An Overview of the Tigger Object-Support Operating System Framework

Vinny Cahill[*]

Distributed Systems Group
Department of Computer Science
Trinity College Dublin
Ireland
http://www.dsg.cs.tcd.ie/

Abstract. This paper describes the motivations for and main features of Tigger – a framework for the construction of a family of object-support operating systems that can be tailored for use in a variety of different application domains. An important goal of the design of Tigger is that instantiations of the framework should be able to support (a number of) different object models in order to allow a range of object-oriented languages for distributed or persistent programming to be supported without unnecessary duplication of effort. A further goal of the design is that instantiations of the framework should be able to support the same object model in different ways depending on the requirements of the applications to be supported by those instantiations. This paper describes the main features of the Tigger framework that allow these goals to be realised.

1 Introduction

While the use of object-support operating systems – supporting distributed or persistent objects – has been advocated for many application domains [5], research within the Distributed Systems Group (DSG) at Trinity College has recently been considering their deployment in two specific areas: support for Concurrent Engineering (CE) environments[1]; and support for the development and execution environments of next-generation, multi-user/distributed, arcade and personal-computer (PC) video games [15].

These application domains are similar at one level in that they are fundamentally concerned with multiple distributed users interacting via shared distributed and persistent objects. Moreover, these two application domains are not necessarily distinct; game development is inherently a CE activity involving game designers, artists, musicians, and software developers. Furthermore, the

[*] Email: vinny.cahill@dsg.cs.tcd.ie

[1] By "environment" is really meant what is known in the CE community as a "framework", i.e., a system encapsulating a set of tools, together with the data used by those tools, under a common design management protocol.

technologies being deployed in both domains, for example, real-time video and audio or three-dimensional graphics, overlap.

Despite these similarities, there are many differences that must be accommodated. At the highest level, the object models appropriate to each application domain differ. CE environments often employ what may be described as a *shared data object model* in which objects are passive and accessed by active threads of control. In contrast, the game execution environment described in [15] employs a *reactive object model* with event-based communication, i.e., objects representing game entities are autonomous but react to events raised by other objects. Objects receive notifications of events of interest determined by reference to parameters of those events. While these particular models represent different positions along a continuum of possible object models, the key observation is that an important feature of any flexible object-support operating system should be the ability to support a number of different object models without duplication of effort, if not necessarily simultaneously.

While these application domains exhibit similar functional requirements, their non-functional requirements vary considerably from application to application and installation to installation. Requirements in areas such as support for security, heterogeneity, reliability and fault tolerance, allowable memory usage, and real-time behaviour vary considerably. For example, the requirements for supporting the *same* video game on a stand-alone arcade machine or PC, in a private network of arcade machines, or across the public telephone network vary considerable but must be supported by a common system interface. Likewise the requirements imposed by supporting a concurrent software engineering environment in a traditional workstation/server environment are different from those of any of the above scenarios. Nevertheless, the game developer still needs easy access to the game execution system during testing and debugging and, more importantly, the game designer needs immediate access to the execution system during the game tuning phase when the "playability" of the game is being improved. The key observation here is that any flexible object-support operating system should be capable of supporting the same object model in different ways depending on the way in which the applications supported by the system are to be deployed.

In keeping with these observations, the Tigger project[2] undertook the design, not of a single object-support operating system, but of a family of object-support operating systems whose members can be customised for use in a variety of different application domains. The two primary goals of this design were:

1. to allow members of the family to support (a number of) different object models in order to allow a range of different object-oriented programming languages for distributed and persistent programming to be supported without unnecessary duplication of effort; and
2. to allow the same object model to be supported in different ways subject to differing non-functional requirements.

[2] which was named after A.A. Milne's famously bouncy character!

To support customisability, the design is captured as a framework that can be instantiated to implement the individual members of the family. The Tigger framework can be instantiated to implement particular object-support operating systems meeting particular functional and non-functional requirements. Instantiations of the Tigger framework can be layered above bare hardware, (real-time) microkernels, or conventional operating systems. The Tigger framework is sufficiently general so as to allow a set of possible instantiations that is capable of supporting a wide range of object-oriented programming languages for distributed or persistent programming and that is suitable for use in a wide range of application areas exhibiting different non-functional requirements. In addition, the Tigger framework has been designed to be extensible so that new functionality can be supported when required.

The major abstractions supported by the Tigger framework are distributed or persistent objects, threads, and extents (i.e., protected collections of objects). A given Tigger instantiation may support only distributed objects, only persistent objects, or both. Of course, different instantiations will support these abstractions in different ways (for example, in order to accommodate different object models) by employing different mechanisms and policies.

Individual Tigger instantiations may support additional abstractions, such as activities (i.e., distributed threads) and object clusters as required. Moreover, all of these abstractions are based on lower-level abstractions such as contexts (i.e., address spaces) and endpoints (i.e., communication channels) that are not normally expected to be used directly by supported languages or individual applications.

The simplest object-support operating system that can be instantiated from the framework is one supporting a single user and a single object model providing either distributed or persistent objects. Other instantiations of the framework may support additional abstractions, multiple users, or multiple object models.

The remaider of this paper gives an overview of the Tigger framework, concentrating on the way in which it provides support for distributed and persistent programming languages. For a complete description of the framework see [3].

2 Related Work

This section introduces a number of previous systems that have particularly influenced Tigger: the Amadeus object-support operating system, which was developed by DSG, and the Choices and PEACE object-oriented operating systems.

Amadeus [9] was a general-purpose object-support operating system that supported distributed and persistent programming in multi-user distributed systems. Amadeus was targeted for use in what may broadly be described as co-operative applications concerned with access to shared data in domains such as computer-aided design (CAD), office automation, and software engineering.

A major feature of Amadeus was that it was designed to support the use of a range of existing object-oriented programming languages. A language could be extended to support a set of (inter-related) properties including distribution, persistence, and atomicity for its objects by using the services of the Amadeus Generic Runtime Library (GRT), while maintaining its own native object reference format and invocation mechanism [4]. The Amadeus GRT provided a range of mechanisms from which the language designer could choose those appropriate for the intended use of the extended language. Extended versions of C++ and Eiffel, which were known as C** [7] and Eiffel** [13] respectively, and an implementation of the E persistent programming language [12] were supported by Amadeus.

Other major features of Amadeus included language-independent support for atomic objects and transactions [14, 24] based on the use of the RelaX transaction manager and libraries [11], and a novel security model supporting access control for objects at the level of individual operations as well as isolation of untrustworthy code [17].

