.

[64] D. Haban and W. Wigel. Global Events and Global Breakpoints in Distributed Systems. In *Proceedings of the 21st International Conference of System Sciences*, pages 166–175, Hawaii, USA, 1988.

[65] S. Habert, L. Mosseri, and V. Abrossimov. COOL: Kernel Support for Object-Oriented Environments. In *Joint Proceedings of the Fifth Annual Conference on Object-Oriented Programming: Systems, Languages and Application, and the Fourth European Conference on Object-Oriented Programming, OOPSLA/ ECOOP'90 Conference*, pages 269–277, Ottawa, Canada, Oct. 1990.

[66] E. Hanson. A Performance Analysis of View Materialization Strategies. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 440–453, San Francisco, CA, USA, May 1987.

[67] D. Harel. Statecharts: A Visual Formalism for Complex Systems. *Science of Computer Programming*, 8(3):231–274, June 1987.

[68] D. Harel. On Visual Formalisms. *Communications of the ACM*, 31(5):514–530, May 1988.

[69] D. Harel et al. STATEMATE: A Working Environment for the Development of Complex Reactive Systems. *IEEE Transactions of Software Engineering*, 16(4):403–414, Apr. 1990.

[70] D. Harel and A. Pnueli. *On the Development of Reactive Systems*, volume 13 of *NATO ASI Series F*, pages 477–498. Springer-Verlag, Berlin, 1985.

[71] M. P. Herlihy and J. M. Wing. Avalon: Language Support for Reliable Distributed Systems. In *Proceedings of the 17th IEEE Fault-Tolerant Computing Symposium, FTCS-17*, pages 89–74. IEEE Press, Pittisburgh, Pennsylvania, USA, July 1987. IEEE.

[72] A. S. Hornby and A. P. Cowie, editors. *Oxford Advanced Learner's Dictionary.* Oxford University Press, 1989. 4th edition.

[73] D. Jason Penney and J. Stein. Class Modification in the GemStone Object-Oriented Database Management System. In *Proceedings of the Second Annual Conference on Object-Oriented Programming: Systems, Languages and Application, OOPSLA' 87*, pages 111–117, Orlando, FL, USA, Oct. 1987.

[74] E. Jul, H. Levy, N. Hutchinson, and A. P. Black. Fine-Grained Mobility in the Emerald System. *ACM Transactions on Computer Systems*, 6(1):109–133, Feb. 1988.

[75] M. F. Kaashoek, A. S. Tanenbaum, S. Flynm Hummel, and H. E. Bal. An Efficient Reliable Broadcast Protocol. *Operating Systems Review*, 23(4):5–19, Oct. 1989.

[76] W. Kim. *Introduction to Object-Oriented Databases.* Computer Systems Series. MIT Press, Cambridge, Mass., first edition, 1990.

[77] W. Kim and F. H. E. Lochovsky. *Object-Oriented Concepts, Databases, and Applications.* Frontier Series. ACM Press, 1989.

[78] S. Krakowiak, M. Meysembourg, H. Nguyen Van, M. Riveill, C. Roisin, and X. Rousset de Pina. Design and Implementation of an Object-Oriented, Strongly Typed Language for Distributed Applications. *Journal of Object Oriented Programming*, 3(3):11–22, September/October 1990.

[79] J. Kramer and J. Magee. Dynamic Configuration of Distributed Systems. *IEEE Transaction on Software Engineering*, SE-11(4):424–436, Apr. 1985.

[80] J. Kramer and J. Magee. The Evolving Philosophers Problem: Dynamic Change Management. *IEEE Transactions on Software Engineering*, 16(11):1293–1306, Nov. 1990.

[81] J. Kramer, J. Magee, and M. Sloman. Configuration Support for System Description, Construction and Evolution. In *Proceedings of the IEEE Fifth International Workshop on Software Specification and Design*, pages 28–33. Pittisburgh, PA, USA, May 1989.

[82] J. Kramer, J. Magee, M. Sloman, and N. Dulay. Configuring Object-Based Distributed Programs in REX. *IEE Software Engineering Journal*, 7(2):139–149, Mar. 1992.

[83] C. Lamb, G. Landis, J. Orenstein, and D. Weinreb. The ObjectStore Database System. *Communications of the ACM*, 34(10):50–63, Oct. 1991.