Experience with the design and implementation of Amadeus has obviously had a major influence on Tigger. Tigger shares the goal of language independence and has adopted several of the key features of Amadeus including the idea of a GRT and the basic security model. However, the goal of Tigger is to allow the implementation of a variety of object-support operating systems providing more or less functionality as required, rather than a single general-purpose system as was the goal of Amadeus.

Apart from Amadeus, Tigger has been most influenced by Choices [6], which developed a C++ framework for the construction of operating systems for distributed and shared memory multiprocessors, and Peace [19], which addressed the use of object-oriented techniques in the construction of a family of operating systems for massively parallel computers. The Peace family encompasses a number of different members ranging from one supporting a single thread of control per node to one supporting multiple processes per node.

In some sense, Tigger may be seen as combining these two research areas to develop a family of *object-oriented object-support operating systems*. The development of an object-oriented object-support operating system, to be known as Soul, was proposed previously by Shapiro in [20]. Shapiro envisaged developing a "hierarchy of object-support object types and classes" that could "be re-used, parameterized, and combined together, in order to build specific object-support functions". This is indeed a reasonable description of the Tigger framework! A later paper on Soul, [21], elaborated on the original proposal and described a "preliminary design" for the interface that should be provided by a microkernel suitable for hosting the Soul class hierarchy. However, no other description of the Soul object-oriented object-support operating system appears to be available in the literature. The Soul project has apparently instead concentrated on the development of specific mechanisms for object reference management and garbage collection in distributed systems.

3 Overview

Tigger is a framework for the construction of a family of object-support operating systems. Every instantiation of the framework is an object-support operating system to which one or more object-oriented programming languages are bound in order to provide an application programming interface. Like other object-support operating systems, Tigger instantiations will typically provide support for features such as creation of distributed or persistent objects; access to remote objects; object migration; access to stored persistent objects; dynamic loading of objects on demand; dynamic loading and linking of class code; and protection of objects.

In fact, the heart of any Tigger instantiation is a generalised object-access mechanism that allows local, remote, stored, protected, or unprotected objects to be accessed in a uniform manner. This mechanism provides support for all aspects of locating the target object, mapping the object and its class into memory, and forwarding the access request to the object as required. In fact, this basic mechanism subsumes much of the functionality provided by the Tigger framework and provides the basis for supporting a high degree of network transparency for object access. Of course, the details of what this mechanism does, and how it does it, are subject to customisation and will differ from one Tigger instantiation to another.

It is important to understand two points about the nature of the functionality provided by a Tigger instantiation. First, a Tigger instantiation, in cooperation with the runtime libraries of supported languages, *only* provides the necessary support for the use of objects by applications, i.e., a Tigger instantiation is *an object-support system*. The semantics and function of the objects that they support are opaque to Tigger instantiations. A particular object might implement a spreadsheet, one cell in a spreadsheet, a file, or a file server. The distinction is not visible to Tigger instantiations. While some objects will implement (parts of) particular applications (such as the "spreadsheet cell" object above), other objects may provide common services including those that are usually thought of as being part of an operating system (such as the "file server" object above).

The second major point to be understood is that a Tigger instantiation is *a language-support system* – the functionality provided by a Tigger instantiation is intended to be used by object-oriented programming languages to provide programming models based on distributed or persistent objects to their application programmers. Thus, the main interface provided by a Tigger instantiation is that provided for the language implementer. The interface used by an application developer is that provided by a supported language. Moreover, a Tigger instantiation provides only basic support for distribution or persistence that is intended to be supplemented by each language's runtime library in order to implement the programming model of the language. How support for distribution or persistence is made available in any language – whether transparently to application programmers, via a class library, or even via the use of particular language constructs – is not mandated by the Tigger framework. Likewise, the degree of network transparency provided by the language is a function of the programming model supported by the language. Of course, the Tigger framework has been designed to support languages that provide a high degree of network transparency.

4 Software Architecture

Tigger instantiations are intended to support both conventional object-oriented programming languages that have been extended to support distributed or persistent programming as well as object-oriented languages originally designed for that purpose. Moreover, this is intended to be done in a way that does not impose particular constructs and models on the language and, where an existing language is being extended, that does not necessarily require changes to its compiler nor to its native object reference format or local invocation mechanism. In this way, the language designer is free to choose the object model to be provided to application programmers independently. Supporting existing (local) object reference formats and invocation mechanisms allows the common case of local object invocation to be optimised. Finally, where an existing language is to be supported, this approach facilitates the reuse and porting of its existing compiler and runtime libraries.

In order to achieve these goals, every Tigger instantiation provides one or more GRTs providing common runtime support for one or more languages supporting distributed or persistent programming that have similar requirements on their runtime support. A more precise characterisation of a GRT is given in [3]. Suffice it to say here that a GRT is *generic* in the sense that it provides only that part of the support for distribution or persistence that is independent of any language. Every GRT is bound to a *Language Specific Runtime Library* (LSRT) for each language to be supported. The LSRT provides language-dependent runtime support. Each GRT provides an interface to the language implementer that has been designed to interface directly and easily to an LSRT. Thus, the interface to a Tigger instantiation seen by a language implementer is that of one of the GRTs that it provides.

This basic approach to language support is derived from the Amadeus project. Unlike Amadeus, which provided a single GRT supporting a (fairly limited) range of mechanisms that could be used by supported languages, the Tigger framework allows GRTs to be customised depending on the object model and intended use of the language(s) to be supported. For example, GRTs supporting remote object invocation (ROI) and/or distributed shared memory (DSM) style access to distributed objects, GRTs supporting the use of different object fault detection or avoidance schemes, and GRTs supporting the use of eager, lazy, or no swizzling can all be instantiated from the framework. A given Tigger instantiation can support one language or several similar languages with one GRT, or a number of different languages with several GRTs. For example, figure 1 shows one possible scenario in which one Tigger instantiation provides two different GRTs: one GRT is being used to support the C** and Eiffel** programming languages while the other is being used to support the E programming language. Both C** and

Eiffel** support distributed and persistent objects using ROI and eager swizzling respectively, while E is a non-swizzling persistent programming language. The figure depicts a scenario in which one application is written using some combination of two supported languages. While such interworking between languages may be facilitated when the languages involved have some of their runtime support in common, it should be noted that it cannot be implemented completely at this level – additional mechanisms are still required at higher levels to, for example, support inter-language type checking.

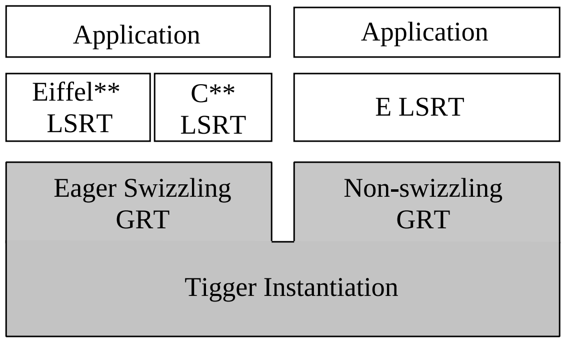

Fig. 1. A Tigger instantiation.

Amadeus supported exactly the set of languages depicted in figure 1 but with a single GRT. However, the Amadeus GRT was both complex and large, and hence penalised languages and applications that typically only required a subset of the features that it provided. The Tigger approach allows the GRT to be customised according to the specific requirements of the language implementer.

4.1 Logical Model

The classes making up the Tigger framework are divided into five main class categories [2]. Essentially, each of these class categories is responsible for supporting some subset of the fundamental abstractions provided by the Tigger framework as follows:

- the GRT class category – known as Owl[3] – supports distributed and persistent objects and optionally clusters, and provides the main interface to supported languages. An instantiation of Owl corresponds to a GRT as described above and, a single Tigger instantiation may include multiple Owl instantiations.

[3] Yes, you've guessed it! All the class categories are called after characters from A.A. Milne's books.

- the threads class category – known as Roo – supports threads and related synchronisation mechanisms, and may support activities and jobs. Supported languages (i.e., their LSRTs) and applications may use Roo directly.
- the communications class category – known as Kanga – supports endpoints. Again, supported languages and applications may use Kanga directly.
- the storage class category – known as Eeyore – supports containers and storage objects. Supported languages are not expected to use Eeyore directly and hence its main client is Owl.
- the protection class category – known as Robin – supports extents and related abstractions.

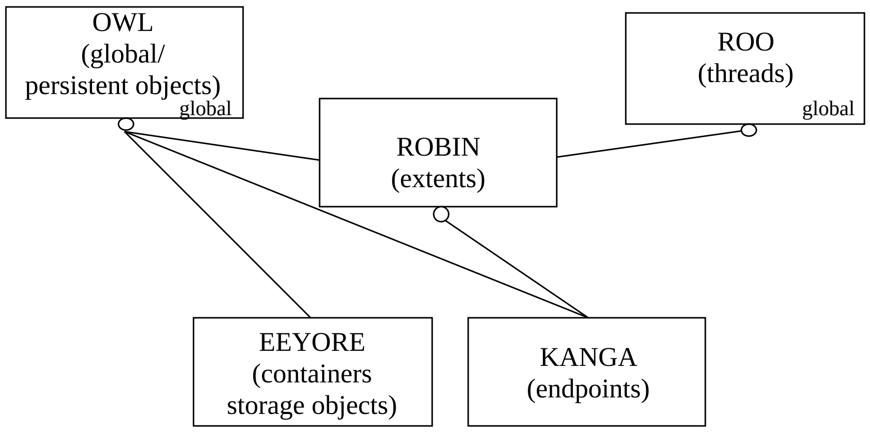

Fig. 2. Tigger class categories.

Figure 2 is a top-level class diagram for the Tigger framework showing the class categories introduced above and their using relationships. Note that both Owl and Roo are labelled as `global` meaning that they may be used by all the other class categories. In the case of Roo, this reflects the fact that all the components of a Tigger instantiation are expected to be thread-aware. In the case of Owl, this reflects the fact that components of a Tigger instantiation may use distributed or persistent objects.

While Owl is specialised depending primarily on the needs of the language(s) to be supported, the other class categories can also be specialised to support different mechanisms and policies. In particular, different instantiations of Robin determine whether the Tigger instantiation of which it is a part supports only a single extent or multiple extents, as well as whether it supports one or more contexts. Other responsibilities of Robin include supporting cross-extent object invocation and unique object identification. Decisions made about the implementation of Robin are therefore of fundamental importance for the overall structure of a Tigger instantiation.

The Robin and Owl class categories are described in detail in [3]. For descriptions of early versions of Roo and Kanga see [8]. Eeyore is based on the Amadeus persistent object store (POS) framework described in [16]. The remainder of this paper gives an overview of the Owl class category.

5 Object-Support Functionality

This section considers how the functionality required to support distributed and persistent objects can be divided between an LSRT and a GRT. Given this separation of responsibilities, the major services supported by Owl, as well as the main options for implementing those services, are then identified. An important goal of Tigger was that Owl should provide as much of the required functionality as possible. Only where a service is clearly language-specific or is intimately connected with the code that is generated by the language's compilation system is that service assigned to the LSRT.

In reading the following sections, it should be borne in mind that a particular Owl instantiation (i.e., GRT) might support only persistent objects, only distributed objects, or both. Hence, not all of these services will need to be supported by all Owl instantiations. Furthermore, the list presented here is not exhaustive; Owl can be extended to support other services.

Object layout and naming: Unless the compilation system is to be seriously constrained by the use of a GRT, the LSRT should be able to dictate the layout of objects in memory, the format of internal/local references used by such objects, and the mapping from such an object reference to the address of a collocated object. To reflect this fact, internal/local references are referred to as *language references* or lREFs from here on.

A GRT, on the other hand, should be responsible for the provision of globally unique object identifiers (OIDs) suitable for identifying every object in the system[4] A GRT should also be responsible for the provision of external/global references (referred to simply as global references or gREFs from here on). A GRT should also implement the mapping from a gREF for an object to the location of that object in the system.

Where the language to be supported already supports distribution or persistence, its LSRT will already support its own form of OID and gREF. Moreover, its lREFs and gREFs may be the same. Owl does not support all possible existing gREF formats but only those that support an Owl-defined protocol. Owl instantiations may however use (virtual) addresses as gREFs. Owl also supports languages in which gREFs and lREFs are the same as long as the gREFs support the appropriate protocol.

Object access, binding, and dispatching: Each language is free to determine how objects may be accessed by their clients. However, it is important

[4] In the Tigger framework, responsibility for the format and allocation of OIDs actually rests with Robin rather than Owl.

to realise that this decision has important repercussions for the choice of object fault avoidance or detection mechanisms that are available and for the ways in which object faults can be resolved. Typically, the choice of possible object fault avoidance, detection, or resolution mechanisms is constrained by the form of access to objects allowed. For example, the use of proxies to represent absent objects is not appropriate where direct access to the instance data of an object by its clients is allowed.

Since the means of binding code to objects and of dispatching invocations (including the layout of parameter frames) is usually intrinsic to the compilation system, these must continue to be implemented in the LSRT. Thus, a GRT need not be involved in local object invocation. However, this also means that when ROI is used to access remote distributed objects, the marshalling and unmarshalling of ROI requests, as well as the dispatching of incoming requests to their target objects, must be done by or in cooperation with the LSRT.