[84] L. Lamport. *LaTeX: User's Guide and Reference Manual.* Reading, Massachusetts, 1986. Appendix B: The Bibliography Database, pages 140-147.

[85] E. Lazowska, H. Levy, G. Almes, M. Fischer, R. Fowler, and S. Vestal. The Architecture of the Eden System. In *Proceedings of the Eigth Symposium on Operating Systems Principles*, pages 148–159, Pacific Grove, CA, USA, Dec. 1981.

[86] R. Lea, C. Jacquemont, and E. Pillevesse. COOL: System Support for Distributed Programming. *Communications of the ACM*, 36(9):37–46, Sept. 1993.

[87] R. LeBlanc and C. T. Wilkes. Systems Programming with Objects and Actions. In *Proceedings of the Fifth International Conference on Distributed Computing Systems*, pages 132–139. Computer Society Press, Silver Spring, MD, USA, May 1985.

[88] C. Lécluse, P. Richard, and F. Velez. O_2, an Object-Oriented Data Model. In *Proceedings ACM SIGMOD International Conference on Management of Data*, pages 424–433, Chicago, IL, USA, June 1988.

[89] B. Lindsay, L. Haas, and C. Mohan. A Snapshot Differential Refresh Algorithm. In *Proceedings of ACM SIGMOD International Conference on Management of Data*, pages 53–60, Washington DC, USA, May 1986.

[90] B. Liskov. Distributed Programming in Argus. *Communications of the ACM*, 31(3):300–312, Mar. 1988.

[91] B. Liskov, D. Curtis, P. Johnson, and R. Scheifler. Implementation of Argus. In *Proceedings of the 11th ACM Symposium on Operating Systems Principles*, pages 111–122, Austin, TX, USA, Nov. 1987.

[92] B. Liskov and R. Scheifler. Guardians and Actions: Linguistic Support for Robust Distributed Programs. *ACM Transactions on Programming Languages and Systems*, 5(3):381–404, July 1983.

[93] G. M. Lohman, B. Lindsay, H. Pirahesh, and K. Bernhard Schiefer. Extensions to Starburst: Objects, Types, Functions, and Rules. *Communications of the ACM*, 34(10):94–109, Oct. 1991.

[94] D. B. Lomet. Process Structure, Synchronisation and Recovery using Atomic Actions. *ACM SIGPLAN Notices*, 12(3):128–137, Mar. 1977.

[95] N. Lynch, M. Merrit, W. Weihl, and A. Fekete. *Atomic Transactions*. Morgan Kaufman Publishers, San Mateo, CA, USA, first edition, 1994.

[96] P. W. Madany, N. Islam, P. Kougiouris, and R. H. Campbell. Reification and Reflection in C++: An Operating Systems Perspective. Technical Report TTR92-44, University of Illinois, Department of Computer Science, Urbana, IL, USA, Mar. 1992.

[97] P. Maes. Concepts and Experiments in Computational Reflection. *ACM SIGPLAN Notices*, 22(12):147–155, Dec. 1987.

[98] J. Magee, N. Dulay, and J. Kramer. Structuring Parallel and Distributed Programs. In *Proceedings of the First International Workshop on Configurable Distributed Systems*, pages 102–117, Imperial College, London, GB, Mar. 1992.

[99] J. Magee, N. Dulay, and J. Kramer. A Constructive Development Environment for Parallel and Distributed Programs. In *Proceedings of the Second International Workshop on Configurable Distributed Systems*, pages 4–14, Pittsburgh, PY, USA, Mar. 1994.

[100] J. Magee, J. Kramer, and M. Sloman. Constructing Distributed Systems in Conic. *IEEE Transactions of Software Engineering*, 15(6):663–675, June 1989.

[101] J. Magee, J. Kramer, M. Sloman, and N. Dulay. An Overview of the REX Software Architecture. In *Proceedings of the Second IEEE Computer Society Workshop on Future Trends of Distributed Computing Systems*. Cairo, Egypt, Oct. 1990.

[102] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems: Specification*, volume I. Springer-Verlag, New York, first edition, 1992.

[103] C. E. McDoWell and D. P. HelmBold. Debugging Concurrent Programs. *ACM Computing Surveys*, 21(4):593–622, Dec. 1989.