Object allocation and garbage collection: Owl only supports allocation of objects on the heap or embedded in other (heap-allocated) objects. Moreover, Owl is responsible for management of the heap and hence provides the routines to allocate (and where supported, deallocate) objects.

When necessary, Owl supports garbage collection of distributed or persistent objects both within memory and within the POS as required.

Object fault detection and avoidance: Detection or avoidance of object faults is the responsibility of the LSRT since it depends on the type of access to objects supported and the mechanisms used may need to be tightly integrated with the compilation system. For example, if presence tests are used to detect absent objects, the language compiler or preprocessor will usually be required to generate the code necessary to perform these tests before any access to an object proceeds.

Owl does however provide underlying support for a number of common object fault detection mechanisms (for example, presence tests and proxies) as well as support for object fault avoidance. Other object fault detection mechanisms may be implemented entirely at the language level.

Object fault resolution: Where object fault detection is used, Owl provides the underlying means of resolving object faults including locating the target objects, mapping objects, transferring ROI requests to objects, and/or migrating threads as appropriate. The choice of object fault resolution policy is however constrained by the LSRT.

In the case of ROI requests, the formatting of the request must however be carried out by the LSRT since only it understands the format of parameter frames. Owl does however support the marshalling and unmarshalling of lREFs and values of basic types. The translation of lREFs to the corresponding gREFs and vice versa must be carried out in cooperation with the LSRT. Similar comments apply to migration of objects. On the remote side, the LSRT must be prepared to accept incoming ROI requests from the GRT, unmarshal the parameters, dispatch the request in the language-specific manner, and, once the request has been completed, to marshal the reply. Note that the dispatching of the request must be carried out by the LSRT since only it understands the dispatching mechanism to be used.

Mapping, unmapping, and migration: Owl provides the basic support for the mapping and unmapping of persistent objects as well as the migration of distributed objects.

During mapping or migration, Owl supports the conversion of objects to local format where heterogeneity is supported; no, lazy, and eager swizzling of references as required; and binding of code to mapped objects. In each case, these actions require language- (and indeed type-) specific information. Hence, while Owl supports each of these, it does so in cooperation with the LSRTs of supported languages.

Where swizzling is used, the GRT must be able to translate a gREF to the appropriate lREF (whether or not the target object is mapped into the current address space). This again requires cooperation with the LSRT depending on the object faulting strategy in use.

Likewise, binding of code to a recently-mapped object must be done in a language-specific way. However, Owl provides the underlying support for dynamic linking where this is required including supporting the storage and retrieval of class code.

Determining which objects can be unmapped or migrated also depends on the object faulting strategy in use. Nevertheless, Owl supports both anchored and non-anchored code.

Clustering: Owl supports the use of both application-directed and transparent clustering as required.

Directory Services: Finally, Owl provides a (persistent) name service (NS) that can be used to attach symbolic names to object references.

6 An Overview of the Owl Class Category

Just as the overall Tigger framework describes the architecture of a family of object-support operating systems, Owl may be said to describe the architecture of a family of GRTs. A GRT supporting one or more specific languages is instantiated by providing appropriate implementations of (a subset of) the classes that constitute the Owl class category. The process of instantiating a GRT from Owl is obviously driven by the requirements of the language(s) to be supported but is also constrained by the model of a GRT and of GRT–LSRT interaction embodied in the design of Owl. This section describes the abstract model of a GRT, and of its interaction with an LSRT, that underpins the design of Owl. The next section describes the organisation of the Owl class category in more detail.

6.1 GRT Model

A GRT provides runtime support for distribution or persistence in cooperation with the LSRTs of the languages that it supports and the other components of the Tigger instantiation of which it is a part. Some GRTs support only distributed objects, others only persistent objects, while some support both. Whether a GRT supports distributed or persistent objects is determined by the way in which it is instantiated. Thus, distributed or persistent objects can be seen as specialisations of abstract *GRT objects* supported by Owl. Every GRT supports at least the following services for GRT objects[5]:

- object creation;
- location-independent object naming;
- object faulting;
- object mapping and unmapping;
- directory services.

Together these services constitute the basic runtime support that must be provided for any distributed or persistent programming language. Depending on how each is implemented, the resulting GRT can support distributed or persistent objects using various policies and mechanisms. A given GRT can also provide additional services such as object deletion or garbage collection, object clustering, or marshalling and unmarshalling of ROI requests. The Owl class category described in the remainder of this paper includes classes providing a number of these additional services. Moreover, Owl has been designed to be extensible so that support for further services, for example, transaction management, can be provided in the future.

Typically, each of these services is invoked by a *downcall* from the LSRT to the GRT and makes use of *upcalls* from the GRT to the LSRT when a language-specific action has to be performed or language-specific information obtained.

Every GRT provides exactly one form of gREF and one swizzling policy as dictated by the language(s) to be supported. A GRT may support either object fault avoidance or object fault detection. In the case of object fault detection, the actual detection of object faults is the responsibility of the LSRT. A given GRT may support the LSRT in using a number of different techniques for object fault detection or the object fault detection technique used may be completely transparent to the GRT. A GRT supporting object fault detection may provide a number of different interfaces for object fault reporting. Each object fault reporting interface implies a set of allowable object fault resolution techniques that the GRT can apply. In addition, a GRT for use in a multi-extent Tigger instantiation always provides interfaces supporting cross-extent object invocation and object migration between extents.

6.2 Object Model

Abstractly, at the language level, an *object* is an entity with identity, state, and behaviour [2]. Every language object is assumed to have an associated *type* that specifies the interface to the object available to its clients.

[5] In the following, the term *"object"* is used as a synonym for *"GRT object"* unless otherwise noted.

On the other hand, a GRT object can be viewed as being essentially a container for one or more language objects that can be uniquely identified and to which code implementing the interface to the contained object(s) can be bound dynamically by the appropriate LSRT. Distributed or persistent language objects must be mapped, in a way specific to their language, onto appropriate GRT objects. The most obvious mapping is to use a single GRT object for each dynamically allocated language object. Other mappings are also possible. For example, an array of language objects could be contained within a single GRT object or a language object might be embedded within another language object that is contained within a GRT object. The main consequence of supporting arrays of language objects or embedded language objects is that lREFs may map to arbitrary addresses within a GRT object rather than just the start address of the object.

In any case, both the internal structure of a particular GRT object and the semantics implemented by the contained language objects are dictated by the language level. Such information can be acquired by the GRT if necessary only by making upcalls to the LSRT. In particular, a set of upcall methods, which are implemented by the appropriate LSRT and which the GRT can call when required, must be bound to every GRT object in a way defined by Owl.

Object Allocation and Layout New GRT objects are created dynamically in the GRT's heap by explicitly calling the GRT. Neither static allocation of GRT objects in some per-context data segment nor stack allocation of GRT objects is supported.

Every GRT object has a header that is used to store information required by the GRT to manage the object. Depending on the GRT instantiation, this header may be allocated contiguously with the GRT object in memory or separately (perhaps to allow GRT objects to be moved within memory while mapped). In normal operation, an object's GRT header is transparent to the language level although it may be accessed by upcall code provided by the LSRT.

Language objects are expected to be contiguous in memory but may have contiguous or non-contiguous headers containing information required by their LSRTs. In order to support LSRTs that use non-contiguous object headers, a GRT may be specialised to allow GRT objects to be split into (at most) two memory regions resulting in the four possible GRT object layouts being supported

Object Naming GRT objects are uniquely identified by Robin OIDs. GRTs may assign OIDs to objects either eagerly, i.e., when they are created, or lazily, i.e., at least some time before they become visible outside of their cradle extent, i.e., the extent in which they were created. A GRT that supports lazy OID allocation may for example allocate OIDs to objects only when they become known outside of their cradle extent, when they become known outside of the context in which they were created, or, if clustering is supported, when they become known outside of their initial cluster.

Supporting lazy OID allocation requires that the GRT can detect when an object reference is about to be exported from an extent, context, or cluster as appropriate. This means that lazy OID allocation is only possible if the GRT supports swizzling and may additionally require an address space scan [22].

A GRT object to which no OID has been allocated is known as an *immature* object. By definition immature objects exist and are known only within the extent in which they were created. When allocated an OID, an object is said to be *promoted* to being a *mature* object.

The gREFs provided by a GRT serve not only to allow the referenced object to be located but are also used to support object fault handling mechanisms. For example, as well as providing the target object's OID or storage identifier, a gREF might contain information to allow a proxy for the object to be created when required.

In addition, since most GRTs will support embedded language objects within a GRT object, a gREF may refer to a particular offset within a GRT object. This is useful where a gREF is to be converted to an lREF referring to such an embedded language object rather than its enclosing language object.

Code Management The code to be bound to each language object is provided by its LSRT as a *class*. A given type may be represented by one or more classes. For example, if the LSRT uses proxies for object fault detection, then every type may be represented by a real class bound to language objects of that type and a proxy class bound to proxies for objects of that type. Each class consists of *application code*, which implements the methods required by the object's type, and *upcall code*, which implements the upcall methods to be bound to GRT objects containing objects of that type[6]. As mentioned previously, the upcall code is bound to the appropriate GRT object by the GRT while the application code is bound to the language object in a language-specific way by its LSRT, usually in response to an upcall from the GRT. Note however that only a single set of upcall methods can be associated with each GRT object.

Each class is represented by a *class descriptor* and named by a *class identifier* that acts as an index for the class descriptor in the GRT's *class register* (CR).

Objects and Representatives A distributed or persistent language object can have *representatives* in many contexts. The representatives of an object might be used to implement an object and its proxies, the replicas of a replicated object, or the fragments of a fragmented object. The mapping of a distributed or persistent language object onto a set of representatives is thus language-specific. Moreover, depending on the object model supported by the language, the existence of multiple representatives of an object in the system may or may not be transparent to application programmers.

To support this model, a GRT object can likewise have representatives in many contexts. The representatives of a GRT object share its identity. However,

[6] Upcall code may be specific to one type or shared between different types, for example, Eiffel** uses the same upcall code for all types.

the representatives may be different sizes and may or may not have application code bound to them. Moreover, the code bound to each representative may be the same or different. All representatives of a GRT object do however have GRT object headers and all have (possibly different) upcall code bound to them. If, when, and how representatives for GRT objects are created depends on the GRT instantiation. For example, to support a language that uses proxies for object fault detection, a GRT might be instantiated that creates representatives for absent GRT objects that are the same size as the real object and have proxy application code bound to them. If the language uses descriptors to represent absent objects, the GRT instantiation might create representatives for absent objects that are smaller that the actual object and have no application code bound to them.

When the GRT creates or maps an object or a representative for an object, such as a proxy, the GRT will ask the LSRT to *prepare* the object/representative for possible accesses by its clients by making an upcall to the object/representative. This upcall allows the LSRT to carry out any appropriate language-specific actions necessary to make the object/representative ready to be accessed. Typically, this will include binding application code to the object/representative but may also involve initiating swizzling or doing other format conversions which are necessary prior to the object/representative being accessed. Thus, initiating swizzling is the responsibility of the LSRT and not the GRT. When exactly the GRT makes this upcall depends on the particular GRT instantiation.

7 The Organisation of the Owl Class Category

A GRT consists of a number of major functional components that can be individually customised to implement a GRT providing some required set of object-support mechanisms and policies. The seven major components of all GRTs are illustrated in figure 3 along with one optional component. These major components are implemented by instances of classes derived from the major class hierarchies that make up the Owl class category. Other Owl class hierarchies describe GRT objects, clusters, and various support classes used by the major components of a GRT.

The main interface between an LSRT and a GRT is provided by an instance of **`OwlGRT`**$_{sc}$[7]. Subclasses of **`OwlGRT`** provide the major GRT methods related to object (and cluster) management callable from LSRTs and are also responsible for the translation between lREFs and gREFs that takes place at the LSRT/GRT interface.

Every GRT has a heap in which objects are created and mapped as required. A GRT's heap is implemented by an instance of **`OwlHeap`**$_{sc}$ that provides the methods to allocate and deallocate memory from the heap. Higher-level methods, such as those to create objects and clusters within the heap or those to map and unmap objects and clusters into and out of the heap, are provided by

[7] The notation "`ClassName`$_{sc}$" is used to denote subclasses of `ClassName`, i.e., "`ClassName`$_{sc}$" can be read as "one of the subclasses of `ClassName`".