[104] J. D. McGregor and D. M. Dyer. A Note on Inheritance and State Machines. *Software Engineering Notes*, 18(4):61–69, Oct. 1993.

[105] P. M. Melliar-Smith and B. Randell. Software Reliability: The Role of Programmed Exception Handling. In Shrivastava [125], pages 143–153.

[106] B. Meyer. *Object-Oriented Software Construction*. Prentice-Hall, Hertfordshire, GB, first edition, 1988.

[107] Meyer, B. *Eiffel: The Language*. Prentice Hall, first edition, 1991.

[108] B. Miller and J. Choi. Breakpoints and Halting in Distributed Programs. In *Proceedings of the Eighth International Conference on Distributed Computing Systems*, pages 316–323, San Jose, CA, USA, June 1988.

[109] M. Morgenstern. Active Databases as a Paradigm for Enhanced Computing Environments. In *Proceedings of the Ninth Very Large Databases Conference*, Florence, Italy, 1983.

[110] R. Morrison, A. L. Brown, R. C. H. Connor, and A. Dearle. Napier88 Reference Manual. Technical Report PPRR-77-89, University of St. Andrews, Department of Computer Science, 1989.

[111] J. E. B. Moss. Nested Transactions: An Approach to Reliable Computing. Ph.D. Thesis LCS-TR-260, MIT, Department of Computer Science, 1981.

[112] S. J. Mullender, G. van Rossum, A. S. Tanenbaum, R. van Renesse, and H. van Staveren. Amoeba: A Distributed Operating System for the 1990s. *IEEE Computer*, 23(5):44–53, May 1990.

[113] P. O'Brien, D. Halbert, and M. Kilian. The Trellis Programming Environment. In *Proceedings of the Second Annual Conference on Object-Oriented Programming: Systems, Languages and Applications, OOPSLA'87*, Orlando, FL, USA, Oct. 1987.

[114] G. Parrington, S. Shrivastava, S. Wheater, and M. Little. The Design and Implementation of Arjuna. Broadcast Project Deliverable Report, University of Newcastle upon Tyne, Department of Computing Science, GB, Oct. 1994.

[115] G. D. Parrington. Reliable Distributed Programming in C++: the Arjuna Approach. In *Second Usenix C++ Conference*, pages 37–50, San Francisco, USA, Apr. 1990.

[116] R. K. Rajendra, H. M. Tempero, Ewan andLevy, A. P. Black, N. C. Hutchinson, and E. Jul. Emerald: A General-Purpose Programming Language. *Software-Practice and Experience*, 21(1):91–118, Jan. 1991.

[117] J. E. Richardson and M. J. Carey. Persistence in the E Language. *Software Practice and Experience*, 19(2):1115–1150, Dec. 1988.

[118] J. E. Richardson and M. J. Carey. Implementing Persistence in E. In *Proceedings of the Workshop on Persistent Object Systems*, pages 302–319, Newcastle, Australia, Jan. 1989.

[119] M. Rozier et al. Chorus Distributed Operating Systems. *Computing Systems*, 1(4):305–370, Fall 1988.

[120] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object-Oriented Modeling and Design.* Prentice-Hall, Englewood Cliffs, NJ, USA, 1991.

[121] F. Schneider. Implementing Fault-Tolerant Services Using the State Machine Approach : A Tutorial. *ACM Computing Surveys*, 22(4):299–319, Dec. 1990.

[122] M. Shapiro, Y. Gourhant, S. Habert, L. Mosseri, M. Ruffin, and C. Valot. SOS: An Object-Oriented Operating System—Assessment and Perspectives. *Computing Systems*, 2(4):287–338, Dec. 1989.

[123] S. Shlaer and S. Mellor. *Object Lifecycles: Modeling the World in States.* Yourdon Press/Prentice Hall, Englewood Cliffs, NJ, USA, 1992.

[124] S. Shrivastava and S. Wheater. Implementing Fault-Tolerant Distributed Applications Using Objects and Multi-Coloured Actions. In *Proceedings of the 10th International Conference on Distributed Computing Systems*, pages 203–210, Paris, France, May 1990.