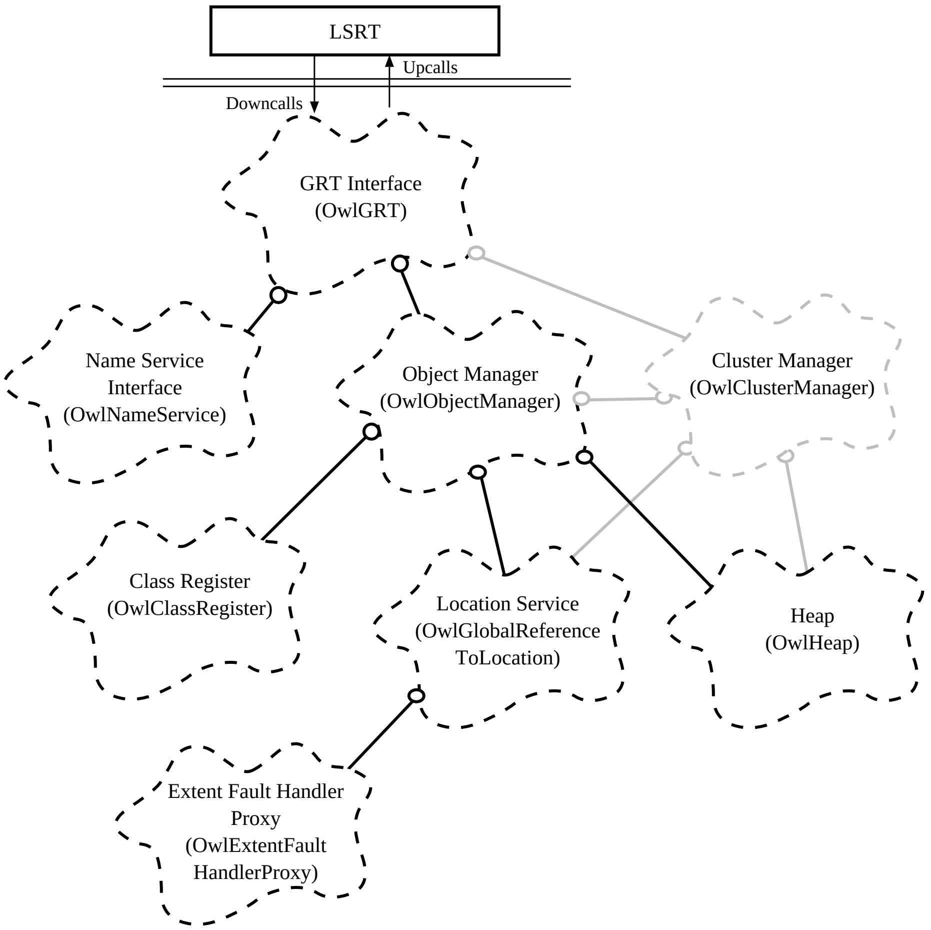

Fig. 3. The major components of a GRT and their main using relationships.

a heap manager – an instance of `OwlManager`$_{\mathrm{sc}}$. Heap managers come in two varieties: object managers (OMs) and cluster managers (CLMs). OMs – instances of `OwlObjectManager`$_{\mathrm{sc}}$ – provide methods related to the creation, mapping, and unmapping of objects, while CLMs – instances of `OwlClusterManager`$_{\mathrm{sc}}$ – provide methods related to the creation, mapping, and unmapping of clusters. Every GRT has an OM. A GRT that supports application-directed clustering will also have a CLM. Thus, as indicated by the shaded lines in figure 3, a CLM is an optional component of a GRT. The OM or CLM is also the component of the GRT that interacts with Eeyore – the storage class category – to store and retrieve objects or clusters respectively when required.

While heap managers are responsible for control of the heap, the location of, and, where necessary, forwarding of access requests to absent objects (be they persistent objects stored in the POS, distributed objects located on another node, or objects belonging to a different extent) is encapsulated within the location service (LS) component of the GRT, which is implemented by an instance of **`OwlGlobalReferenceToLocation`**$_{SC}$. The LS implements the GRT's mapping from the gREF for an object to its current location in the (possibly distributed) system. Since an absent object reported to the LS may actually be non-existent or, in a multi-extent Tigger instantiation, belong to a different extent, the LS is also responsible for raising extent faults. In a GRT supporting distribution, the LS is a distributed component and uses Kanga – the communications class category – for communication between its distributed parts.

Every GRT has a proxy for its local EFH, which is an instance of **`OwlExtentFaultHandlerProxy`**$_{SC}$. Thus, instances of **`OwlExtentFaultHandlerProxy`**$_{SC}$ are kernel-aware objects that allow cross-extent object invocation to be implemented.

A CR is a repository for class descriptors and code. Every GRT uses a CR – an instance of **`OwlClassRegister`**$_{SC}$ – to obtain the class code for new and recently mapped objects when required. A CR is normally persistent and may also be remotely accessible. Likewise, the objects that it uses to store classes and their code would normally be expected to be persistent. Thus, a CR represents a good example of a service provided by the Tigger framework that is itself implemented using distributed and persistent objects. The design of the Tigger framework assumes that there is a single CR in each system, which is shared between all the GRTs (and all the extents) in that system. It is worth noting that although the CR is a trusted service, it can belong to any desired extent.

Finally, every GRT also provides a NS to supported languages via an instance of **`OwlNameService`**$_{SC}$. Although instances of **`OwlNameService`**$_{SC}$ are local volatile objects that are private to one GRT, the directories to which they refer are typically implemented by distributed persistent objects. Thus the NS as a whole can be seen as another example of a service provided by the Tigger framework that is itself implemented using distributed and persistent objects. Moreover, individual directories may belong to different extents.

In addition to the class hierarchies describing the main components of the GRT, further class hierarchies describe objects and clusters. The `OwlObject` class hierarchy describes the methods supported by GRT objects and the structure of GRT object headers. The `OwlObject` hierarchy also describes the upcalls that must be provided for each object by the LSRT and provides the means of binding the upcall code to a GRT object/representative. Similarly, the `OwlCluster` class hierarchy describes the methods supported by clusters. In addition, Owl includes a number of other important class hierarchies that are introduced briefly here.

- `OwlLanguageReference` Describes the protocol to be supported by lREFs.
- `OwlGlobalReference` Describes the protocols supported by gREFs.
- `OwlGlobalReferenceToAddress` Describes the GRT's mapping from a gREF for an object to its address in the current context.
- `OwlClusterIdentifier` Describes the protocol for cluster identifiers.

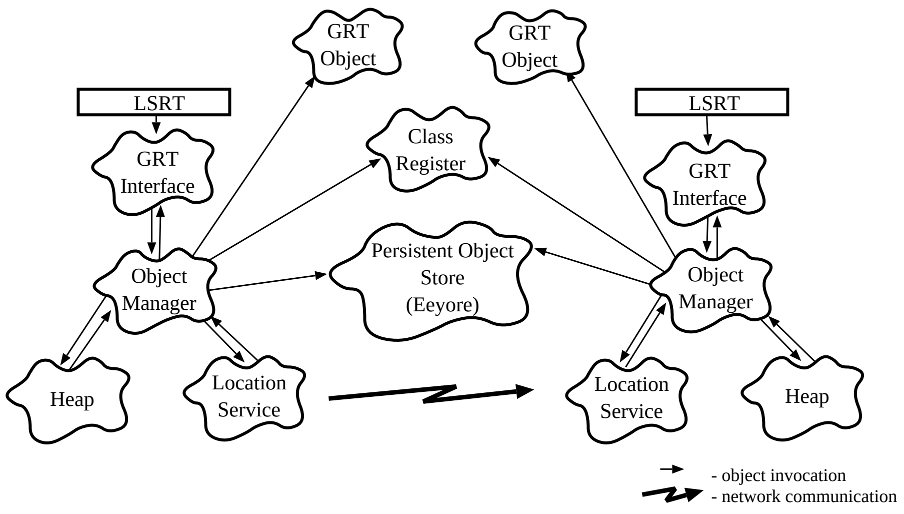

Fig. 4. The interactions between the major components of a GRT.

- `OwlClusterIdentifierToAddress` Describes the GRT's mapping from the identifier of a cluster to its address in the current context.
- `OwlMarshalStream` Provides methods for constructing messages including ROI request messages and replies.
- `OwlRequestDescriptor`, `OwlRPCDescriptor`, and `OwlMigrationDescriptor` describe messages sent by GRT components that are constructed by LSRTs.
- `OwlDirectory` Describes the interface to an NS directory.
- `OwlDirectoryEntry` Describes an entry in an NS directory.
- `OwlCode` Describes objects used to store executable code.
- `OwlClassDescriptor` Describes a class descriptor.

It should be understood that above list is not exhaustive and that other classes are required to implement a GRT. Those presented typically use the services of other simpler classes describing their internal data structures or providing "house-keeping" functionality.

Interactions Between GRT Components Figure 4 shows the main interactions that occur between the major components of a GRT. For the sake of generality, the GRT in question is assumed to support both distribution and persistence and is hence distributed over multiple nodes and makes use of a POS instantiated from Eeyore.

The LSRTs of supported languages usually invoke methods provided by the GRT interface. This will typically result in the GRT interface invoking one of the other components of the GRT, normally the NS interface or OM. In the case where the request from the LSRT is related to object management (for example, requests to create or delete objects and requests related to object faulting), the GRT interface calls the OM. The OM will typically use the services of the LS, the heap, or the POS to carry out the request. During object fault handling, the request is typically forwarded to the LS. The LS may indicate that the object should be retrieved from the POS and mapped locally, return the object immediately, forward the request to the OM at the node where the object is located, or raise an extent fault if the object may belong to a different extent. In handling the request, the LS will typically communicate with its remote peers who may, in turn, need to upcall their local OMs. Thus, an OM typically provides a downcall interface for use by the GRT interface and an upcall interface for use by the LS during object fault handling. Like the interface to the GRT, the interfaces to both the OM and LS must be specialised depending on the approach to object faulting supported. In addition, as a heap manager, the OM also provides an upcall interface for use by the heap when heap space is exhausted. This interface typically causes the OM to try to unmap some objects. The OM may use the CR to load class code for newly created objects or objects that have been mapped recently and is also the component that most commonly makes upcalls to GRT objects. Finally, the OM may upcall the GRT interface – usually to convert a gREF to an lREF or vice versa.

Both the CR and POS are potentially shared by different GRTs in different extents including GRTs of different types. Moreover, they are typically implemented by distributed objects and are accessible from multiple GRTs using location-transparent object invocation.

Figure 5 shows the interactions that occur between the major components of a GRT that supports application-directed clustering. Such a GRT has an additional component, its CLM, that is interposed between the OM and other components such as the heap, LS, and POS. Requests related to clusters (for example, requests to create or delete clusters) are passed by the GRT interface directly to the CLM while requests related to objects are still passed to the OM. A request concerning some object might result in the OM making a corresponding request to the CLM for that object's cluster. Since the unit of location, mapping, and unmapping is a cluster rather than an individual object, the CLM is responsible for interacting with the LS, POS, and heap to resolve the request in much the same way as the the OM is in a GRT that does not support application-directed clustering. Resolving the request might require that a CLM make an upcall to its local OM. Like the interfaces to the GRT interface, OM, and LS, the interface to the CLM is also specialised depending on the approach to object faulting supported.

8 Status

At the current time, the design of the first complete version of the framework has been completed and a number of instantiations are being implemented. [23] describes the first Tigger instantiation implemented. The so-called T1 instantiation supports an extension to C++ for distributed and persistent programming

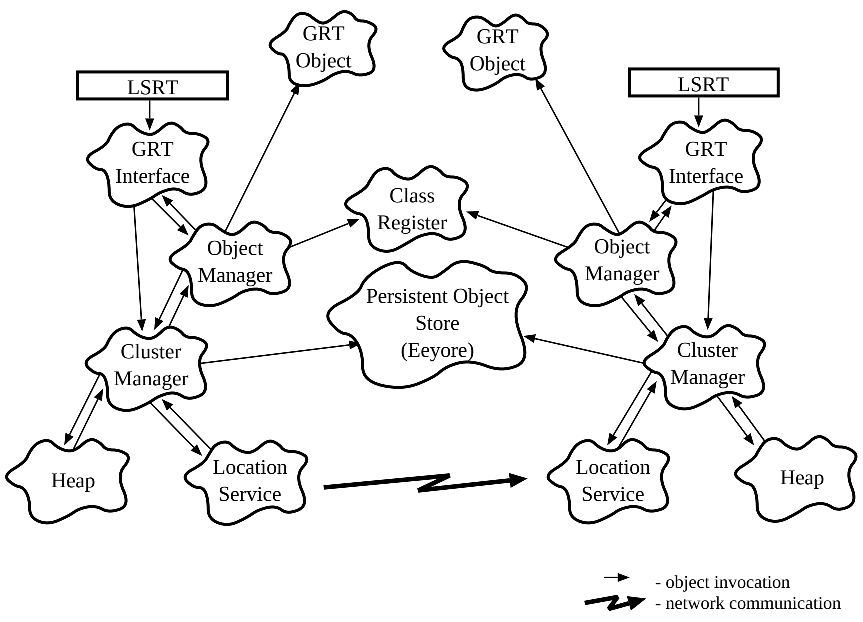

Fig. 5. Interactions between parts of a GRT supporting application-directed clustering.

inspired by PANDA/C++ [1]. T1 is a single-extent Tigger instantiation layered above UNIX that implements a single distributed and persistent address space and supports DSM-style access to global and persistent objects. Object faults are detected as memory protection faults. No swizzling is employed and virtual addresses are used as gREFs. In addition, T1 supports application-directed clustering.