[125] S. K. Shrivastava, editor. *Reliable Computer Systems.* Springer-Verlag, Berlin, Germany, first edition, 1985.

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

[127] R. Snodgrass. *Monitoring Distributed Systems: A Relational Approach.* Ph.D. Thesis, Carnegie-Mellon University, Department of Computer Science, Dec. 1982.

[128] R. Snodgrass. A Relational Approach to Monitoring Complex Systems. *ACM Transactions on Computer Systems*, 6(2):157–196, May 1988.

[129] A. Z. Spector et al. Support for Distributed Transactions in the TABS Prototype. *IEEE Transactions on Software Engineering*, SE-11(6):520–530, June 1985.

[130] A. Z. Spector, R. F. Pausch, and G. Bruell. Camelot: A Flexible Distributed Transaction Processing System. In *Proceedings of the IEEE Compcon*, San Francisco, CA, USA, Mar. 1988.

[131] M. Spezialetti. *A Generalized Approach to Monitoring Distributed Computations for Event Ocurrences.* Ph.D. Thesis, University of Pittsburgh, Department of Computer Science, 1989.

[132] M. Spezialetti and J. Kearns. A General Approach to Reconizing Event Occurrences in Distributed Computations. In *Proceedings of the Eighth International Conference on Distributed Computing Systems*, pages 300–307, San Jose, CA, USA, June 1988.

[133] G. L. Steele Jr., S. E. Fahlman, R. P. Gabriel, D. A. Moon, and D. L. Weinreb, editors. *Common Lisp.* Digital Press, first edition, 1984.

[134] M. Stonebraker, E. Hanson, and C. Hong. The Design of the POSTGRES Rules System. In M. Stonebraker, editor, *Readings in Database Systems*, pages 556–565. Morgan Kaufmann Publishers, Inc., 1988.

[135] M. Stonebraker and G. Kemnitz. The POSTGRES Next-Generation Database Management System. *Communications of the ACM*, 34(10):78–92, Oct. 1991.

[136] M. Stonebraker, L. A. Rowe, and M. Hirohama. The Implementation of POSTGRES. *IEEE Transactions on Knowledge and Data Engineering*, 2(1):43–59, Mar. 1990.

[137] B. Stroustrup. *The C++ Programming Language.* Addison-Wesley, Reading, Massachusetts, 1986.

[138] A. S. Tanenbaum. *Modern Operating Systems.* Prentice-Hall International, Inc., Englewood Cliffs, NJ, 1992.

[139] Transarc Corporation, Pittsburgh, PA, USA. *Encina from Transarc: Product Overview*, 1991. TP-00-M235.

[140] Transarc Corporation, Pittsburgh, PA, USA. *A Competitive Analysis of Transarc's Encina and UNIX System Laboratories' Tuxedo*, 1993. White paper.

[141] Transarc Corporation, Pittsburgh, PA, USA. *An Introduction to Programming the Encina Monitor*, 1993. White paper.

[142] W. Weihl. Implementation of Resilient Atomic Data Types. *ACM Transactions on Programming Languages and Systems*, 7(2):244–269, 1985.

[143] M. Wood. *Fault-Tolerant Management of Distributed Applications Using the Reactive System Architecture.* Ph.D. Thesis, Cornell University, Department of Computer Science, Dec. 1991.

[144] Y. Yokote. The Apertos Reflective Operating System: the Concept and its Implementation. In *Proceedings of the Seventh Annual Conference on Object-Oriented Programming: Systems, Languages and Applications, OOPSLA'92*, pages 414–434, Vancouver, British Columbia, Canada, Oct. 1992.

[145] Y. Yokote, F. Teraoka, and A. Mitsuzawa. The Muse Object Architecture: A New Operating System Structuring Concept. *Operating Systems Review*, 25(2):22–46, Apr. 1991.

[146] Y. Yokote, F. Teraoka, and M. Tokoro. A Reflective Architecture for an Object-Oriented Distributed Operating System. In *Proceedings of the Third European Conference on Object-Oriented Programming, ECOOP'89*, pages 89–106, Nottingham, GB, July 1989.

[147] S. Zdonik and D. Maier, editors. *Readings in Object-Oriented Database Systems.* Morgan-Kaufmann Publishers, Inc., Palo Alto, CA, USA, 1990.