Another Tigger instantiation is currently being implemented to support a novel object model providing application-consistent DSM [10].

9 Summary and Conclusions

An object-support operating system may be described as one that has been designed specifically to support object-oriented applications, especially distributed applications or those that manipulate persistent data. Unfortunately, most existing object-support operating systems can support only a single language or else severely constrain the way in which different languages can be supported, in particular, by supporting only a single object model. In contrast, the Tigger project undertook the design, not of a single object-support operating system, but of a family of object-support operating systems whose members can be customised for use in a variety of different application domains. The two primary goals of this design were to allow members of the family to support (a number of) different object models and to allow the same object model to be supported in different ways subject to differing non-functional requirements. This design is captured as a framework that can be instantiated to implement the individual members of the family.

While framework technology is well-established and the use of frameworks to implement customised operating systems is not new, the use of a framework as the basis for implementing customised object-support operating systems is novel. The Tigger framework provides a common basis for the implementation of both single and multi-user object-support operating systems that support a range of object-oriented programming languages for distributed and persistent programming and encompass different non-functional requirements such as heterogeneity or protection. While traditional operating system architectures emphasise the distinction between the operating system kernel, which runs in supervisor mode, and user-level servers and applications, which do not, the design of Tigger emphasises the orthogonality between protection and operating system structure. Thus, the resulting framework encompasses both single-user systems with no kernel and multi-user systems having a distinguished kernel.

Acknowlegdements

Thanks to Brendan Tangney, Neville Harris, Paul Taylor, and Alan Judge for their many and varied contributions to the work described in this paper.

References

1. Holger Assenmacher, Thomas Breitbach, Peter Buhler, Volker Huebsch, and Reinhard Schwarz. PANDA - supporting distributed programming in C++. In Oscar M. Nierstrasz, editor, *Proceedings of the* 7^{th} *European Conference on Object-Oriented Programming*, volume 707 of *Lecture Notes in Computer Science*, pages 361–383. Springer-Verlag, 1993.
2. Grady Booch. *Object-Oriented Analysis and Design with Applications.* Benjamin/Cummings, Redwood City, CA, 1994.
3. Vinny Cahill. On The Architecture of a Family of Object-Support Operating Systems. Ph.D. thesis, Department of Computer Science, Trinity College Dublin, September 1996.
4. Vinny Cahill, Seán Baker, Gradimir Starovic, and Chris Horn. Generic runtime support for distributed persistent programming. In Paepcke [18], pages 144–161.
5. Vinny Cahill, Roland Balter, Xavier Rousset de Pina, and Neville Harris, editors. *The COMANDOS Distributed Application Platform.* ESPRIT Research Reports Series. Springer-Verlag, 1993.
6. Roy H. Campbell, Nayeem Islam, and Peter Madany. *Choices*, Frameworks and Refinement. *Computing Systems*, 5(3):217–257, Summer 1992.
7. Distributed Systems Group. C** programmer's guide (Amadeus v2.0). Technical Report TCD-CS-92-03, Department of Computer Science, Trinity College Dublin, February 1992.

8. Christine Hogan. The Tigger Cub Nucleus. Master's thesis, Department of Computer Science, Trinity College Dublin, September 1994.
9. Chris Horn and Vinny Cahill. Supporting distributed applications in the Amadeus environment. *Computer Communications*, 14(6):358–365, July/August 1991.
10. Alan Judge. *Supporting Application-Consistent Distributed Shared Objects.* PhD thesis, Department of Computer Science, Trinity College Dublin, 1996. In preparation.
11. Reinhold Kröeger, Michael Mock, Ralf Schumann, and Frank Lange. RelaX - an extensible architecture supporting reliable distributed applications. In *Proceedings of the* 9^{th} *Symposium on Reliable Distributed Systems*, pages 156–165. IEEE Computer Society Press, 1990.
12. John McEvoy. E**: Porting the E database language to Amadeus. Master's thesis, Department of Computer Science, Trinity College Dublin, 1993.
13. Colm McHugh and Vinny Cahill. Eiffel**: An implementation of Eiffel on Amadeus, a persistent, distributed applications support environment. In Boris Magnusson, Bertrand Meyer, and Jean-Francois Perot, editors, *Technology of Object-Oriented Languages and Systems (TOOLS 10)*, pages 47–62. Prentice Hall, 1993.
14. Michael Mock, Reinhold Kroeger, and Vinny Cahill. Implementing atomic objects with the RelaX transaction facility. *Computing Systems*, 5(3):259–304, 1992.
15. Karl O'Connell, Vinny Cahill, Andrew Condon, Stephen McGerty, Gradimir Starovic, and Brendan Tangney. The VOID shell: A toolkit for the development of distributed video games and virtual worlds. In *Proceedings of the Workshop on Simulation and Interaction in Virtual Environments*, 1995.
16. Darragh O'Grady. An extensible, high-performance, distributed persistent store for Amadeus. Master's thesis, Department of Computer Science, Trinity College Dublin, September 1994.
17. Joo Li Ooi. Access control for an object-oriented distributed platform. Master's thesis, Department of Computer Science, Trinity College Dublin, August 1993.
18. Andreas Paepcke, editor. *Proceedings of the 1993 Conference on Object-Oriented Programming Systems, Languages and Applications.* ACM Press, September 1993. Also SIGPLAN Notices 28(10), October 1993.
19. Wolfgang Schröder-Preikschat. *The Logical Design of Parallel Operating Systems.* Prentice Hall, London, 1994.
20. Marc Shapiro. Object-support operating systems. *IEEE Technical Committee on Operating Systems and Application Environments Newsletter*, 5(1):39–42, Spring 1991.
21. Marc Shapiro. Soul: An object-oriented OS framework for object support. In A. Karshmer and J. Nehmer, editors, *Operating Systems of the 90s and Beyond*, volume 563 of *Lecture Notes in Computer Science.* Springer-Verlag, July 1991.
22. Pedro Sousa, Manuel Sequeira, André Zúquete, Paulo Ferreira, Cristina Lopes, José Pereira, Paulo Guedes, and José Alves Marques. Distribution and persistence in the IK platform: Overview and evaluation. *Computing Systems*, 6(4):391–424, Fall 1993.
23. Paul Taylor. The T1 cub. Tigger document T16-94, Distributed Systems Group, Department of Computer Science, Trinity College Dublin, November 1994.
24. Paul Taylor, Vinny Cahill, and Michael Mock. Combining object-oriented systems and open transaction processing. *The Computer Journal*, 37(6), August 1994.

This article was processed using the LaTeX macro package with LLNCS style